Network Sync Transform (Release v4)

Click here for most current/complete version of this document

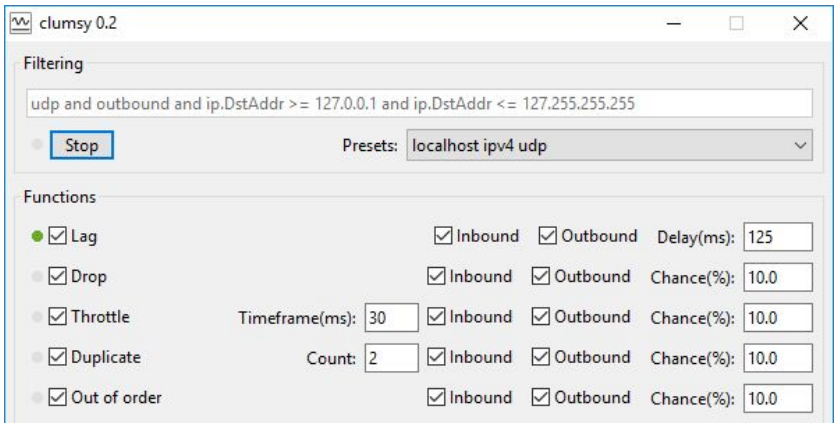
Contact the author at: davincarten@yahoo.com
(Yes, yahoo... though you can reach me at the same name @ gmail - I just don't like putting my email uri into public documents.)

Getting Started Tutorial:
https://docs.google.com/document/d/1M_zvtlqs99xGDuOW7EDqy7MUafs_JDT99HBvTuvtgbw/edit?usp=sharing

Forum started at:
https://redd.it/7f8fdk

I am also regularly in #Unity-Dev on Discord with the screenname **emotitron** if you have questions.
https://discordapp.com/channels/85338836384628736/85593628650504192

I recommend having a network distressor of some kind running while working on networking, in order to more accurately see how your settings will fare when your game is on the internet. Lag, Loss, Jitter, and out of sequence packets will ruin your day if you haven't planned for them. NST was built with Clumsy always running with these settings:



So most adverse conditions are already pretty well handled with the default settings. Feel free to try the built in HLAPI NetworkTransform with these settings - it will reveal why it should never be used.

About the Network Sync Transform

Originally created as a replacement for the very poorly implemented, broken and incomplete Unet HLAPI NetworkTransform component, this asset evolved to become more of a networking engine in of itself.

As a networking asset, it sits somewhere between the HLAPI NetworkTransform and more complete assets like Photon Bolt. Unlike Bolt however there are some very core differences in the networking philosophy applied - some by choice, some because of the nature of UNET and how this asset started life.

Here are some key things to understand about how this asset works.

It does NOT sync using player inputs. This choice is what drives 90% of what this solution is. Unlike networking solutions like Photon Bolt and most AAA titles, NST syncs everything by compressed transforms. With this choice comes a great deal of trade-off. Future versions may include an input sync, but right now everything stems from transform syncing.

There is no master server clock. While all clients generate outgoing updates using the FixedUpdate timing segment, the server does not make any attempts to time align synced objects to a master clock. Instead it it immediately passes incoming updates through to listening clients (with minor alterations as needed for server rule enforcement). Networking apps like Photon Bolt alternatively collect all incoming updates and align them into a master buffer, which allows for a solid frame by frame server picture of the world to be formed. NST makes no attempt to create an authoritative world view like this. It only creates an authoritative history of each object, so that all rewinds are accurate for client.

NST is designed to avoid using buffers any more than needed, so as mentioned above - instead of feeding into a master server buffer, all components update on their own clock so they can be passed along as they occur. Rewind still works (networked objects are rewound using their own history rather than a world history), but what is noticeably different about this approach is that the world will look different to every client and will be different on the server than the clients.

