State Synchronization

Keeping simulations in sync by sending state

Posted by Glenn Fiedler (http://web.archive.org/web/20181107181459/https://gafferongames.com/about) on Monday, January 5, 2015

Introduction

Hi, I'm Glenn Fiedler (http://web.archive.org/web/20181107181459/https://gafferongames.com/about) and welcome to **Networked Physics (http://web.archive.org/web/20181107181459/https://gafferongames.com/categories/networked-physics/)**.

In the previous article (http://web.archive.org/web/20181107181459/https://gafferongames.com/post/snapshot_compression/) we discussed techniques for compressing snapshots.

In this article we round out our discussion of networked physics strategies with **state synchronization**, the third and final strategy in this article series.

State Synchronization

What is state synchronization? The basic idea is that, somewhat like deterministic lockstep, we run the simulation on both sides but, *unlike* deterministic lockstep, we don't just send input, we send both input and state.

This gives state synchronization interesting properties. Because we send state, we don't need perfect determinism to stay in sync, and because the simulation runs on both sides, objects continue moving forward between updates.

This lets us approach state synchronization differently to snapshot interpolation. Instead of sending state updates for every object in each packet, we can now send updates for only a few, and if we're smart about how we select the objects for each packet, we can save bandwidth by concentrating updates on the most important objects.

So what's the catch? State synchronization is an approximate and lossy synchronization strategy. In practice, this means you'll spend a lot of time tracking down sources of extrapolation divergence and pops. But other than that, it's a quick and easy strategy to get started with.

Implementation

Here's the state sent over the network per-object:

```
    vec3f position;
    quat4f orientation;
    vec3f linear_velocity;
    vec3f angular_velocity;
};
```

Unlike snapshot interpolation, we're not just sending visual quantities like position and orientation, we're also sending *non-visual* state such as linear and angular velocity. Why is this?

The reason is that state synchronization runs the simulation on both sides, so it's *always extrapolating* from the last state update applied to each object. If linear and angular velocity aren't synchronized, this extrapolation is done with incorrect velocities, leading to pops when objects are updated.

While we must send the velocities, there's no point wasting bandwidth sending (0,0,0) over and over while an object is at rest. We can fix this with a trivial optimization, like so:

```
void serialize_state_update( Stream & stream,
                             int & index,
                             StateUpdate & state_update )
{
    serialize_int( stream, index, 0, NumCubes - 1 );
    serialize_vector( stream, state_update.position );
    serialize_quaternion( stream, state_update.orientation );
    bool at_rest = stream.IsWriting() ? state_update.AtRest() : false;
    serialize_bool( stream, at_rest );
    if ( !at_rest )
    {
        serialize_vector( stream, state_update.linear_velocity );
        serialize_vector( stream, state_update.angular_velocity );
    }
    else if ( stream.IsReading() )
    {
        state_update.linear_velocity = vec3f(0,0,0);
        state_update.angular_velocity = vec3f(0,0,0);
    }
}
```

What you see above is a *serialize function*. It's a trick I like to use to unify packet read and write. I like it because it's expressive while at the same time it's difficult to desync read and write. You can read more about them here (http://web.archive.org/web/20181107181459/https://gafferongames.com/post/serialization_strategies/).

Packet Structure

Now let's look at the overall structure of packets being sent:

```
const int MaxInputsPerPacket = 32;
const int MaxStateUpdatesPerPacket = 64;

struct Packet
{
    uint32_t sequence;
    Input inputs[MaxInputsPerPacket];
    int num_object_updates;
    StateUpdate state_updates[MaxStateUpdatesPerPacket];
};
```

First we include a sequence number in each packet so we can determine out of order, lost or duplicate packets. I recommend you run the simulation at the same framerate on both sides (for example 60HZ) and in this case the sequence number can work double duty as the frame number.

deterministic lockstep, if don't have the next input we don't stop the simulation and wait for it, we continue extrapolating forward with the last input received.

