Networked Physics (2004)

How to network a physics simulation

Posted by Glenn Fiedler (http://web.archive.org/web/20181107181430/https://gafferongames.com/about) on Saturday, September 4, 2004

Introduction

Hi, I'm Glenn Fiedler (http://web.archive.org/web/20181107181430/https://gafferongames.com/about) and welcome to **Game Physics (http://web.archive.org/web/20181107181430/https://gafferongames.com/categories/game-physics/)**.

In the previous article (http://web.archive.org/web/20181107181430/https://gafferongames.com/post/spring_physics) we discussed how to use spring-like forces to model basic collision response, joints and motors.

In this article we're going to discuss how to network a physics simulation.

First Person Shooters

First person shooter physics are usually very simple. The world is static and players are limited to running around and jumping and shooting.

Because of cheating, first person shooters typically operate on a client-server model where the server is authoritative over physics. This means that the true physics simulation runs on the server and the clients display an approximation of the server physics to the player.

The problem then is how to allow each client to control his own character while displaying a reasonable approximation of the motion of the other players.

In order to do this elegantly and simply, we structure the physics simulation as follows:

2. Physics state is fully encapsulated in a state structure.

To do this we need to gather all the user input that drives the physics simulation into a single structure and the state representing each player character into another.

Here is an example from a simple run and jump shooter:

```
struct Input
{
     bool left;
     bool right;
     bool forward;
     bool back;
     bool jump;
};

struct State
{
     Vector position;
     Vector velocity;
};
```

Next we need to make sure that the simulation gives the same result given the same initial state and inputs over time. Or at least, that the results are as close as possible. I'm not talking about perfect floating point determinism here, just a reasonable $^1/_2$ second prediction giving approximately the same result.

Network Fundamentals

I will briefly discuss actually networking issues in this section before moving on to the important information of what to send over the pipe. It is after all just a pipe after all, networking is nothing special right? Beware! Ignorance of how the pipe works will really bite you. Here are the two networking fundamentals that you absolutely need to know:

Number one. If your network programmer is any good at all he will use UDP, which is an unreliable data protocol, and build some sort of application specific networking layer on top of this. The important thing that you as the physics programmer need to know is that you absolutely must design your physics communication over the network so that you can receive the most recent input and state without waiting for lost packets to be resent. This is important because otherwise your physics simulation will stall out under bad networking conditions.

Two. You will be very limited in what can be sent across the network due to bandwidth limitations. Compression is a fact of life when sending data across the network. As physics programmer you need to be very careful what data is compressed and how it is done. For the sake of determinism, some data must not be compressed, while other data is safe. Any data that is compressed in a lossy fashion should have the same quantization applied locally where

simulation.

Physics Runs On The Server

The fundamental primitive we will use when sending data between the client and the server is an unreliable data block, or if you prefer, an unreliable non-blocking remote procedure call (rpc). Non-blocking means that the client sends the rpc to the server then continues immediately executing other code, it does not wait for the rpc to execute on the server! Unreliable means that if you call the rpc is continuously on the the server from a client, some of these calls will not reach the server, and others will arrive in a different order than they were called. We design our communications around this primitive because it suits the transport layer (UDP).

The communication between the client and the server is then structured as what I call a “stream of input” sent via repeated rpc calls. The key to making this input stream tolerant of packet loss and out of order delivery is the inclusion of a floating point time in seconds value with every input rpc sent. The server keeps track of the current time on the server and ignores any input received with a time value less than the current time. This effectively drops any input that is received out of order. Lost packets are ignored.

Thinking in terms of our standard first person shooter, the input we send from client to server is the input structure that we defined earlier:

```
struct Input
{
     bool left;
     bool right;
     bool forward;
     bool back;
     bool jump;
};

class Character
{
public:

     void processInput( double time,
                        Input input );
};
```

Thats the bare minimum data required for sending a simple ground based movement plus jumping across the network. If you are going to allow your clients to shoot you’ll need to add mouse input as part of the input structure as well because weapon firing needs to be done server side.

Notice how I define the rpc as a method inside an object? I assume your network programmer has a channel structure built on top of UDP, eg. some way to indicate that a certain rpc call is directed as a specific object instance on the remote machine.

input rpcs are received from the client that owns it. This means that the physics state of different client characters are slightly out of phase on the server, some clients being a little bit ahead and others a little bit behind in time. Overall however, the different client characters advance ahead roughly in sync with each other.

