Gaffer on Games

Glenn Fiedler, an Australian Game Developer in Los Angeles

Networked Physics

Introduction

Hi, I'm Glenn Fiedler and welcome to the final article in my series on Game Physics.

In the previous article we discussed how to use spring-like forces to model basic collision response, joints and motors.

Now we're going to discuss how to network a physics simulation.

Networking a physics simulation is the holy grail of multiplayer gaming and the massive popularity of first person shooters on the PC is a testament to the just how immersive a networked physics simulation can be.

In this article I will show you how apply the key networking techniques from first person shooters to network your own physics simulation.

First person shooters

First person shooter physics is 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 will physics simulation in the following way:

1. Character physics are completely driven from input data
2. Physics state is known and can be fully encapsulated in a state structure
3. The physics simulation is reasonably deterministic given the same initial state and inputs

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 determinism to the level of floating point accuracy and rounding modes, just a reasonable, 1-2 second prediction giving basically 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 possible, so that the result is the same on both machines. Bottom line you'll need to be involved in this compression in order to make it as efficient as possible without breaking your simulation.

For more details, see my new article series called Networking for Game Programmers.

Physics is run on the server according to a stream of input from clients

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 receiveInput(float time, Input input);          // rpc method called on server
    };
```

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.

So how does the server process these rpc calls? It basically sits in a loop waiting for input from each of the clients. Each character object has its physics advanced ahead in time individually as 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 receiveInput(float time, Input input)
    {
        if ( time < currentTime )
            return;

        const 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 server 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 positionDifference = state.position - currentState.position;
        float distanceApart = positionDifference.length();
        if ( distanceApart > 2.0 )
            currentState.position = state.position;
        else if ( distanceApart > 0.1 )
            currentState.position += positionDifference * 0.1f;
        currentState.velocity = velocity;
        currentInput = input;
    }
```

What is being done here is this: if the two positions are significantly different (>2m apart) just snap to the corrected position, otherwise if the distance between the server position and the 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.

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 resulting corrected position at present time on the client.

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
{
     float 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 being the corrected physics state at the current time on the client:

```
const int maximum = 1024;

Move moves[maximum];

void advance(int &index)
{
     index ++;
     if (index>=maximum)
          index -= maximum;
}

int head = 0;
int tail = 100;          // lets assume 100 moves are currently stored

void clientCorrection(float time, State state, Input input)
{
     while (time>moves[index].time && head!=tail)
          advance(head);          // discard old moves
```

```
        if (head!=tail && time==moves[head].time)
        {
            if ((moves[head].state.position-currentState.position).length>threshold)
            {
                // rewind and apply correction

                currentTime = time;
                currentState = state;
                currentInput = input;

                advance(head);          // discard corrected move

                int index = head;

                while (index!=tail)
                {
                    const float deltaTime = moves[index].time - currentTime;

                    updatePhysics(currentTime, deltaTime, currentInput);

                    currentTime = moves[index].time;
                    currentInput = moves[index].input;

                    moves[index].state = currentState;

                    advance(index);
                }
            }
        }
    }
```

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* we recalculate the corrected position via rewind and replay.

Disadvantages of client side prediction

Client side prediction seems to good to be true. With one simple trick we can eliminate latency when the client character moves. But at what cost? The answer is that whenever two physics simulations on the server interact then snapping will occur on the clients. What happened? In effect, the client predicted the physics ahead using its own approximation of the world but ended up at an entirely different position from the physics on the server. Snap.

In our simple run and jump fps, this situation would occur if one player runs into another, tries to stand on another player's head or gets knocked back by an explosion. The bottom line is that any change to the physics of a character that occurs on the server that is not directly related to a change in input sent from its owner client will cause snapping. In truth there is no way to avoid this short of throwing client side prediction completely out the window and accepting the round trip latency.

This leads me to an interesting point. The evolution of networked physics games is from first person shooters where characters run around in a static world while shooting each other, towards a dynamic world where players interact with the surroundings and each other. Given this trend I am willing to make the following bold prediction: traditional client side prediction as presented in this article, may soon be obsolete.

General networked physics

The solutions presented so far work great for first person shooter games. The key limitation is that there is a clear ownership of objects by a certain client, meaning that in the vast majority of cases the owner client is the only influence on the physics of a physics object being networked.

It is this simplifying assumption that enables all of the common techniques and tricks used when writing netcode for first person shooters. If your physics simulation is structured in this way, then these techniques are perfect for you. For example, a car racer has each client controlling a single car could extend this technique by simply adding more physics state and input.

But what if the simulation you want to network has no clear concept of object ownership? For example, consider a physics simulation of a stack of blocks where each client is free to click and drag to move the blocks around. In this case no client owns an object, and there could even be multiple clients pulling on the same block simultaneously. Perhaps one player is standing on top of the blocks, while another player decides to drive through this stack of blocks in his car! Very complicated!

In a case such as this, more general techniques need to be used. For starters, client side prediction is obviously out because the snapping that it would cause would be unacceptable. So the key then is to run the physics simulation in such a way that the stream of input is sent from the clients to the server but the server does not wait for a clients input before proceeding with the simulation, because the server has no concept of a client owning the object like a player client owns their character in an first person shooter.

This means that the server physics update will actually look more traditional because all objects would be updated simultaneously on the server using the last known input from the clients. This makes the server simulation much more susceptible to network problems such as delayed delivery of packets, bunching up of input data arrival on the server, plus work needs to be done to keep the client's concept of time in sync with that of the server.