Where this is most obvious is if the server owns any networked objects. It will send out updates for these objects in real time, while incoming updates to the server go on a buffer. The result is server objects will be seen happening on clients later than updates from other clients.

The advantage of this approach is that every object can operate on different network rates. For example, a player object can update every 2 FixedUpdate ticks, and an NPC can update only every 5 ticks. A second advantage is that there is less induced latency of the server waiting to collect and buffer all inputs before sending out a world frame. Instead it immediately forwards all incoming client owner updates to all other clients.

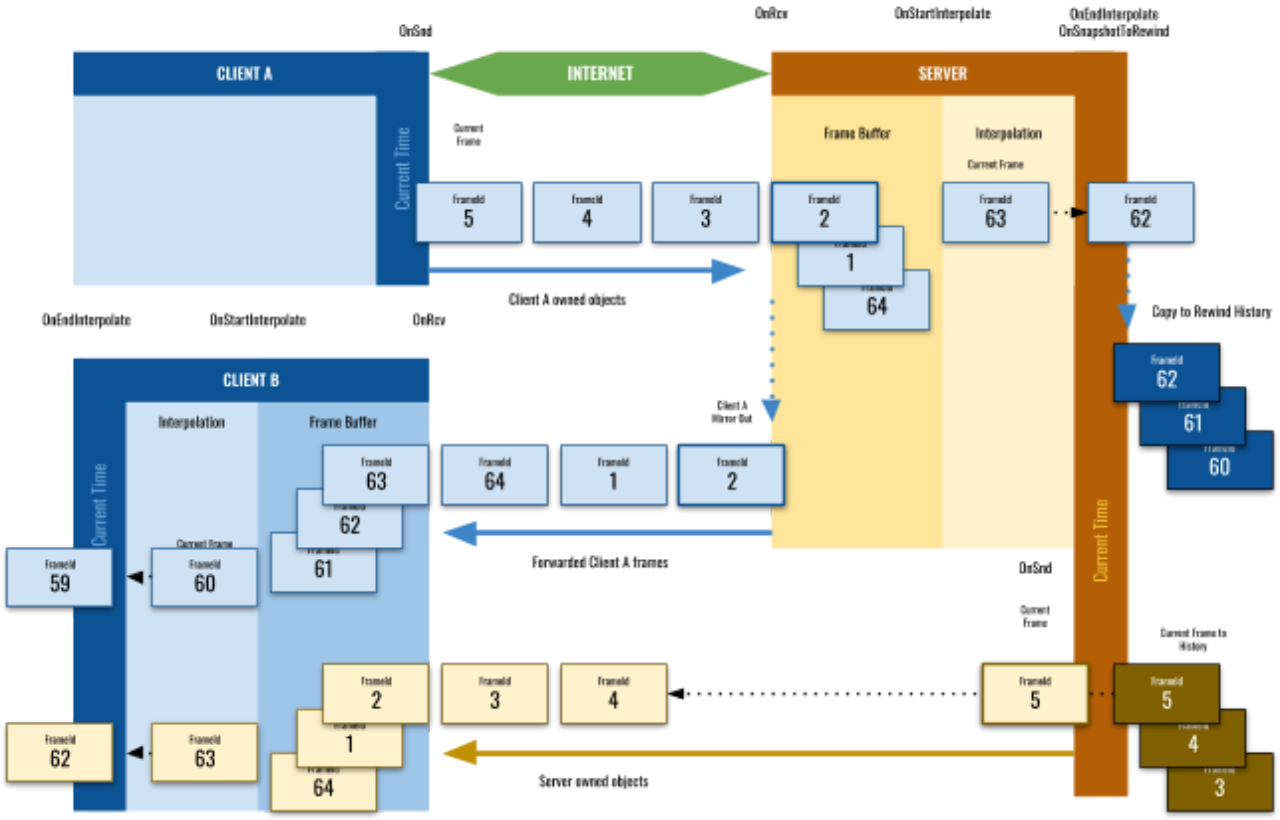
The disadvantages are that this for one makes the math much more difficult. It also means that any server authority has to be passed along to clients after the fact. For example, if a player runs a wall hack and their client claims to enter a wall, the server will not discover this until it runs its own physics simulation until its own client applies the update from its buffer (it runs as a client). It now has to retroactively start applying corrections to subsequent updates to nudge the player back into a legal

position. While less than ideal, this still will break any attempts at cheating, as the wall hackers illegal movement will never register as correct with the server.

