Snapshot Compression

Advanced techniques for optimizing bandwidth

Posted by Glenn Fiedler (http://web.archive.org/web/20181107181434/https://gafferongames.com/about) on Sunday, January 4, 2015

Introduction

Hi, I'm Glenn Fiedler (http://web.archive.org/web/20181107181434/https://gafferongames.com/about) and welcome to **Networked Physics (http://web.archive.org/web/20181107181434/https://gafferongames.com/categories/networked-physics/)**.

In the previous article (http://web.archive.org/web/20181107181434/https://gafferongames.com/post/snapshot_interpolation/) we sent snapshots of the entire simulation 10 times per-second over the network and interpolated between them to reconstruct a view of the simulation on the other side.

The problem with a low snapshot rate like 10HZ is that interpolation between snapshots adds interpolation delay on top of network latency. At 10 snapshots per-second, the minimum interpolation delay is 100ms, and a more practical minimum considering network jitter is 150ms. If protection against one or two lost packets in a row is desired, this blows out to 250ms or 350ms delay.

This is not an acceptable amount of delay for most games, but when the physics simulation is as unpredictable as ours, the only way to reduce it is to increase the packet send rate. Unfortunately, increasing the send rate also increases bandwidth. So what we're going to do in this article is work through every possible bandwidth optimization (*that I can think of at least*) until we get bandwidth under control.

Our target bandwidth is **256 kilobits per-second**.

Starting Point @ 60HZ

Life is rarely easy, and the life of a network programmer, even less so. As network programmers we're often tasked with the impossible, so in that spirit, let's increase the snapshot send rate from 10 to 60 snapshots per-second and see exactly how far away we are from our target bandwidth.

That's a LOT of bandwidth: **17.37 megabits per-second!**

Let's break it down and see where all the bandwidth is going.

Here's the per-cube state sent in the snapshot:

```
struct CubeState
{
    bool interacting;
    vec3f position;
    vec3f linear_velocity;
    quat4f orientation;
};
```

And here's the size of each field:

- quat orientation: **128 bits**
- vec3 linear_velocity: **96 bits**
- vec3 position: **96 bits**
- bool interacting: **1 bit**

This gives a total of 321 bits bits per-cube (or 40.125 bytes per-cube).

Let's do a quick calculation to see if the bandwidth checks out. The scene has 901 cubes so **901*40.125 = 36152.625** bytes of cube data per-snapshot. 60 snapshots per-second so **36152.625 * 60 = 2169157.5** bytes per-second. Add in packet header estimate: **2169157.5 + 32*60 = 2170957.5**. Convert bytes per-second to megabits per-second: **2170957.5 * 8 / (1000 * 1000) = 17.38mbps**.

Everything checks out. There's no easy way around this, we're sending a hell of a lot of bandwidth, and we have to reduce that to something around 1-2% of it's current bandwidth to hit our target of 256 kilobits per-second.

Is this even possible? *Of course it is*! Let's get started :)

Optimizing Orientation

Many people when compressing a quaternion think: "I know. I'll just pack it into 8.8.8.8 with one 8 bit signed integer per-component!". Sure, that works, but with a bit of math you can get much better accuracy with fewer bits using a trick called the "smallest three".

How does the smallest three work? Since we know the quaternion represents a rotation its length must be 1, so x^2+y^2+z^2+w^2 = 1. We can use this identity to drop one component and reconstruct it on the other side. For example, if you send x,y,z you can reconstruct w = sqrt(1 - x^2 - y^2 - z^2). You might think you need to send a sign bit for w in case it is negative, but you don't, because you can make w always positive by negating the entire quaternion if w is negative (in quaternion space (x,y,z,w) and (-x,-y,-z,-w) represent the same rotation.)

Don't always drop the same component due to numerical precision issues. Instead, find the component with the largest absolute value and encode its index using two bits [0,3] (0=x, 1=y, 2=z, 3=w), then send the index of the largest component and the smallest three components over the network (hence the name). On the other side use the index of the largest bit to know which component you have to reconstruct from the other three.