Lets see how this rpc call is implemented in code on the server:

```
void processInput( double time, Input input )
{
    if ( time < currentTime )
        return;

    float deltaTime = currentTime - time;

    updatePhysics( currentTime, deltaTime, input );
}
```

The key to the code above is that by advancing the server physics simulation for the client character is performed only as we receive input from that client. This makes sure that the simulation is tolerant of random delays and jitter when sending the input rpc across the network.

Clients Approximate Physics Locally

Now for the communication from the server back to the clients. This is where the bulk of the server bandwidth kicks in because the information needs to be broadcast to all the clients.

What happens now is that after every physics update on the server that occurs in response to an input rpc from a client, the server broadcasts out the physics state at the end of that physics update and the current input just received from the rpc.

This is sent to all clients in the form of an unreliable rpc:

```
void clientUpdate( float time, Input input, State state )
{
    Vector difference = state.position -
                        current.position;

    float distance = difference.length();

    if ( distance > 2.0f )
        current.position = state.position;
    else if ( distance > 0.1 )
        current.position += difference * 0.1f;

    current.velocity = velocity;

    current.input = input;
}
```

current position on the client is more than 10cms, move 10% of the distance between the current position and the correct position. Otherwise do nothing.

Since server update rpcs are being broadcast continually from the server to the the clients, moving only a fraction towards the snap position has the effect of smoothing the correction out with what is called an exponentially smoothed moving average.

This trades a bit of extra latency for smoothness because only moving some percent towards the snapped position means that the position will be a bit behind where it should really be. You don't get anything for free. I recommend that you perform this smoothing for immediate quantities such as position and orientation, while directly snapping derivative quantities such as velocity, angular velocity because the effect of abruptly changing derivative quantities is not as noticeable.

Of course, these are just rules of thumb. Make sure you experiment to find out what works best for your simulation.

Client-Side Prediction

So far we have a developed a solution for driving the physics on the server from client input, then broadcasting the physics to each of the clients so they can maintain a local approximation of the physics on the server. This works perfectly however it has one major disadvantage. Latency!

When the user holds down the forward input it is only when that input makes a round trip to the server and back to the client that the client's character starts moving forward locally. Those who remember the original Quake netcode would be familiar with this effect. The solution to this problem was discovered and first applied in the followup QuakeWorld and is called client side prediction. This technique completely eliminates movement lag for the client and has since become a standard technique used in first person shooter netcode.

Client side prediction works by predicting physics ahead locally using the player's input, simulating ahead without waiting for the server round trip. The server periodically sends corrections to the client which are required to ensure that the client stays in sync with the server physics. At all times the server is authoritative over the physics of the character so even if the client attempts to cheat all they are doing is fooling themselves locally while the server physics remains unaffected. Seeing as all game logic runs on the server according to server physics state, client side movement cheating is basically eliminated.

The most complicated part of client side prediction is handling the correction from the server. This is difficult, because the corrections from the server arrive *in the past* due to client/server communication latency. We need to apply this correction in the past, then calculate the

The standard technique to do this is to store a circular buffer of saved moves on the client where each move in the buffer corresponds to an input rpc call sent from the client to the server:

```
struct Move
 {
     double time;
     Input input;
     State state;
 };
```

When the client receives a correction it looks through the saved move buffer to compare its physics state at that time with the corrected physics state sent from the server. If the two physics states differ above some threshold then the client rewinds to the corrected physics state and time and replays the stored moves starting from the corrected state in the past, the result of this re-simulation being the corrected physics state at the current time on the client.

Sometimes packet loss or out of order delivery occurs and the server input differs from that stored on the client. In this case the server snaps the client to the correct position automatically via rewind and replay. This snapping is quite noticeable to the player, so we reduce it with the same smoothing technique we used above for the other player characters. This smoothing is done *after* recalculating the corrected position via rewind and replay.

Conclusion

We can easily apply the client side prediction techniques used in first person shooters to network a physics simulation, but only if there is a clear ownership of objects by clients and these object interact mostly with a static world.