Solving these problems in general networked physics will be a challenge, and I hope to present solutions and sample source code for multiplayer interaction with a stack of objects in the future!

Conclusion

Networked physics is complicated, but can be made a bit easier to understand once you understand the core techniques used in first person shooters.

To accompany this article I have created a networked physics simulation where the FPS character is replaced by a cube. You can run and jump with the cube, and the cube will roll and tumble amount in response to your input. No shooting I'm afraid, sorry!

There are many visualizations included in the example program to help you understand the concepts of rewind and replay and smoothing, so download the example today and play around with it!

Thats it for my articles on Game Physics.

You can read more about game networking in my new article series called Networking for Game Programmers!

See you later,

Glenn Fiedler
gaffer@gaffer.org

70 Responses leave one →

1. 2006 October 13
   pTymN permalink

   There is another problem that you didn't mention, that makes the simple "state-diff" method not work. Different numerical coprocessors generate slightly different values given identical inputs. I have confirmed this by adding a replay system to my game. When I run a replay file on another computer, errors start building up. You can read about this problem on Gamasutra.

   1. character physics are completely driven from input data
   2. physics state is known and can be fully encapsulated in a state structure
   3. the physics simulation is deterministic given the same initial state and input

   Therefore, #3 is not completely true. It is deterministic on any one machine, but not across machines. A partial stepped state update or full state update is required in order to overcome this problem.

2. 2006 October 13
   pTymN permalink

P.S. You are correct about client side prediction becoming useless in the future. I performed some tests to confirm what I've heard about human reaction time. You can safely add a local latency of 70ms to user input without the user even noticing. That 70ms is enough time to immediately send out the input over the wire to another client to be used at during the frame with the same timestamp on both machines. As broadband continues to get better, we may find that we get to remove a whole slew of dead reckoning tricks, and focus more on utilizing bandwidth.

3. 2006 October 18
netDude permalink

Actually, that's not quite true. There are many floating point math functions that act deterministically across machines. Ones that generally don't act deterministically include the transcendental functions. Use doubles but not sin, cos, sqrt, etc. and you should be fine in most cases.

4. 2006 October 20
Glenn Fiedler permalink

no thats incorrect, you cannot get perfectly deterministic results from general floating point across different machines. rounding errors, differences of loading floating point values into the stack vs. writing them out to memory can change the results of floating point calculations. then it gets even more interesting, lets take an intel vs. amd, or intel vs. powerpc. totally different floating point implementation internally. you get slightly different results!

remember, for true "determinism" you need the *exact* same results from your floating point, not just nearly the same!

so dont try deterministic floating point, its impossible!

5. 2006 October 20
Glenn Fiedler permalink

comment #1, i should clarify that for the techniques in this article, you dont require exact determinism, just that you need to get it as deterministic as possible. i should clarify this in the article — its confusing!

6. 2006 October 27
Pinky permalink

Hey Glenn,

As a hobbyist games programmer, I have really learnt a lot from your articles. I am interested in the actual structure you send across the network in your UDP packets. I had proper Structures, then a StructToByte() function that used the .NET serialization - but found there was far to much overhead in the resulting Byte array because it keeps information about the Packet Type, etc. It would make an interesting topic for an article, if you are keen! Keep up the great work, love your writing style.

Cheers, Pinky.

7. 2006 October 29
Glenn Fiedler permalink

i dont limit my systems to structures, instead i provide a bitstream class and a serialize method, which both reads and writes the data — bit packing reduces bools to 1 bit, and integers can be written with n bits only

once you have this bit packer to get your aggregate string of bytes, send that as your packet

8. 2006 October 29
Pinky permalink

Ok, thanks for the info. I guess the answer I needed was that I should write my own (de)serialize methods which accepts PacketType as a parameter, and creates a Byte[] as appropriate. The first data in the packet could be an Enum for PacketType, or something like that. I will give it a shot. Cheers. -

Pinky

9. 2006 November 19
gah! permalink

your use of un-escaped less than and greater than signs [in psuedocode] messes up this page badly in opera 9

[came here from a link on a mailing list, that's why I'm so late]

10. 2006 November 30
Alex permalink

Hi, you have a syntax error in your structure definition for struct Input. (If it was meant to be pseudo-code, then ignore this.)

Since you are separating variables with a comma delimited list, you do not specify 'bool' each time. Else if you want to, substitute the commas (,) for semicolons (;).

11. 2007 January 19
Barney permalink

Hi,
This seems like a very simple answer to the networking issue, however how does it stack up with larger simulation.
The problem then is the need to keep a historic list for ALL objects in the simulation! With the kind of latency that can often be found, 1 second is a minumum. with each record taking just a few bytes, and a few hundred objects in the simulation, the memory requirements grow quickly!
Are there any tricks you've found to reduce this limitation?

12. 2007 January 19
Glenn Fiedler permalink

stacks up just fine, the more interesting issue is how to handle time synchronization between server and multiple clients in order for this technique to work, taking into account latency and jitter in packet delivery, because larger simulations need to step the whole world at once, and cannot update each client independently

and its not really the memory cost, its the simulation cost. rewinding and replaying a complex simulation (eg. havok) is expensive - and thats the main thing to overcome with this technique

13. 2007 January 21
Barney permalink

i would guess a good method to handle this would be to run two thread, one for the physics simulator, and one for the network listener.

If a packet is recieved and requires a rewind to be performed, start the physics thread, rewound.

If another packet is recieved with a timestamp previous to the current simulation time (possibly rewound) then kill the current simulation, rewind again and continue.