NST Order of Execution

- Each **owned** NST runs FixedUpdate() and determines if [`frameTransmitDue = true`]
- NSTMaster Update()
    - NSTMaster Calls MasterPollForFrameUpdates() on all NST objects
        - NST callback iNstPreInterpolate.OnNstPreInterpolate();
        - NST checks if previous interpolation is complete, if Snapshots and advances to next frame.
        - NST applies interpolation/extrapolation
        - NST callback iNstPostInterpolate.OnNstPostInterpolate();
- NSTMaster ReceiveUpdate() - triggered by UNET
    - nst.ReceiveUpdate() called on every NST with an update in the bitstream.
        - INstBitstreamInjectFirst.NSTBitstreamIncomingFirst() callback
          TransformElements are read from the bitstream
        - INstTeleportIncoming.OnRcvSvrTeleportCmd(); (if frame == 0)
        - INstOnOwnerIncomingRootPos.OnOwnerIncomingRootPos()
          Used by ServerAuthority component to get the position error value
        - INstOnSvrOutgoingRootPos.OnSvrOutgoingRootPos();
          Collects server-side interventions
        - INstBitstreamInjectFirst.NSTBitstreamMirrorFirst();
          If (isServer) forward server modified frame to all clients
        - INstBitstreamInjectSecond.NSTBitstreamOutgoingSecond() callback
          NSTAnimator values are read from the bitstream
        - INstBitstreamInjectThird.NSTBitstreamOutgoingThird() callback
          NSTAnimator values are read from the bitstream
        - INstOnRcvUpdate.OnRcv()
        - If incoming frame is the frame currently being interpolated:
          INstOnStartInterpolate.OnStartInterpolate();
- NSTMaster (if isServer) sends modified updates returned from ReceiveUpdate() to all clients

NST Frame Lifecycle



https://docs.google.com/presentation/d/1q8zOIFuYpsZr_31UHVdFTrc5n-j3G_azO6tsN5Eb5XI/edit#slide=id.p

The **NetworkSyncTransform (NST)** consists of a set of components that work together on top of the Unity UNET HLAPI as a replacement for the mostly broken NetworkTransform that is part of the HLAPI.

Some features of the free core Network Sync Transform:

1. A large range of compression levels for position/rotation data.
    a. User defined position resolution- NST will determine the fewest bits required per tick to achieve that resolution at the current map size. Set the min resolution once and apply NSTMapBounds to the map, and NST handles it from there.
    b. Optional Delta-like frames that drop the upper bits when they have not changed.
    c. Quaternion compression that ranges from 3 bytes to 7 bytes. 5 bytes per tick is the default and is extremely accurate. (raw quaternions are 16 bytes).
    d. Per-axis Euler compression allows specific axis and ranges of motion to be defined. For example an first person shooter likely only needs 360° of rotation on the Y-axis and <180° on the X-axis.
2. Custom Messages that allow user data to piggyback on the NST updates. Useful for events that rely on having the position/rotation, such as weapon fire.
    a. An experimental struct serializer is included for custom messages (not tested on all platforms) so you can avoid having to deal with serialization.

3. Frame buffering to reduce hitching from net jitter/loss. User adjustable to find the balance between induced latency and the desired smoothing of internet noise.
4. Server initiated Teleport.
5. Adjustable max number of Networked game objects to reduce header sizes to smallest possible size. No need to send a 32bit ID every tick if your game only will have 8 synced player objects (which instead only requires 3 bits per update).