Next you can see that we only send a maximum of 64 state updates per-packet. Since we have a total of 901 cubes in the simulation so we need some way to select the n most important state updates to include in each packet. We need some sort of prioritization scheme.

To get started each frame walk over all objects in your simulation and calculate their current priority. For example, in the cube simulation I calculate priority for the player cube as 1000000 because I always want it to be included in every packet, and for interacting (red cubes) I give them a higher priority of 100 while at rest objects have priority of 1.

Unfortunately if you just picked objects according to their current priority each frame you'd only ever send red objects while in a katamari ball and white objects on the ground would never get updated. We need to take a slightly different approach, one that prioritizes sending important objects while also *distributing* updates across all objects in the simulation.

Priority Accumulator

You can do this with a priority accumulator. This is an array of float values, one value per-object, that is remembered from frame to frame. Instead of taking the immediate priority value for the object and sorting on that, each frame we add the current priority for each object to its priority accumulator value then sort objects in order from largest to smallest priority accumulator value. The first n objects in this sorted list are the objects you should send that frame.

You could just send state updates for all n objects but typically you have some maximum bandwidth you want to support like 256kbit/sec. Respecting this bandwidth limit is easy. Just calculate how large your packet header is and how many bytes of preamble in the packet (sequence, # of objects in packet and so on) and work out conservatively the number of bytes remaining in your packet while staying under your bandwidth target.

Then take the n most important objects according to their priority accumulator values and as you construct the packet, walk these objects in order and measure if their state updates will fit in the packet. If you encounter a state update that doesn't fit, skip over it and try the next one. After you serialize the packet, reset the priority accumulator to zero for objects that fit but leave the priority accumulator value alone for objects that didn't. This way objects that don't fit are first in line to be included in the next packet.

The desired bandwidth can even be adjusted on the fly. This makes it really easy to adapt state synchronization to changing network conditions, for example if you detect the connection is having difficulty you can reduce the amount of bandwidth sent (congestion avoidance) and the quality of state synchronization scales back automatically. If the network connection seems like it should be able to handle more bandwidth later on then you can raise the bandwidth limit.

Jitter Buffer

The priority accumulator covers the sending side, but on the receiver side there is much you need to do when applying these state updates to ensure that you don't see divergence and pops in the extrapolation between object updates.

world is you'll typically receive two packets one frame, 0 packets the next, 1, 2, 0 and so on because packets tend to clump up across frames. To handle this situation you need to implement a jitter buffer for your state update packets. If you fail to do this you'll have a poor quality extrapolation and pops in stacks of objects because objects in different state update packets are slightly out of phase with each other with respect to time.

All you do in a jitter buffer is hold packets before delivering them to the application at the correct time as indicated by the sequence number (frame number) in the packet. The delay you need to hold packets for in this buffer is a much smaller amount of time relative to interpolation delay for snapshot interpolation but it's the same basic idea. You just need to delay packets just enough (say 4-5 frames @ 60HZ) so that they come out of the buffer properly spaced apart.

Applying State Updates

Once the packet comes out of the jitter how do you apply state updates? My recommendation is that you should snap the physics state hard. This means you apply the values in the state update directly to the simulation.