If this new packet is timestamped later than the current sim time, just sort it into the input que at the end of the current step, and continue as normal.

clients only need to be updated upon collision events. this is the only time the simulation becomes seriously discontiuous. client prediction should then keep us back in (roughly) the right place with the system already implemented handling any irregularities.

14. 2007 January 21
Barney permalink

I see your point on memory requirements, memory is cheap these days!

although rewinding may be a costly exercise, rewinding could be sped up by also storing state information in the history que. the only time this state information would become invalid is after a collision event. once this is flagged for the object we can recalculate the new states for each frame, othewise it can be ignored.

Collision detection would be the major bottleneck. In this case i would imagine using lower detail models for the physics, and a higher detail model for the renderer would be the simplest system (along with the usual hierarchal bounding volumes)

15. 2007 February 16
Suchon permalink

Now i'm follow this article in my game that it is a racing game and now i try to use the technic that call 'an exponentially smoothed moving average' but in my game it is a 3d games that have not only position velocity but also rotation matrix I really want to know how to apply this technic with rotation matrix?? anybody can help ??

16. 2007 February 16
Glenn Fiedler permalink

use quaternions, not matrices to represent your rotation - then its easy to implement exponentially smoothed moving average for orientation

take a look at the zen of networked physics source code, there is in example in there of smoothing position and orientation of a cube

cheers

17. 2007 February 16
Suchon permalink

In my game I use ODE as a physic engine and it use a rotation matrix, so by your suggestion I have to convert my rotation matrix to quatemions right? I already try to look in your code that load from zen of network,the box code with presentation, But I don't know where to look .If you can point me to where to look, it would be a lot easier for me. Thank you.

18. 2007 February 17
Glenn Fiedler permalink

See Matrix.h and Quaternion.h for conversion routines, and Cube.h for the exponentially smoothed average.

Its not hard at all, look!

```
current.position = previous.position + (target.position-previous.position) * tightness;
current.orientation = slerp(previous.orientation, target.orientation, tightness);
```

Tightness is in the range [0,1]. If you set tightness to 1.0, then there is no smoothing, reducing tightness makes it follow more sloppy. Try values like 0.5, 0.25 and adjust to get the amount of smoothing you want — this technique for smoothing is framerate dependent, so you'll want to have a fixed delta time

cheers

19. 2007 February 18
Suchon permalink

First of all thank you for your suggestion it's help me a lot but I try to adapt it, and now I can work with smooth method but another problem for me is that I'm confuse with the time that we need to update, first I run physic on each client and server send position to update frequntly. when I receive the position from server I try to update it by use smooth method try to move car to its new position that come from server by call smooth method in the updatemethod that run by game loop, so if i'm right it should smooth it 1 time in 1 frame. And in that loop client also simulate physic so it's position should be

change. What if message come from server comes late or position is alreay absolete or client already pass that point. ???
Oh and one more problem is that if 2 computer has different performance one can run at 60 frames per second and another can run at 30 frames per sec, so in my simulation(racing game) it's performance of car doesn't the same. So can this tricks work for this problem? or i have to implement anything else?

Thank you…

20. 2007 February 18
Glenn Fiedler permalink

ok you are right - the article doesnt cover this problem

in the networking biz we solve this with "input and state buffering" and "time synchronization"

input and state buffering means that when a packet arrives early or is jittered a bit, we store it in a buffer. this makes sure that we get a reliable stream of input and state from the remote computer, even in real world conditions, where packets sent at 30fps clearly dont arrive at nice regular 30th of a second intervals (they are jittered, clumped together, out of order, duplicated etc. - welcome to the internets!)

time synchronization is the way that we make sure that the client and server advance their concept of time together. in a nutshell, you send the current server time back to the client, and the client adjusts his own time to match the server's time. if the client time is close to the server, it'll smooth towards it (using you guessed it, an exponentially smoothed average), if its outside some threshold (say > 1second out), it'll just snap.

the client does this adjustment by speeding up and slowing down its concept of time progression. eg. you scale deltaTime on the client so it simulates a little bit faster, or a little bit slower, so it approaches server time and holds.

bonus points if you realize that the client can speed up its local time ahead of the server by latency seconds, so that it delivers input packets ahead of the time they are required on the server.

super secret bonus points if you measure the amount of jitter (uncertainty in packet delivery), and further make the client speed up such that it delivers packets so they arrive in time given the latency and jitter conditions.

RADICAL bonus points if you realize that you can do fixed timestep updating on the client and server, but the client can adjust by subtly adjusting its concept of local time in a bresenham line drawing type concept - eg. to speed up it simulations 1,1,1,1,1,2,1,1,1,1,1,2 frames etc, to slow down it goes 1,1,1,1,1,0,1,1,1,1,1,0 etc.

combine this with the "fix your timestep technique" and you can have both fixed timestep and perfectly smooth motion on the client, as well as a network time synchronization that delivers input packets on time to the server, even under latency and jittery packet delivery.

cheers

21. 2007 February 22
Nicolas permalink

Hi

I read your article several times but there is one thing I'm not sure to understand.

Each client has its own physics simulation on the server. When the server receives an input packet (p1) from a client, its simulation steps.
What happened when the server receive an other input packet (p2) from an other client, knowing that the date of p2 is earlier than p1 ? Does the server perform a rollback in the first client simulation to take into account p2 ?