Available addons for purchase:

1. **NST Elements Add-on**
    a. Up to 32 child positions/rotations objects can be synced per NST, such as turrets, heads, arms, wheels, etc.
    b. Works with NST Rewind Add-on, so all child elements are rewound, applied and tested.

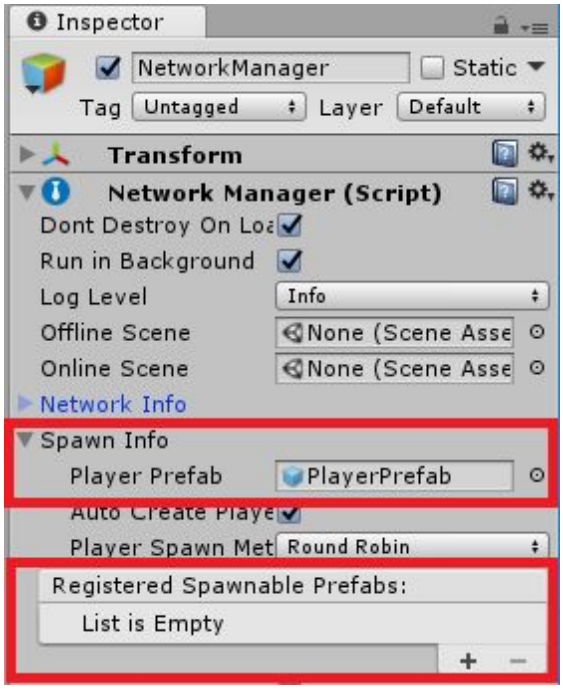
2. **NST Rewind Add-on**
    a. Comes with components that can handle the entire process of firing hitscan weapons, propagating them over the network, testing them against the server rewind, initiating graphics and sound effects, as well as triggering callbacks on server and client.
    b. Uses standard physics objects for raycast and overlap cast tests.
    c. Hitbox Layers can be assigned to gameobject children for things like critical hit colliders and such.
    d. Works with the Child Elements Engine, so all child elements are rewound and tested as well.

The NetworkSyncTransform component

The core component of the NetworkSyncTransform asset is the NetworkSyncTransform component itself. The **NetworkSyncTransform** should be added to any prefabs you want to sync the position/rotation of over the network. **Place this just how you would place the UNET HLAPI NetworkTransform component.** This will replicate the position and rotation of the object on the client with authority to all other machines.

There are 3 required components in scene for the NetworkSyncTransform (NST) to work.

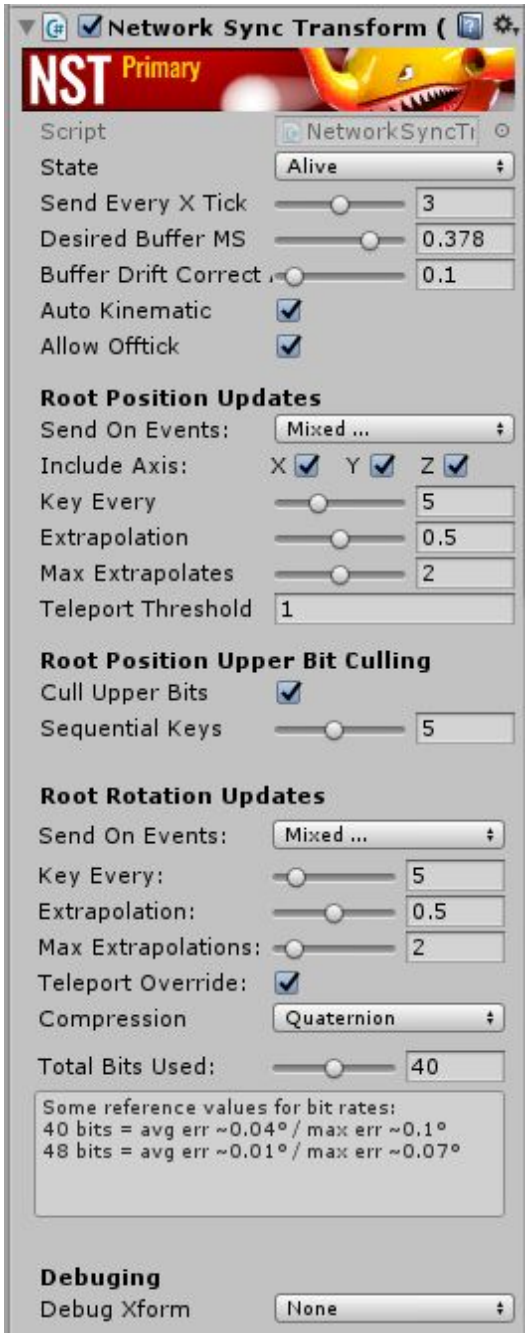
1. **The NetworkManager**
   This is needed since this entire system current rides on top of the UNET HLAPI. NST prefabs will automatically try to register themselves with the NetworkManager as you work, but if for some reason they do not, here is where you register them with the UNET NetworkManager.