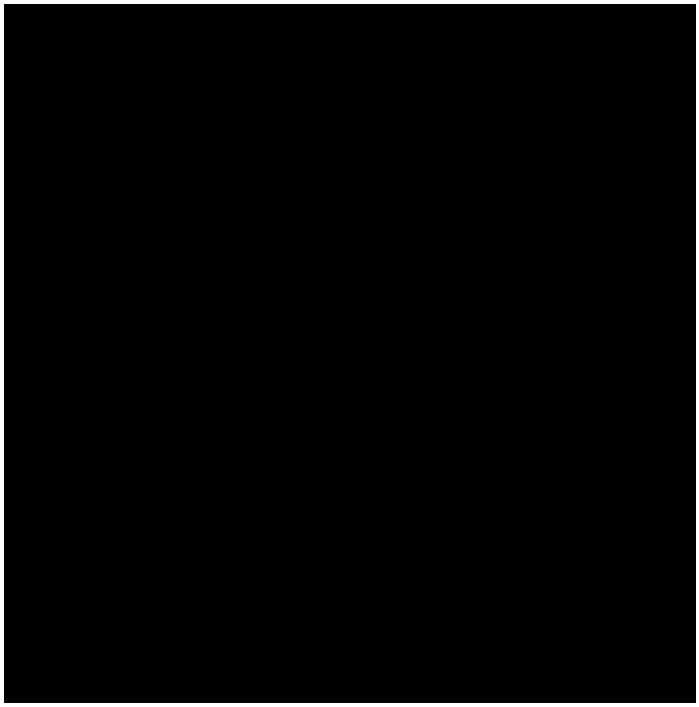
I recommend against trying to apply some smoothing between the state update and the current state at the simulation level. This may sound counterintuitive but the reason for this is that the simulation extrapolates from the state update so you want to make sure it extrapolates from a valid physics state for that object rather than some smoothed, total bullshit made-up one. This is especially important when you are networking large stacks of objects.

Surprisingly, without any smoothing the result is already pretty good:

As you can see it's already looking quite good and barely any bandwidth optimization has been performed. Contrast this with the first video for snapshot interpolation which was at 18mbit/sec and you can see that using the simulation to extrapolate between state updates is a great way to use less bandwidth.

compression techniques such as bounding and quantizing position, linear and angular velocity value and using the smallest three compression as described in snapshot compression (http://web.archive.org/web/20181107181459/https://gafferongames.com/post/snapshot_compression/).

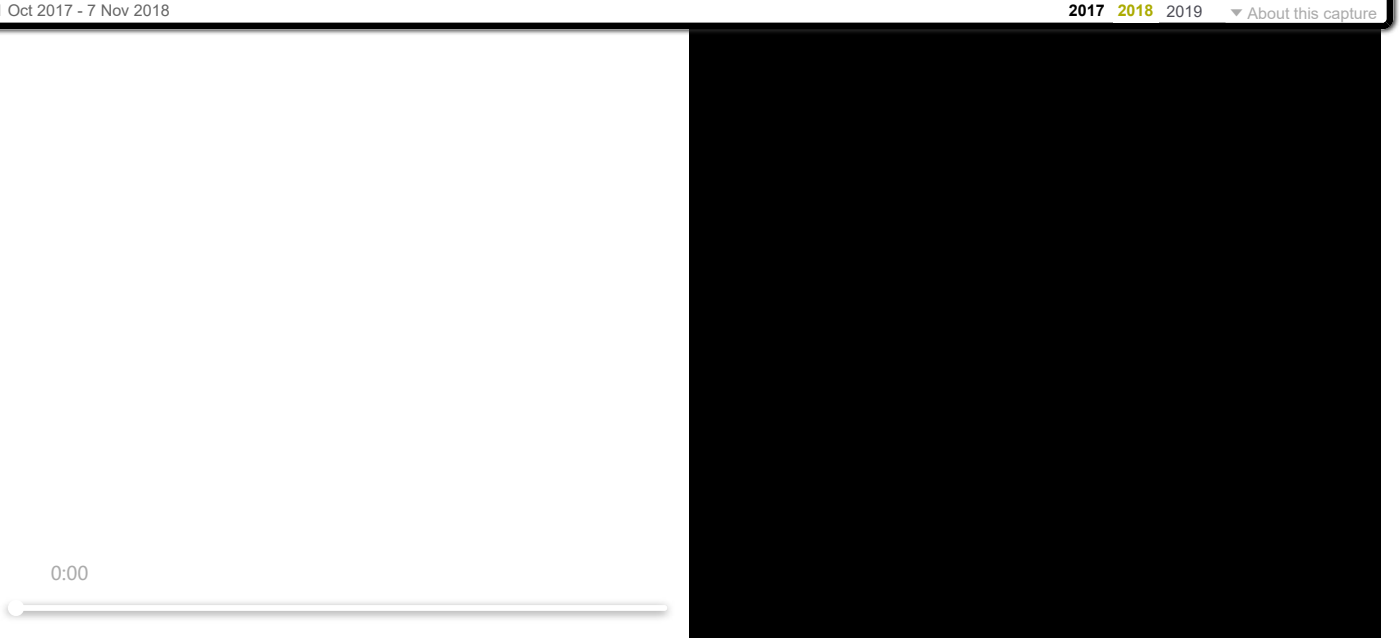
But here it gets a bit more complex. We are extrapolating from those state updates so if we quantize these values over the network then the state that arrives on the right side is slightly different from the left side, leading to a slightly different extrapolation and a pop when the next state update arrives for that object.