One final improvement. If v is the absolute value of the largest quaternion component, the next largest possible component value occurs when two components have the same absolute value and the other two components are zero. The length of that quaternion (v,v,0,0) is 1, therefore v^2 + v^2 = 1, 2v^2 = 1, v = 1/sqrt(2). This means you can encode the smallest three components in [-0.707107,+0.707107] instead of [-1,+1] giving you more precision with the same number of bits.

With this technique I've found that minimum sufficient precision for my simulation is 9 bits per-smallest component. This gives a result of 2 + 9 + 9 + 9 = 29 bits per-orientation (down from 128 bits).

This optimization reduces bandwidth by over 5 megabits per-second, and I think if you look at the right side, you'd be hard pressed to spot any artifacts from the compression.

Optimizing Linear Velocity

To compress linear velocity we need to bound its x,y,z components in some range so we don't need to send full float values. I found that a maximum speed of 32 meters per-second is a nice power of two and doesn't negatively affect the player experience in the cube simulation. Since we're really only using the linear velocity as a *hint* to improve interpolation between position sample points we can be pretty rough with compression. 32 distinct values per-meter per-second provides acceptable precision.

Linear velocity has been bounded and quantized and is now three integers in the range [-1024,1023]. That breaks down as follows: [-32,+31] (6 bits) for integer component and multiply 5 bits fraction precision. I hate messing around with sign bits so I just add 1024 to get the value in range [0,2047] and send that instead. To decode on receive just subtract 1024 to get back to signed integer range before converting to float.

11 bits per-component gives 33 bits total per-linear velocity. Just over $^1/_3$ the original uncompressed size!

We can do even better than this because most cubes are stationary. To take advantage of this we just write a single bit "at rest". If this bit is 1, then velocity is implicitly zero and is not sent. Otherwise, the compressed velocity follows after the bit (33 bits). Cubes at rest now cost just 127 bits, while cubes that are moving cost one bit more than they previously did: 159 + 1 = 160 bits.

But why are we sending linear velocity at all? In the previous article (http://web.archive.org/web/20181107181434/https://gafferongames.com/networked-physics/snapshots-and-interpolation/) we decided to send it because it improved the quality of interpolation at 10 snapshots per-second, but now that we're sending 60 snapshots per-second is this still necessary? As you can see below the answer is *no*.

0:00

Linear interpolation is good enough at 60HZ. This means we can avoid sending linear velocity entirely. Sometimes the best bandwidth optimizations aren't about optimizing what you send, they're about what you *don't* send.

Optimizing Position

Now we have only position to compress. We'll use the same trick we used for linear velocity: bound and quantize. I chose a position bound of [-256,255] meters in the horizontal plane (xy) and since in the cube simulation the floor is at z=0, I chose a range of [0,32] meters for z.

Now we need to work out how much precision is required. With experimentation I found that 512 values per-meter (roughly 2mm precision) provides enough precision. This gives position x and y components in [-131072,+131071] and z components in range [0,16383]. That's 18 bits for x, 18 bits for y and 14 bits for z giving a total of 50 bits per-position (originally 96).

This reduces our cube state to 80 bits, or just 10 bytes per-cube.

This is approximately $^1/_4$ of the original cost. Definite progress!

Now that we've compressed position and orientation we've run out of simple optimizations. Any further reduction in precision results in unacceptable artifacts.

Delta Compression

Can we optimize further? The answer is yes, but only if we embrace a completely new technique: **delta compression**.

Delta compression sounds mysterious. Magical. Hard. Actually, it's not hard at all. Here's how it works: the left side sends packets to the right like this: "This is snapshot 110 encoded relative to snapshot 100". The snapshot being encoded relative to is called the baseline. How you do this encoding is up to you, there are many fancy tricks, but the basic, big order of magnitude win comes when you say: "Cube n in snapshot 110 is the same as the baseline. One bit: Not changed!"