Thanks

22. 2007 February 22
Glenn Fiedler permalink

no, generally the server has one simulation, and the client has one simulation - we're just relying on the fact that in typical FPS style physics, the player character is mostly moving around in a static world, so you can effectively rewind/replay the character physics in the simulation as input arrives, while leaving the rest alone — furthermore, you can keep different "time streams" for each client, so that they advance forward as input arrives for them, instead of necessarily together (because they dont interact)

however, this breaks down when you have a modern physics simulation like havok, where the whole world needs to advance together, you cant just update object A independently of object B.

my solution in this case is to adjust client time (by speeding them up), such that each client is ahead of server time exactly by the amount of latency and uncertainty (jitter), such that for any given frame n, the input for each client has arrived at the server.

this is tricky, but possible, i have it working - its my fulltime job to work on this stuff!

sure, if input arrives late you could consider replaying it, but in general this new technique works better with havok type simulations, — the cost of replaying a whole havok simulation even a few frames is prohibitive (as most games will have already maxxed out the CPU cost of havok to do just a single frame… how can you then go and replay 10-20 frames in one frame?)

so. the new technique is this - speed up the clients so they deliver input and state to the server just before its needed, step the whole simulation forward on the server as normal. no special rewinding.

on the client, give limited authority to the objects that the client controls, eg. his player, objects the player pushes, the vehicle the player drives. send updates for these to the server. the server accepts these positions and trusts the client. yes, this makes cheating possible, but frankly, we have no choice - we simply cannot afford the cost of rewinding on the client and replaying the whole havok simulation.

cheers

23. 2007 February 24
Nicolas permalink

Thanks for your answer. It makes sense now.

Indeed I am using a modern physics simulation (for a racing game), and I was wondering how you could afford the cost of rewinding. I have my answer for that question too.

I think that I'll have to choose between two solutions (considering using an havok type physics engine) :
- Simulation by each client. Everybody trust everybody. The server just broadcast the state of the clients. Clients interpolate the other client's states.
- Simulation on the server. Prediction on the client. Speed up clients so that they deliver input in time to the server. Interpolate between prediction state and server state.

Any advise? 😃

24. 2007 February 27
Glenn Fiedler permalink

advice: try both, pick the one that works best - have fun! 😃

25. 2007 March 13
Suchon permalink

Hello Again, And long time no see.

Glenn, Thank you for all advise you gave me. And now My Game network is a lot better and be able to sync. And for Nicolas I use the method that the same as your first method that is

"- Simulation by each client. Everybody trust everybody. The server just broadcast the state of the clients. Clients interpolate the other client's states." and this was work well.

Thank you. Glenn for this valuable article and your advice.

26. 2007 March 17
Tim permalink

First, thanks for such great information, your advice has helped me tremendously. I do have one question however, I'm curious about how the bresenham line drawing concept relates to speeding up time or slowing it down. I know it's a line rasterization algorithm but how exactly does it help with time? Also, you mention "frames" in regard to that. Is this rendered frames or game "ticks"? Thanks!

27. 2007 March 17
Glenn Fiedler permalink

what i mean by bresenham line is that time is discrete (fixed dt) so its measured in simulation frames, n, n+1, n+2 etc.

so how do you speed up and slow down time when its fixed rate time?

i keep a floating point time values on the client and i increase it by the delta i calculate (somewhere between 0.75 and 1.25, depending on whether the client wants to speed up or slow down)

then i look at the current and previous floating point time, and see how many integer frames it spans

you can see with delta being in [0.75,1.0] that it must span either 0, 1 or 2 frames.

so while running without any time adjustment looks like this (simulated frames per fixed real dt, eg: 30fps)

1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1

slowing down looks like this:

1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,0 …

and speeding up looks like:

1,1,1,1,1,2,1,1,1,1,1,2,1,1,1,1,1,1,2 …

(not that i dont actually simulate 2 frames, i just sim one frame with double the delta t, so it has the same cost)

so this is where the bresenham line comes in, i just meant that its an aliased looking line as you adjust a floating point number and then alias it onto integer frames

ps. if you dont want to see temporal aliasing (jitter) when it skips or doubles a frame when adjusting time, use the techniques in the "fix your timestep" article

cheers

28. 2007 March 17
Tim permalink

Thank you so much for the clarification, manipulating time makes much more sense to me after your info. I must be misunderstanding one element though—you mentioned that by increasing a floating point time by a delta between .75 and 1.25 you would get a range that would span 0,1, or 2 frames respectively.

I'm currently using your "fix your timestep" technique in my simulation(which works great). Let's say I have a fixed timestep of 100hz, and I'm using a millisecond timer. Say I start at time 0, I keep rendering and accumulating time, then check my current time and I'm at 11 millseconds, just over a 100th and time for a simulation frame. If I multiply 11 milliseconds by 1.25 (I'm assuming you multiply

100th and time for a simulation frame. If I multiply 11 milliseconds by 1.25 (I'm assuming you multiply when you say you increase?), I get 13.75 milliseconds, which is not 20, the amount needed for 2 simulation frames. Should I be multiplying after every render and not just after a 100th has passed?

I think I see how multiplying by .75 would equal 0 frames (because it would reduce the time and mean you're not quite at a 100th), but the 1.25 equals 2 frames I'm not sure about. Am I totally misunderstanding what you meant? Thanks!