Quantize Both Sides

The solution is to quantize the state on both sides. This means that on both sides before each simulation step you quantize the entire simulation state as if it had been transmitted over the network. Once this is done the left and right side are both extrapolating from quantized state and their extrapolations are very similar.

Because these quantized values are being fed back into the simulation, you'll find that much more precision is required than snapshot interpolation where they were just visual quantities used for interpolation. In the cube simulation I found it necessary to have 4096 position values per-meter, up from 512 with snapshot interpolation, and a whopping 15 bits per-quaternion component in smallest three (up from 9). Without this extra precision significant popping occurs because the quantization forces physics objects into penetration with each other, fighting against the simulation which tries to keep the objects out of penetration. I also found that softening the constraints and reducing the maximum velocity which the simulation used to push apart penetrating objects also helped reduce the amount of popping.



With quantization applied to both sides you can see the result is perfect once again. It may look visually about the same as the uncompressed version but in fact we're able to fit many more state updates per-packet into the 256kbit/sec bandwidth limit. This means we are better able to handle packet loss because state updates for each object are sent more rapidly. If a packet is lost, it's less of a problem because state updates for those objects are being continually included in future packets.

Be aware that when a burst of packet loss occurs like $^1/_4$ a second with no packets getting through, and this is inevitable that eventually something like this will happen, you will probably get a different result on the left and the right sides. We have to plan for this. In spite of all effort that we have made to ensure that the extrapolation is as close as possible (quantizing both sides and so on) pops can and will occur if the network stops delivering packets.

Visual Smoothing

We can cover up these pops with smoothing.

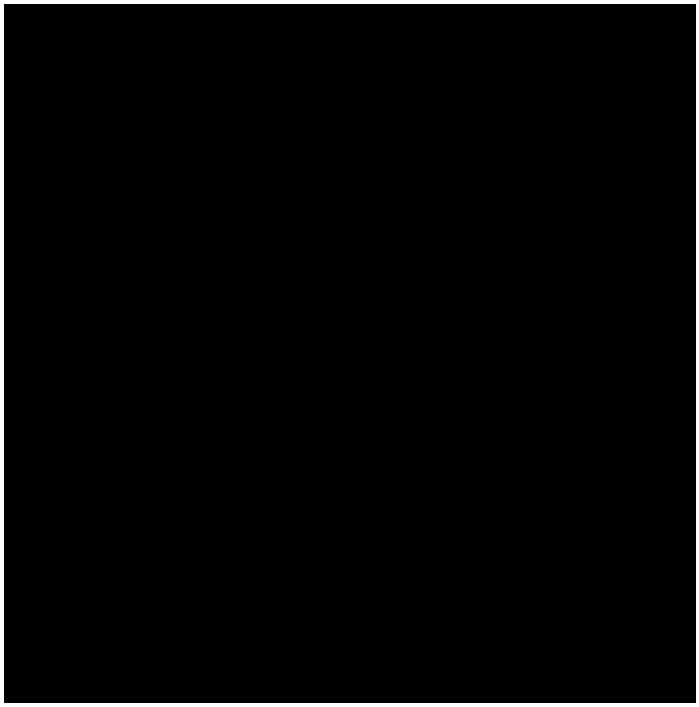
Remember how I said earlier that you should not apply smoothing at the simulation level because it ruins the extrapolation? What we're going to do for smoothing instead is calculating and maintaining position and orientation error offsets that we reduce over time. Then when we render the cubes in the right side we don't render them at the simulation position and orientation, we render them at the simulation position + error offset, and orientation * orientation error.