To implement delta encoding it is of course essential that the sender only encodes snapshots relative to baselines that the other side has received, otherwise they cannot decode the snapshot. Therefore, to handle packet loss the receiver has to continually send "ack" packets back to the sender saying: "the most recent snapshot I have received is snapshot n". The sender takes this most recent ack and if it is more recent than the previous ack updates the baseline snapshot to this value. The next time a packet is sent out the snapshot is encoded relative to this more recent baseline. This process happens continuously such that the steady state becomes the sender encoding snapshots relative to a baseline that is roughly RTT (round trip time) in the past.

There is one slight wrinkle: for one round trip time past initial connection the sender doesn't have any baseline to encode against because it hasn't received an ack from the receiver yet. I handle this by adding a single flag to the packet that says: "this snapshot is encoded relative to the initial state of the simulation" which is known on both sides. Another option if the receiver doesn't know the initial state is to send down the initial state using a non-delta encoded path, eg. as one large data block, and once that data block has been received delta encoded snapshots are sent first relative to the initial baseline in the data block, then eventually converge to the steady state of baselines at RTT.

As you can see above this is a big win. We can refine this approach and lock in more gains but we’re not going to get another order of magnitude improvement past this point. From now on we’re going to have to work pretty hard to get a number of small, cumulative gains to reach our goal of 256 kilobits per-second.

Incremental Improvements

First small improvement. Each cube that isn’t sent costs 1 bit (not changed). There are 901 cubes so we send 901 bits in each packet even if no cubes have changed. At 60 packets per-second this adds up to 54kbps of bandwidth. Seeing as there are usually significantly less than 901 changed cubes per-snapshot in the common case, we can reduce bandwidth by sending only changed cubes with a cube index [0,900] identifying which cube it is. To do this we need to add a 10 bit index per-cube to identify it.

There is a cross-over point where it is actually more expensive to send indices than not-changed bits. With 10 bit indices, the cost of indexing is 10*n bits. Therefore it’s more efficient to use indices if we are sending 90 cubes or less (900 bits). We can evaluate this per-snapshot and send a single bit in the header indicating which encoding we are using: 0 = indexing, 1 = changed bits. This way we can use the most efficient encoding for the number of changed cubes in the snapshot.

This reduces the steady state bandwidth when all objects are stationary to around 15 kilobits per-second. This bandwidth is composed entirely of our own packet header (uint16 sequence, uint16 base, bool initial) plus IP and UDP headers (28 bytes).

Next small gain. What if we encoded the cube index relative to the previous cube index? Since we are iterating across and sending changed cube indices in-order: cube 0, cube 10, cube 11, 50, 52, 55 and so on we could easily encode the 2nd and remaining cube indices relative to the previous changed index, e.g.: +10, +1, +39, +2, +3. If we are smart about how we encode this index offset we should be able to, on average, represent a cube index with less than 10 bits.