29. 2007 March 18
Glenn Fiedler permalink

ok, i dont understand your example but i'll explain my time adjustment again in detail:

what you normally do, is you increase simulation time by 1.0, so its all very easy:

time is 10.0, 11.0, 12.0, 13.0, 14.0, 15.0

i call this time "normalized time", because one time unit is a frame. you'll find this very useful with this technique, instead of using seconds.

ok, so when the client adjusts to meet a target, it is either going to speeding up or slowing down, during this period, it changes the delta time between 0.75 and 1.25 approximately. note that the client is still framelocking and actually simulating at exactly 30fps, its just that we apply a time scale on the client, in real time, so the client seems to be running faster/slower than usual. the goal being, catchup to the time value at which the server wants the client to be at.

to the user, the time scale is at worst case 3/4 time when slowing down, or 5/4 time when speeding up

note that my time adjustment technique doesnt actually multiply dt, it actually adds an offset to it clamped in the range of [-0.25f, +0.25f] towards the direction it needs to adjust, it does this by looking at the 'target time' sent from the server, and using an exponentially smoothed average and some filters to smoothly adjust towards this. calculating the target time is complex and requires lots of filters, i cant go into detail about how i do this yet, sorry.

now, lets think in terms of fixed time again, 'floating point' time is really just frame number + some fractional number

in terms of our fixed rate simulation, frame 10.26432 doesnt really have much meaning, but frame 10 does, and frame 11 does. fractional frames are meaningless to us - at frame n, we take state n, and input n, and apply the simulation to get state n + 1

so how do we reconcile the two? let me show you how it works:

lets say we're speeding up so client delta time is 1.25:

current frame time is 10.9 (almost 11), so we add our 1.25 to get frame 12.15

how many integer frames have we crossed? we have crossed over frame 11, and frame 12. so 2 frames.

so this simulation step, we do 2 frames instead of the normal one.

next time through the simulation, delta time is still 1.25, and the frame time is 12.15, so we add 1.25 to this to get 13.4

how many frames have we crossed? just 13. this time we do one simulation step.

and so on.

now consider the maximum slowdown case, client delta time will be 0.75, lets start at time 10.2:

10.2 + 0.75 = 10.95

how many frames have we crossed? zero. so this time around we do no simulation. this frame is skipped.

next frame we have time = 10.95, and we add 0.75 to it, getting 11.7

now we have cross one frame, so we simulate that.

and so on, so you see when we are slowed down, the actual *effect* is to simulate 1,1,1,0,1,1,1,0,1,1,1,0,1,1,1,0

and when we are speeding up 1,1,1,2,1,1,1,2,1,1,1,2 etc.

get it?

cheers

30. 2007 March 18
Glenn Fiedler permalink

i should also make it clear, that the technique i'm talking about now, is about how to do networking and time adjustment with a *fixed time step*, this is desirable for networking and for consoles

when i say "use the fix your timestep" technique, i dont mean do the time accumulation and decouple render and simulation rate as per that article

i mean, used fixed time step, and use the interpolation technique in that to handle the fact that on the client, you now have fractional frames, and interpolate between them

hope that makes it clearer

cheers

31. 2007 March 18
Glenn Fiedler permalink

however, i should point out that it seems possible to combine the two techniques

cheers

32. 2007 March 19
Tim permalink

Ok, I think I get it now, thanks for taking the time to explain it, I really appreciate it. (I also understand you can't reveal everything you are doing, I'm just grateful for the snippets you can release  When you say the server sends the correct time to the client, you meant it sends the correct integer simulation frame to the clients, and that's what they catch up too, right? (looking at the source code for this article also seems to point to that conclusion) I had been assuming that the server sent an actual time value in seconds to the clients, that's probably where I screwed up. Your sequence of 0,1, or 2 frames makes perfect sense under that conclusion.

When you say the client is framelocked and simulating 30 frames to second, do you mean that the client just passes in 1/30 as dt to the physics function every rendered frame? Wouldn't this cause the physics speed to fluctuate based on rendering rate, even with a fixed timestep?

What I've been doing currently is using the time accumulation in the "fix your timestep" technique to regulate all sorts of time intervals—i.e. I only send updates to the server 30 times per second, as well as run all my game logic at that rate. I need this kind of exact game loop timing so I can save commands and play them back within my simulation engine, but it has caused me a nightmare in trying to network as each client accumulates time at a different rate, even though each is accumulating at 30 frames per second. The server just relays input as received, but sometimes a client will not get an update from another client in time to render it, thus causing stuttering. This is all just on a lan testing environment with no real latency.

Perhaps I do need to work on combining the two methods as you mentioned, I wonder if there's an easier way to get a fixed rate game loop and still synchronize timing?

33. 2007 March 20
Glenn Fiedler permalink

yes framelocking 30fps means the client passes in 1/30 dt each frame, but also that the client *waits* for 30fps to go past before simulating that frame, i do this by spinlock at the end of the frame (if 1/30th of a second has not been taken up, wait until it is), if you exceed 1/30th of a second, too bad you slow down

the main reason i do this is that my network model advances all players and objects ahead together (unlike the article above, which advances them separately) - if you dont need this, then you dont have to framelock, so in that case do your client time adjustment some other way

btw. the stuttering is called jitter, what you need to do is speed the client up so that he delivers his input ahead of time, both so latency is cancelled out, and jitter is accomodated for. if the client delivers packets +/- 2 frames spread, speed the client so he is 2 frames more ahead. solved.

cheers

34. 2007 March 23
Tim permalink

Just wanted to say thanks again for your patience and advice. I am going to try to implement your suggestions.

35. 2007 April 24
TB permalink

I have a newbie question regarding how to build the networkedPhysics demo app. I downloaded and installed visual studio 2005 express. When I try to re-build the app I get an error saying windows.h not found. This .h is referenced by the project's own windows.h file. Do you have any suggestions on what may be wrong?