2. **NSTSettings**
   A singleton that collects your preferences that apply to all NetworkSyncTransforms(NST). This will be created automatically when you put a NST into your scene, and the default settings may work fine - but you will likely want to tinker with them as you get familiar with the NST asset.

3. **NSTMaster**
   A singleton that polls all NetworkSyncTransform(NST) objects owned by the running instance for updates, which it serializes into one update - and parcels out incoming updates to the correct NSTs. You don't need to do anything with this. It should automatically be added, and will just sit there doing its work.

NetworkSyncTransform (Main Component)



This is the primary component which is placed on the root of any prefabs you want synced. Some working knowledge of how UNET's HLAPI works is assumed. This is a replacement for the built in NetworkTransform, and needs to be used in a similar fashion. The object this is on MUST be a prefab, and it MUST be spawned with the HLAPI. (more info on spawning in the HLAPI)

State
This current Alive/Dead state of this NST. This uses the NSTState enum, which is actually a two bit mask value actually, consisting of **`[isActive][isVisible].`**Currently only the Alive and Dead states are used. The extra visibility bit is for future functionality. State can be monitored by any component on this prefab by using the **INstState** interface. You can see an example of this is in the NSTSamplePlayer.cs code.

Send Every X Fixed
This is a very critical setting. The NST uses the physics engine as its internal clock, since much of the functionality of the NST surrounds the physics engine. Sending network updates on every tick would be excessive, so it is common to run this about about 1/3rd of the physics tick rate (meaning every three ticks of the physics engine is 1 network tick). The physics engine by default runs at 50 ticks per second, so setting this to 2 would result in a tickrate of 25.

Desired Buffer MS
Number of milliseconds of buffer NST will try to maintain. The higher this number, the more net loss and net jitter will be eliminated. However it will induce this many milliseconds of perceived latency. 100ms - 150ms is a good starting range.

Buffer Drift Correct Amt
This adjust how aggressively the buffer will try to maintain its size. A value of .1 should work for most situations. Too high and network objects may exhibit rubber banding. Too low and they may need to resync more often (resulting occasional position lurches).

Auto Kinematic
Syncing over the network often involves the local object with authority using a rigidbody for physics and for collision detection. However when that movement is being reproduced on the server or other clients, no physics should be applied. How to manage all of this is beyond the scope of this paragraph, but no worries - if you are just getting started set this to auto and it will make a pretty good guess on how it should be handled.

Allow Offtick
If you use Custom Events and Teleport, disabling this will help reduce traffic, but it will create a slight delay on the localplayer between creating the event and the event being sent to the server. Enabling this allows ‘offtick’ NST updates to occur instantly when you add a custom message to the queue - sending an update immediately to the server rather than waiting for the next network position tick to piggyback on.