Over time we work to reduce these error offsets back to zero for position error and identity for orientation error. For error reduction I use an exponentially smoothed moving average tending towards zero. So in effect, I multiply the position error offset by some factor each frame (eg. 0.9) until it gets close enough to zero for it to be cleared (thus avoiding denormals). For orientation, I slerp a certain amount (0.1) towards identity each frame, which has the same effect for the orientation error.

The trick to making this all work is that when a state update comes in you take the current simulation position and add the position error to that, and subtract that from the new position, giving the new position error offset which gives an identical result to the current (smoothed) visual position.

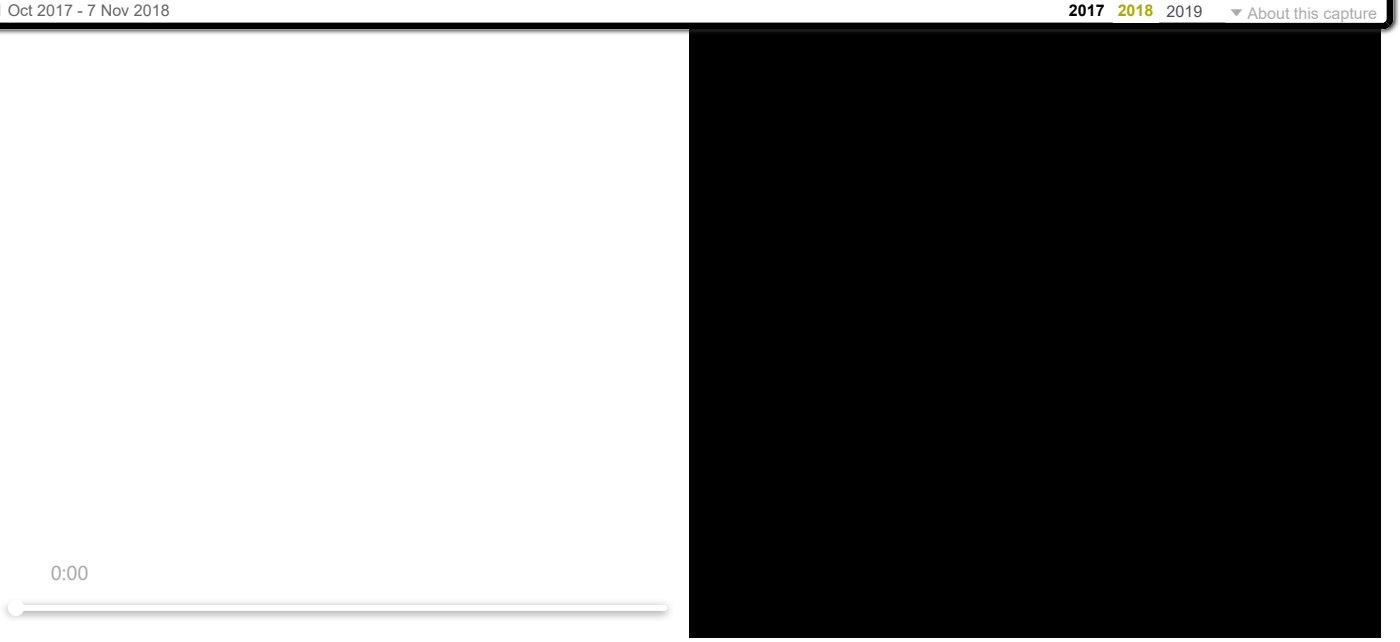
error relative to the new state such that the object appears to have not moved at all. Thus state updates are smooth and have no immediate visual effect, and the error reduction smoothes out any error in the extrapolation over time without the player noticing in the common case.