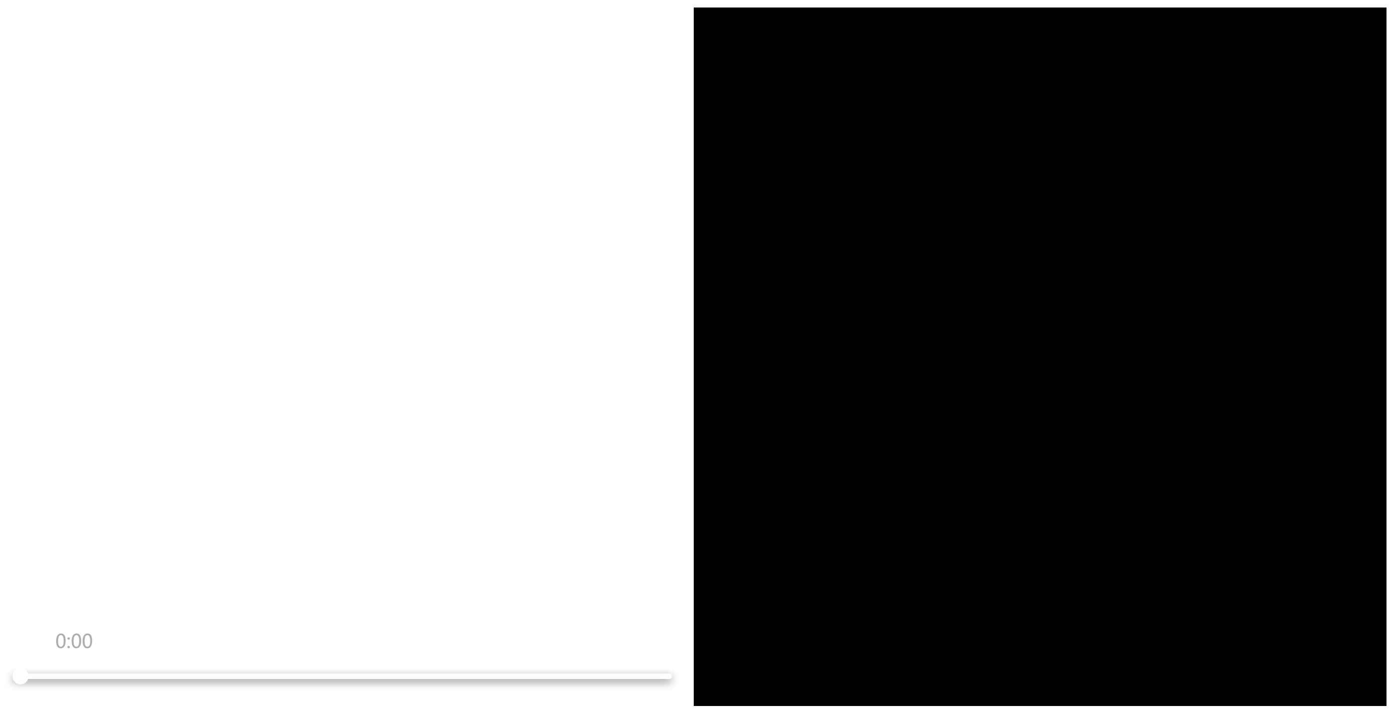
The best encoding depends on the set of objects you interact with. If you spend a lot of time moving horizontally while blowing cubes from the initial cube grid then you hit lots of +1s. If you move vertically from initial state you hit lots of +30s (sqrt(900)). What we need then is a general purpose encoding capable of representing statistically common index offsets with less bits.

- [1,8] => 1 + 3 (4 bits)
- [9,40] => 1 + 1 + 5 (7 bits)
- [41,900] => 1 + 1 + 10 (12 bits)

Notice how large relative offsets are actually more expensive than 10 bits. It's a statistical game. The bet is that we're going to get a much larger number of small offsets so that the win there cancels out the increased cost of large offsets. It works. With this encoding I was able to get an average of 5.5 bits per-relative index.

Now we have a slight problem. We can no longer easily determine whether changed bits or relative indices are the best encoding. The solution I used is to run through a mock encoding of all changed cubes on packet write and count the number of bits required to encode relative indices. If the number of bits required is larger than 901, fallback to changed bits.

Here is where we are so far, which is a significant improvement:



Next small improvement. Encoding position relative to (offset from) the baseline position. Here there are a lot of different options. You can just do the obvious thing, eg. 1 bit relative position, and then say 8-10 bits per-component if all components have deltas within the range provided by those bits, otherwise send the absolute position (50 bits).

This gives a decent encoding but we can do better. If you think about it then there will be situations where one position component is large but the others are small. It would be nice if we could take advantage of this and send these small components using less bits.

It's a statistical game and the best selection of small and large ranges per-component depend on the data set. I couldn't really tell looking at a noisy bandwidth meter if I was making any gains so I captured the position vs. position base data set and wrote it to a text file for analysis.