Root Position and Rotation Common Settings

Include Axis
These look a little different for each, but the root position, position elements and rotation elements all allow for you to selectively choose which axis should be transmitted. Rotations also have the Quaternion option. The quaternion compression is VERY good and is recommended for any objects that rotate freely (such as balls or space ships).

Send On Events
This reduces data by only sending the events of this type.

- *None* - no regular updates. Only keyframes are sent. Set Keyframes to zero as well to disable.
- *Every Tick*- send update every time.
- *On Changes* - send update if there are no changes in position/rotation.
- *On Events* - send update when a custom message or teleport occurs.
- *On Teleport* - send update for this transform when this object teleports.
- *On Custom Msg* - send update when a Custom user message is attached.
- *On Rewind Cast* - send update when a Rewind Cast is attached.

Keyframe Rate
How often in seconds a keyframe will be sent. This is a full update and will override the upperbit culling setting.

Extrapolation
When the frame buffer runs empty, this controls how objects behave. At 0 they will stop in place until a new update arrives. At 1 they will will continue in direction they were during the last update. High extrapolation settings are great for things that tend to have a lot of inertia, like wheels or ships. Lower extrapolation is better for objects prone to rapid and unpredictable direction changes. Too much extrapolation will cause rubberbanding.

Max Extrapolates
The number missing frames that will be extrapolated in a row after the buffer runs empty. Too low a number and buffer underruns will cause objects to hang and stutter. Too high and objects will keep travelling when there is an extended service interruption.

Teleport Override
When selected, Server initiated teleports will include the current positions/rotations of all synced elements as they are on the server. This is used for any items that server will reset when an object is teleported. If unchecked, the relative positions/rotations of the element will persist after teleport.

Root Position Settings

Cull Upper Bits
When enabled this drops the top 1/3rd bits of each compressed axis position when it hasn't changed. This does however make movement more fragile under lossy/jittery network conditions. There are some hard coded attempts to reduce that issue (such as every time the upper bits change it will send the full packet for 7 ticks to try and ensure the new upper bits have arrived, before allowing the upper bits to be dropped again).

Sequential Keys
If '*Cull Upper Bits'* is enabled - This is the number of ticks that a full position will be resent to ensure that even with lossy internet connections the most recent upper bits makes it to all clients.

Teleport Threshold
Large movements will trigger a teleport (rather than a smooth lerp). This threshold determines the number of units a jump between current frame position and the new frame position is allowed before it happens.

Root Rotation Settings

Local Rotation
If this is a child object, using localRotation rather than rotation can help reduce traffic and create better stability. If this is the root object leave this unchecked, as it should move in world space.

Compression Type

- **Quaternion Compression** uses smallest three, which is interesting but I won't go into it here. What is important is to select a bitrate that gives an accurate enough rotation. Testing is your friend here, but for a starting point I recommend 40 bits per tick. This produces an accuracy of about .1°. If more accuracy or less is needed, trial and error to see what the lowest setting you can get away with is. If you start getting too low (below about 24 bits) erratic rotations may occur.
    - **Total Bits Used** - There is only one setting for quaternion compression.

- **Euler Compression** (Any combination of x/y/z) restricts data to only certain axes and rotation ranges ranges. These allow you to selectively set bit depths and ranges for each of the 3 axis or rotation indepenently.
    - **Bits** - Sets the number of bits used for each axis,
    - **Axis Ranges** - Limits the range of motion of this axis. Useful for reducing data. Every time you cut the range of motion in half, data is reduced by one bit.
      360° with .35° increments = 10 bits ... so
      180° with .35° increments = 9 bits
      90° with .35° increments = 8 bits
      45° with .35° increments = 7 bits

Also limiting ranges is useful for avoiding values crossing through gimbal lock, by disallowing offending angles (such as looking straight up).

Debugging

Debug XForm
The Debug Transform is a colored cross widget that can be used to see the raw values of updates. It is only visible in the Editor (It is hard coded with a `[Conditional("UNITY_EDITOR")]` to ensure it has minimal impact on any release builds, without requiring any extra steps to remember.