TIA,
TB

36. 2007 August 27
Curt permalink

Hi Glenn,

Hopefully you still get notifications about comments on this old post. I was wondering if you knew of any other resources regarding the very last section of your article on General networked physics. We want to do a multi-player physics simulation game. Where characters run around and all the players crash into piles of blocks, and or move the blocks. Are there any resources out there? Are you planning on writing the followup article any time soon?

Thanks for any help,
~Curt

37. 2007 October 2
SimZ permalink

Hi Glenn,

I'm a little late but hopefully you'll get to solve my dilemmas =).

I'm implementing your techinques for networked physics and fixed timestep (so useful!!). I've got stuck though on how to deal with collisions. Collisions with the static world are handled quite smoothly but collisions between characters are giving me a headache. The server should be authorative on all clients but having the positions of the character out of sync cannot perform a global collision test accurately.

I was thinking of waiting on the server for the input data to arrive from all clients before advancing to the next timestep, buffering any input arrived ahead of time. This way the server would have a synchronized world state an could make world collision calulations.

Do you think it could work? Or do you have suggestions to solve the issue in another way?

simone

38. 2007 October 2
Glenn Fiedler permalink

yep that works, using that technique on mercs2 right now

only trick is you need to use time synchronization (speed up client / slow down) such that it delivers input and state for the frame *ahead* of server needing it

that way, you advance through a buffer of input and state, and all input from the client has already arrived

cheers

39. 2007 October 31
Cyburg permalink

Hi Glenn

I have been reading both this article, and your other article, “fix your timestep!”, both articles have been a very enlightening read for me, being a game developing novice; thank you for your efforts in putting together these great articles.

In post #31 on this page, you suggest that it may be possible to combine the two techniques, it also seems that Tim (#32) is trying to achieve the same goal; I just wondered if either of you might have any advice on how this might be achieved, or have either of you managed to achieve a working combination of the two techniques?

I would be most grateful for any assistance you can give me, many thanks and best regards.

Cyburg

40. 2007 October 31
Glenn Fiedler permalink

its called “adaptive framerate” - i’ll be talking about it at my GDC 2008 talk next year, hopefully i’ll have an article out then too, cheers

41. 2007 November 8
Todd permalink

I’m a bit late to the game here…but I have a questions related to comment 22 where you suggest the client simulation could run ahead of the server.

In this scenerio, how does the server update the client for objects the client does NOT control. Since the server will always be behind the client, any state updates it gives to the client would be out of date. It seems to cause the reverse problem of when the client has the input before the server.

This article and the subsequent discussion have beeng great. I’m looking foward to your talk at GDC this year.
Thanks for sharing!
Todd

42. 2007 November 9
Glenn Fiedler permalink

you are right, you can solve it one of two ways:

- extrapolate the predicted object ahead to current time on the client
- ignore the time difference, and logically create two “time streams”, client time and lagged server time

i go with #2 for mercs2, because we cant afford to extrapolate using the physics simulation and i dont want to do a dumb predictor (eg. dead reckoning) — as long as the objects from different time streams dont interact, it works awesome

43. 2007 November 30
nigel permalink

Great series of articles

In the last post (42) what do you mean by two time streams? Do you mean that the server controlled objects will actually be n steps behind the client objects (i.e. when the server updates come in you just set that position in the client physics (or at least interpolate between the 2)). If n is small that it hopefully won’t matter too much?

As a matter of interest why not do the dead reckoning?

Presumably there must be also be some correction for the players object as well in case something hit them on the server? Possibly you could keep the last n frames on the client, then when the server comes back compare the history with the server to work out the differences - and then apply them to the current frame on the client?

44. 2007 December 4
Tim permalink

#39 (Cyburg), I’m sorry but at of yet our team hasn’t the time needed to implement the “adaptive framerate”, but I look forward to the GDC info as well.

45. 2008 February 26
Vianney Lecroart permalink

I created an online game with a real physics engine (ODE) on server side and it works quite nice.

http://www.mtp-target.org

46. 2008 March 21
John permalink

#43, I have the same questions as nigel. Glenn, in #42, you mentioned that as long as the objects in two time streams don’t interact, it works fine. However, what if they do? Eg. Client A gets a state update from the server k frames behind its current frame. That state is different from Client A’s state k frame behind. The difference is the position of an object that Client A will collide into. What can be done in this situation? Client A probably can’t afford to change the data and run k frames of simulations to fix the state different.

Thanks for the article and the discussions. They are very helpful.

47. 2008 March 22
Glenn Fiedler permalink

read the slides from my GDC talk, when objects interact then the server authority wins, and the objects are pulled back into the latent time stream - this manifests as a pop on the client

48. 2008 March 22
Glenn Fiedler permalink

why not do dead reckoning? because its breaks down for a stack of objects - fundamentally in this case, the only predictor that actually works is the simulation itself

49. 2008 March 31
Kal permalink

Hey Glenn,

I'm trying to implement network physics from your articles, but I'm confused by a few things.