I wrote a short ruby script to find the best encoding with a greedy search. The best bit-packed encoding I found for the data set works like this: 1 bit small per delta component followed by 5 bits if small [-16,+15] range, otherwise the delta component is in [-256,+255] range and is sent with 9 bits. If any component delta values are

Delta Encoding Smallest Three

Next I figured that relative orientation would be a similar easy big win. Problem is that unlike position where the range of the position offset is quite small relative to the total position space, the change in orientation in 100ms is a much larger percentage of total quaternion space.

I tried a bunch of stuff without good results. I tried encoding the 4D vector of the delta orientation directly and recomposing the largest component post delta using the same trick as smallest 3. I tried calculating the relative quaternion between orientation and base orientation, and since I knew that w would be large for this (rotation relative to identity) I could avoid sending 2 bits to identify the largest component, but in turn would need to send one bit for the sign of w because I don’t want to negate the quaternion. The best compression I could find using this scheme was only 90% of the smallest three. Not very good.

I was about to give up but I run some analysis over the smallest three representation. I found that 90% of orientations in the smallest three format had the same largest component index as their base orientation 100ms ago. This meant that it could be profitable to delta encode the smallest three format directly. What’s more I found that there would be no additional precision loss with this method when reconstructing the orientation from its base. I exported the quaternion values from a typical run as a data set in smallest three format and got to work trying the same multi-level small/large range per-component greedy search that I used for position.

The best encoding found was: 5-8, meaning [-16,+15] small and [-128,+127] large. One final thing: as with position the large range can be extended a bit further by knowing that if the component value is not small the value cannot be in the [-16,+15] range. I leave the calculation of how to do this as an exercise for the reader. Be careful not to collapse two values onto zero.

The end result is an average of 23.3 bits per-relative quaternion. That’s 80.3% of the absolute smallest three.

That’s just about it but there is one small win left. Doing one final analysis pass over the position and orientation data sets I noticed that 5% of positions are unchanged from the base position after being quantized to 0.5mm resolution, and 5% of orientations in smallest three format are also unchanged from base.

These two probabilities are mutually exclusive, because if both are the same then the cube would be unchanged and therefore not sent, meaning a small statistical win exists for 10% of cube state if we send one bit for position changing, and one bit for orientation changing. Yes, 90% of cubes have 2 bits overhead added, but the 10% of cubes that save 20+ bits by sending 2 bits instead of 23.3 bit orientation or 26.1 bits position make up for that providing a small overall win of roughly 2 bits per-cube.

As you can see the end result is pretty good.

Conclusion

And that's about as far as I can take it using traditional hand-rolled bit-packing techniques. You can find source code for my implementation of all compression techniques mentioned in this article here (http://web.archive.org/web/20181107181434/https://gist.github.com/gafferongames/bb7e593ba1b05da35ab6).

It's possible to get even better compression using a different approach. Bit-packing is inefficient because not all bit values have equal probability of 0 vs 1. No matter how hard you tune your bit-packer a context aware arithmetic encoding can beat your result by more accurately modeling the probability of values that occur in your data set. This implementation (http://web.archive.org/web/20181107181434/https://github.com/rygorous/gaffer_net/blob/master/main.cpp) by Fabian Giesen beat my best bit-packed result by 25%.

It's also possible to get a much better result for delta encoded orientations using the previous baseline orientation values to estimate angular velocity and predict future orientations rather than delta encoding the smallest three representation directly.

Chris Doran from Geomerics wrote also wrote an excellent article (http://web.archive.org/web/20181107181434/http://www.geomerics.com/wp-content/uploads/2015/04/rotation_blog_toprint.pdf) exploring the mathematics of relative quaternion compression that is worth reading.

NEXT ARTICLE: State Synchronization (http://web.archive.org/web/20181107181434/https://gafferongames.com/post/state_synchronization/)

● (http://web.archive.org/web/20181107181434/https://twitter.com/gafferongames)

● (http://web.archive.org/web/20181107181434/https://github.com/gafferongames)