NSTSettings

Header Sizes:

Bits For NSTId
The number of bits that will be used with each packet to describe which NST the position and rotation data belongs to. You want to set this as low as you can if you want to reduce network traffic.

Max Networked Objects (Read Only)
The number of unique IDs available at the currently selected bit setting above. This is the max number of NSTs that can exist on the network at a time. Any attempts to add more beyond this number will break things. So be sure you have this set high enough.

Bits For Packet Count
The number of bits that will be used to identify packets. The options are 4, 5 or 6. Going as low as 4 (15 frame rotating buffer) isn't recommended unless you are using a VERY low tickrate. The 1 bit savings is available, but opens the door for havok if network conditions are bad.

Frame Count (Read Only)
The number of unique packet IDs in the rotating buffer, determined by the slider above.

World Vector Compression

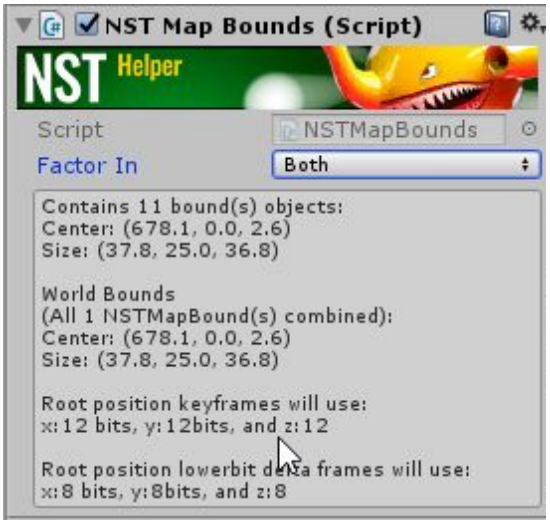
Min Pos Resolution
Another very critical value. This determines how fine the increments of movement can be. I would recommend putting this about about 1/50 if the size of your object. So if your player is 2 units tall… set this to 100.

Default World Bounds
If no NSTMapBounds are in the scene to define the world dimensions, these values will define the limits of the world that objects can traverse. The smaller the extents and resolution, the lower the network data.

Debugging

Log Warnings - Warnings will appear in the Debug.Log
Log Testing Info - The Debug Log will get very busy with info about the NSTs
Log Data Use - Summaries of sent and received data sizes will appear in the Debug.Log

NSTMapBounds



The **NSTMapBounds** component is used to define the size of the world (Root compression needs a fixed world size in order to to make packet sizes as small as it does). When enabled this will find all <MeshFilter> or <Collider> components recursively on the game object this is attached to, and will sum all of the found bounds to a composite bounds that defines the world.

The fallback if no NSTMapBounds have been placed in the scene is the Bounds defined in NSTSettings. If you know the size of your world you can set it there.

Factor In
Choose if MeshRenderer components and/or Collider components will have their Bounds used in the calculations.

Custom Messages

The NST supports piggybacking your own custom data on the back of the NST ticks. This primarily is meant to be used for player actions (like weapon fire) that are tightly linked to player position/rotations. For regular data syncs that don't care about the players transform you can inject them into the bitstream using the INstBitstreamInject interfaces.

Custom Message Methods

Custom messages are added by calling

```
myNST.SendCustomEvent(customdata);
```

Or if there is only one synced object on this machine you can use:

```
NetworkSyncTransform.SendCustomEventSimple(customdata)
```

It comes in two overloads. One that takes a raw byte array of your making, or one that accepts a struct and turns it into a byte array using marshalling.

```
public void SendCustomEvent(byte[] userData)
public void SendCustomEvent<T>(T userData) where T : struct
```

This static function will only work if there is a local player, since only players with authority will be sending out NST transform updates.