In your article "Fix Your Timestep" you point out how important a fixed timestep is for deterministic physics,
but in your Networked Physics article you appear to be using a variable timestep for physics on the server.
Having fixed delta on the client, and variable delta on the server makes determinism impossible.
1.
```
void receiveInput(float time, Input input)
{
if ( time < currentTime )
return;

const float deltaTime = currentTime - time; //This varies greatly depending on how many inputs have been missed or delayed

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

Another thing that has me stumped is your NetworkedPhysics code. The article says to send a timestamp to the server, but your code just implements an Acknowledge number. Is the purpose of the timestamp just for rejecting out of order messages?

Another question I had was regarding other clients. Do you use clientside prediction for all the other players also?

Cheers.

50. 2008 March 31
Glenn Fiedler permalink

first question, you'll note that "updatePhysics" internally assumes that deltaTime is a multiple of n frames at fixed framerate, so even though its updating a different amount of delta time, it does so in fixed delta time increments

this is because the time values coming over the network are always multiples of fixed delta time (multiples of 1/60th second for example)

eg. you have fixed delta time of 1/60, so to do deltaTime = 10/60 frames, you do 10 steps 1/60 each

51. 2008 April 25
John permalink

Thanks for the reply, Glenn. I still got a couple of questions. Please bare with me as I'm a newbie to this.

1) As a follow up to #46 and #47, the client needs to be running ahead of the server for n-frames. If the client is pulled back into the latent time stream because of differences between server state and previous client state, the client will be in the same time stream as the server. How can the client be ahead of the server by n-frames again?

2) Because a client is ahead of the server by n-frames, it runs its simulation based solely on his own input. It seems to me that the client will be always wrong on the states of other players because the client doesn't have the input from them until the server update n-frames later. Won't this cause constant snapping of things not controlled by the client disrupting game player and player to player interaction?

Please bare with my poor English. I hope my questions are understandable. And thanks again for the great article and information.

great article and information.

52. 2008 April 25
Glenn Fiedler permalink

1. the client is never pulled back into the latent time stream, we either rewind and replay to get the correction back to client time, or we dont apply corrections on the client (see my GDC 2008 talk on authority management)

2. the client is wrong on the state of other players, you have a few options here, either just approximate them, extrapolated them to current time (i dont like this one), or to leave them in latent time (my preference)

my recommended approach is the client running ahead of the server, and all objects sent from the server to client running in latent time - so client controlled objects are predicted ahead, but server owned objects are in latent time

cheers

53. 2008 April 25
Glenn Fiedler permalink

also for #2 - if you dont want to be physically correct, research "interpolation and extrapolation" as used in networking, it is here (ghosts of remote objects on client), where this is normally used

interpolation/extrapolation is not my normal strategy, but its something to look into for ghosts

cheers

54. 2008 July 14
Jeff permalink

Hey great info, thanks again for sharing!

Two questions if you would be so kind…

1.) You mention that with Havok (and with most physics engines if I am not mistaken) you can not 'manually reposition' individual objects. You have to update the ENTIRE simulation at once. Which begs the question, how are you correcting the position of out of sync bodies (for example when the server sends a 'no you dont, that object is red' message to the client).

2.) I think the method you describe (giving the client authority for certain objects) is the ONLY way to achieve a truly dynamic multiplayer world. The only 'good' way at least . So it seems like we have no choice but to use this same model for non-coop games as well. Thus, what to do about cheating…

It seems like some 'outside the box' thinking is required… Perhaps you could have the server randomly validate updates from the client, checking to see if something 'fishy' is going on (a teleport from one end of the map to the other, ect…). I feel you could build a fence high enough that, while not perfect, could deter the vast majority of cheaters. Any thoughts?

Thanks in advance!

55. 2008 July 17
Glenn Fiedler permalink

you can of course apply individual updates to physics objects, but the point about not being able to rely on a static world and rewind and replay just the player controlled object - thats what i was talking about, you have to update the whole world together, and this implies that objects move and interact (not just player running around a static world)

for #2, authority management is a good way to get the client feeling no latency (being able to client side predict), without needing to rewind and replay to apply corrections from the server to the physics world - anti-cheat could still be done, eg. running metrics on the server to check the client's actions are

valid, verifying hits and kills on the server are correct relative to the client's perception of the world and so on — this is similar to what counterstrike does for locally predicted hits

56. 2008 August 9
Joel Jensen permalink

Thanks for such an awesome article!

57. 2008 October 5
michael permalink

Sorry if i am so noob but why the client correction is important? why we dont just snap the new input and state without replaying the history's moves?

i dont understand the imporantece of replaying the history's moves if they are not rendered during the game.

58. 2008 October 6
Glenn Fiedler permalink

remember that the client is predicting ahead, so if there is a 1 second round trip lag between client and server, the server will send corrections for time t back when the client is at t+1

so unless you wish for the client to be pulled back by RTT each time the server applies a correction, then you must rewind and replay

the alternative is not client side predicting at all, in which case the client feels the RTT latency in movement, but the server corrections can just be applied without replaying moves

cheers

59. 2008 October 10
gustavo permalink

Hey, hi there ! I trying to implement your article into a real app, but i see there is a problem with receiving timestamps. Sometimes when the server's recieved the input from the server, cant handle well the synchronization because the JITTER.

1:my question is: how do you handle the JITTER when recieving timestamps during the receive input of the server.

Using the code below sometimes i get different delta times. very variable deltatimes. Your example source code works well because it has a fixed deltime times every tick, but in a real network the deltime time is variable because the jitter.