I find that using a single smoothing factor gives unacceptable results. A factor of 0.95 is perfect for small jitters because it smooths out high frequency jitter really well, but at the same time it is too slow for large position errors, like those that happen after multiple seconds of packet loss:



The solution I use is two different scale factors at different error distances, and to make sure the transition is smooth I blend between those two factors linearly according to the amount of positional error that needs to be reduced. In this simulation, having 0.95 for small position errors (25cms or less) while having a tighter blend factor of 0.85 for larger distances (1m error or above) gives a good result. The same strategy works well for orientation using the dot product between the orientation error and the identity matrix. I found that in this case a blend of the same factors between dot 0.1 and 0.5 works well.

The end result is smooth error reduction for small position and orientation errors combined with a tight error reduction for large pops. As you can see above you don't want to drag out correction of these large pops, they need to be fast and so they're over quickly otherwise they're really disorienting for players, but at the same time you want to have really smooth error reduction when the error is small hence the adaptive error reduction approach works really well.



Delta Compression

Even though I would argue the result above is probably good enough already it is possible to improve the synchronization considerably from this point. For example to support a world with larger objects or more objects being interacted with. So lets work through some of those techniques and push this technique as far as it can go.

There is an easy compression that can be performed. Instead of encoding absolute position, if it is within a range of the player cube center, encode position as a relative offset to the player center position. In the common cases where bandwidth is high and state updates need to be more frequent (katamari ball) this provides a large win.

Next, what if we do want to perform some sort of delta encoding for state synchronization? We can but it's quite different in this case than it is with snapshots because we're not including every cube in every packet, so we can't just track the most recent packet received and say, OK all these state updates in this packet are relative to packet X.

What you actually have to do is per-object update keep track of the packet that includes the base for that update. You also need to keep track of exactly the set of packets received so that the sender knows which packets are valid bases to encode relative to. This is reasonably complicated and requires a bidirectional ack system over UDP. Such a system is designed for exactly this sort of situation where you need to know exactly which packets definitely got through. You can find a tutorial on how to implement this in this article (http://web.archive.org/web/20181107181459/https://gafferongames.com/post/reliability_and_flow_control/).

So assuming that you have an ack system you know with packet sequence numbers get through. What you do then is per-state update write one bit if the update is relative or absolute, if absolute then encode with no base as before, otherwise if relative send the 16 bit sequence number per-state update of the base and then encode relative to the state update data sent in that packet. This adds 1 bit overhead per-update as well as 16 bits to identify the sequence number of the base per-object update. Can we do better?

Yes. In turns out that of course you're going to have to buffer on the send and receive side to implement this relative encoding and you can't buffer forever. In fact, if you think about it you can only buffer up a couple of seconds before it becomes impractical and in the common case of moving objects you're going to be sending

So instead of sending the 16 bit sequence base per-object, send in the header of the packet the most recent acked packet (from the reliability ack system) and per-object encode the offset of the base sequence relative to that value using 5 bits. This way at 60 packets per-second you can identify an state update with a base half a second ago. Any base older than this is unlikely to provide a good delta encoding anyway because it's old, so in that case just drop back to absolute encoding for that update.

Now lets look at the type of objects that are going to have these absolute encodings rather than relative. They're the objects at rest. What can we do to make them as efficient as possible? In the case of the cube simulation one bad result that can occur is that a cube comes to rest (turns grey) and then has its priority lowered significantly. If that very last update with the position of that object is missed due to packet loss, it can take a long time for that object to have its at rest position updated.

We can fix this by tracking objects which have recently come to rest and bumping their priority until an ack comes back for a packet they were sent in. Thus they are sent at an elevated priority compared with normal grey cubes (which are at rest and have not moved) and keep resending at that elevated rate until we know that update has been received, thus "committing" that grey cube to be at rest at the correct position.

Conclusion

And that's really about it for this technique. Without anything fancy it's already pretty good, and on top of that another order of magnitude improvement is available with delta compression, at the cost of significant complexity!