Custom Message Events

Custom messages cause the NetworkSyncTransform to generates the following events:

- ```
  OnCustomMsgSndEvent
  (
          NetworkConnection nstOwnerConn,
          byte[] bytearray,
          NetworkSyncTransform nst,
          Vector3 rootPos,
          List<GenericX> positions
          List<GenericX> rotations
  )
  ```

  This is fired on the originating machine when a custom message is being sent. The position and rotations are post compression at the time the message is sent, so they contain the same lossy errors that will be sent over the network. This is useful for making sure that if you perform an action locally without waiting for the server message to bounce back, that it will be in complete agreement.
  NOTE: **the GenericX struct implicitly casts to Vector3 and Quaternions**, so use it just as you would use those. This generic struct is used internally to keep code for position and
  ```

rotations unified.

- **OnCustomMsgRcvEvent (** *same as above ...*)

  This is fired on all receiving clients/servers when a custom message comes it.

  NOTE: HOSTS DO NOT RECEIVE THEIR OWN MESSAGES. So you may need to put in some logic to account for that.

- **OnCustomMsgBeginInterpolationEvent (** *same as above …* **)**
  OnCustomMsgEndInterpolationEvent (***same as above …* **)

  This is fired on all receiving clients/servers when a custom message is APPLIED. Unlike OnCustomMsgRcvEvent which fires immediately when the network message is received, this fires after it comes off the frame buffer and is being interpolated to.

  NOTE: HOSTS DO NOT RECEIVE THEIR OWN MESSAGES. So you may need to put in some logic to account for that.

Teleport

To teleport simply call:

```
myNST.Teleport(pos);
```

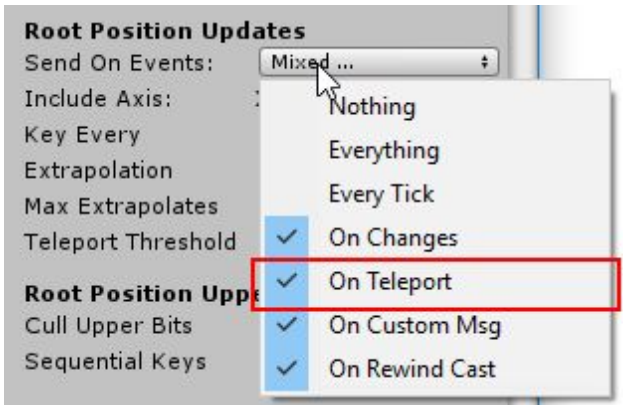
This can currently only be called on the server. Calls on from clients will be ignored.

`Teleport()` causes the server to send out an immediate update with an UpdateType flagged as teleport and with a frameId of `0`. It then sets:

```
SvrWaitingForClientTeleportConfirmation = true
```

The server will modify all incoming updates from owners to indicate an UpdateType of Teleport before mirroring them back out to all clients. It will ignore all incoming position and rotation updates flagged as TeleportOverride while it waits for onwer confirmation, and will continue to mirror out the Teleport transform values until it receives a confirmation from the owner.

The owner will respond to the command by sending out updates with the UpdateType flag of Teleport, and include full keyframes for any elements with **Teleport** selected in **Send On Events**.



Interfaces

NST Timing Callback Interfaces

The Network Sync Transform is loaded with callback interfaces that can be used to access all of the various timing segments of the component. These can be used to:

- Read/Write to the bitstream at at specific points
- Run code before or after events that occur in the NST

In the INSTCallbacks.cs file you can see all of the callback interfaces the NST makes use of. Any component in the hierarchy of an NST networked object that makes use of these interfaces will be found at startup and will be registered as a callback.

You can see examples of this in action in all of the NST Sample files, such as **NSTSampleWeapon** and **NSTSampleHealth**.

Frame Buffer and Frames

The Network Sync Transform works by reading and writing to a circular buffer (known as the Frame Buffer), which is a Frame[] with and assortment of methods used to Read, Write, Interpolate, Extrapolate and otherwise make use of these frames.

The number of frames in this circular buffer is set in NST Settings.