2:How do you fix your deltatime so the simulation is stable?

Thank for your time

60. 2008 October 13
Glenn Fiedler permalink

ok so in the context of this article, on the server you simulate ahead based on the delta time difference between the current time for the player, and the time in the packet for that player

this stepping forward *should* always be some multiple of fixed delta time

but it does mean that as packets arrive each player is slightly out of phase of each other

the solution is to not render directly on the server, but instead to have clients running interpolation/extrapolation client for the remote view, so basically, always have a separate dedicated server running, so even if a computer is playing the game and acting as a server, the actual server is in another process

this way, jitter is handled with traditional client side techniques such as interpolation/extrapolation

more details here:

http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking

61. 2008 October 27
pj permalink

Thanks for the great article!

However I still can't figure out the small question here. Let's assume that a client has time t=100 and network latency=120ms.

1) He sends to the server "START MOVEMENT FORWARD" command.
2) Server will receive it after 100 + 120/GameStep (clientT + Latency/GameStep), react and start moving the object
3) Server will reply to the client with current server time, position of the object and action "START MOVEMENT FORWARD".
4) Client will receive it after 100 + (120/GameStep * 2).

According to your code from history.h old moves will be deleted since the received time is ahead from the current client time. User will notice kind of "JUMP".

while (moves.oldest().time<t && !moves.empty())
moves.remove();

Did I miss something?

Thanks in advance!

62. 2008 October 28
Glenn Fiedler permalink

t is the lagged time, so it is RTT behind actual time - therefore the moves don't get deleted until they are no longer needed

for example if your lag is 100ms each way, then moves won't be removed until 200ms at the earliest

cheers

63. 2008 October 29
pj permalink

Thanks for reply. So it means that server must trust the time of the client and use it for calculation?

e.g. client sends "FORWARD COMMAND" with t=100, server receives it on t=110 and use the time of the client with t=100 to apply this command and resend it to other clients?

64. 2008 October 29
Glenn Fiedler permalink

yes, so some limit on the client advancing time +/- needs to be enforced so that obvious cheating cannot be performed

for example you could calculate the approximate time offset between each player character and limit to some deviation around this, or you could track total amount of time advanced over a period of time and if its some % over/under real time advanced then it could be clamped

alternatively you could just boot clients that have poor time advancing characteristics instead of trying to "fix" the time, and say its disconnecting because of poor network conditions

65. 2008 October 29

Glenn Fiedler permalink

also one other point, note that the rewind and replay occurs on the client after the move goes client -> server, then server -> client again i got the feeling somehow (not sure if this is true) that you might think something is occurring on the server when it is actually on the client

to clarify what goes on the server side, each time an update comes from the client, it advances ahead the time difference between the last update and the current one - meaning that each client player sim is slightly out of phase on the server

this complicates weapon hit detection, because you have to account for all the client players being out of phase - but physics interactions between two players are "undefined" anyway given that you are client side predicting, so its no big deal

see the valve article i linked to above for more details, its pretty good

cheers

66. 2008 November 8
pj permalink

Thank you so much for you explanation!

But as I understood it won't work with approach when the client sends only "commands" to the server. I mean, client presses "UP" button and sends it to the server. While he holds this button pressed you do not send any new commands until he holding it pressed?

1. Client: T=100, UP button pressed
…
2. Client: T=210, UP button unpressed

Am I right here?

67. 2008 November 9
Glenn Fiedler permalink

the client sends the current input repeatedly, whether or not it changes - this is because we send input over an unreliable channel, so sending just commands as deltas is not a good idea

cheers

68. 2009 March 23
Kenshin permalink

very usefull,thank you Glenn Fiedler

69. 2009 March 26
Anonymous permalink

Of course floating point numbers are deterministic for a *given platform*. That's how all RTS games work on the PC. Just make sure everyone is using the same version of SSE or even simpler: disable it.

70. 2009 March 26
Glenn Fiedler permalink

yes that technique works fine for synchronous games

but first person shooters and action games can't just sit around and wait for the input to arrive from all other players, so they run asynchronously

in an asynchronous network game, just sending inputs is not enough to keep the simulation in sync - *even if* the simulation is entirely deterministic.

so in this case, i recommend sending state as frequently as possible

of course, the greater the amount of determinism you have, the lower frequency of object updates you can get away with

and finally you can reduce the % of time spent waiting for other players inputs with time syncronization, time stamped packets and a jitter buffer

but even with all of this, you still need to send state — because sometimes, no packets will get through at all for a second or two, and in a FPS you can't just stop the game and wait!

cheers

Leave a Reply

Name :

Email :

Website:

Comment:

Note: You can use basic XHTML in your comments. Your email address will **never** be published.

Subscribe to this comment feed via RSS

Notify me of follow-up comments via email.

Submit Comment

- ## Search

type and press enter

- ## Articles

  - Game Physics
    - Integration Basics
    - Fix Your Timestep!
    - Physics in 3D
    - Spring Physics
    - Networked Physics
  - Networking for Game Programmers
    - UDP vs. TCP
    - Sending and Receiving Packets
    - Virtual Connection over UDP
    - Reliability and Flow Control
  - GDC 2009

- ## Blogroll

- Alex Carlyle
- Alex Evans
- Ben Board
- Bram de Greve
- Brenda Brathwaite
- Brett Paterson
- Brian Sharp
- Christer Ericson
- Dean Finnigan
- Erin Fusco
- James Everett
- Jason Hutchens
- Jim Tilander
- John Passfield
- Jon Jones
- Justin Saunders
- Matt Collville
- Sam Laughlin
- Wolfgang Engel

Friends

- Dave Taylor
- Ewan Spence
- Tim Moss
- Tom Miller

Recent Comments

- Glenn Fiedler on GDC 2009
- Matt on GDC 2009
- Glenn Fiedler on UDP vs. TCP
- Glenn Fiedler on Almost There
- Darrin West on UDP vs. TCP

Support gaffer.org!

These articles don't just write themselves, so if you find my writing useful, and would like to encourage me to write more, make a donation!

Thanks!

Glenn Fiedler gaffer@gaffer.org