Network Sync Transform (Beta 3)

This is the documentation for the Beta 3.0 release.

THIS IS STILL VERY MUCH IN BETA!!
Feel free to email me with any issues you are having at davincarten@yahoo.com.

Most current documentation available at:
https://docs.google.com/document/d/180v50xK48KhaVfNvxp2knAeXqU17P7VMUkGBM2qgGbU/edit?usp=sharing

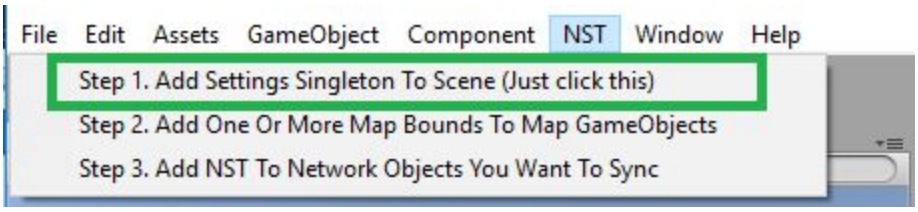
The **NetworkSyncTransform (NST)** set of components work together on top of the Unity UNET HLAPI as a replacement for the mostly broken NetworkTransform that is part of the HLAPI.

Some features of the Network Sync Transform:

1. Large range of compression levels
2. Complete rewind with reconstruction of object, colliders and transforms.
3. User defined root 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.
4. Optional Delta-like frames that drop the upper bits when they have not changed.
5. 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).
6. 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.
7. Up to 32 positions and 32 rotations for child objects can be synced per NST, such as turrets, heads, arms, etc.
8. Custom Messages that allow user data to piggyback on the NST updates. Useful for events that rely on knowing the current position/rotation such as weapon fire. An experimental struct serializer is also included (not tested on all platforms) so you can avoid having to deal with serialization.
9. 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.
10. Network Simulation to test how things will behave under extreme loss/jitter/out-of-order packets/latency conditions.
11. Teleport that can be initiated by the player with authority or the server.
12. Adjustable max number of NST objects to reduce header sizes. No need to send a 32bit ID every tick if your game only will have 8 synced player objects (which would only require 3 bits).

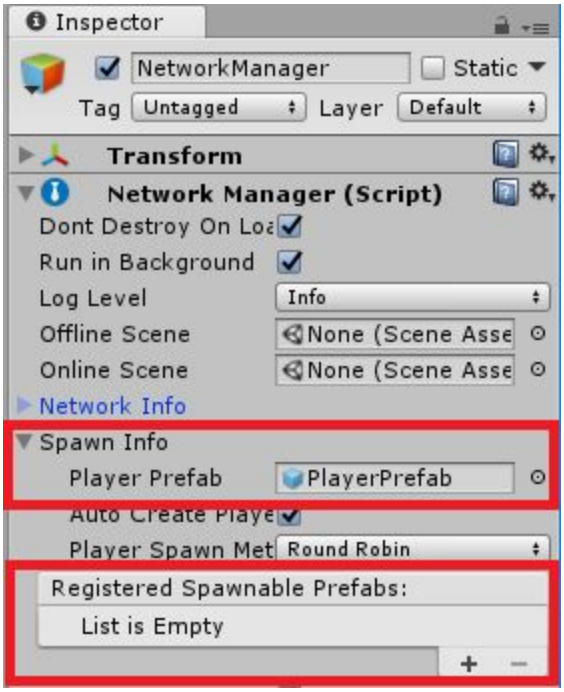
There are 3 required components in scene for the NST to work.

1. The **NSTSettings** component needs to be in the scene. It is a singleton where you adjust the universal compression settings, tick rate, thresholds, etc.



2. The **NetworkSyncTransform** which goes on 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.

   NOTE: This objects must be spawned by the HLAPI in order to work, just as with the [UNET NetworkTransform](). (shown below)



3. The **NSTMapBounds** component is used to define the size of the world. Placing this on the root of your game map GameObject will find all <MeshRenderer>s that are inside of that GameObject (this search is recursive) and will add all found <MeshRenderer>s.bounds to the composite bounding box that defines the world size. Position compression needs this in order to work. MAKE SURE YOU HAVE NSTMAPBOUNDS APPLIED TO AN OBJECT OR OBJECTS THAT DEFINE THE LIMITS OF YOUR WORLD.

   (note: This isn't TECHNICALLY required, as there is a fallback compression method of half-floats if no bounds are defined. But you really don't want to use that... trust me.)

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 Nst 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 number packets. The options are 4 or 5. I recommend 5. The 1 bit savings is available, but with that minor data reduction you cut your numbered packets from 32 to 16… which opens the door for havok if network conditions are bad.

Packet Counter Range (Read Only)
The number of unique packet IDs, determined by the slider above.

General Settings:

Allow Offtick
If you use custom events, 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.

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.

Buffer Drift Correct Amt
How aggressively the buffer will add or remove time from each frame in order to get the buffer back to its desired size. Too aggressive and movement may get a bit janky, not aggressive enough and there is the risk of the buffer getting too big or small at times.

Use Rewind Colliders
This enables the creation of the rewind collider system. When this is enabled a dedicated layer is used for the creation of ghost copies of all gameobjects in the scene containing the NST component. These allow the use of rewind test functions.

Override Fixed Time
This is a convenience that simply lets you change the physics rate here rather than in the Physics menu.

Min Position Resolution
The minimum resolution of objects using global position compression. This is in X per unit. So a value of 100 = .01 units of accuracy.

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.

Debugging

Log Warnings / Testing Info
Turning this off reduces quite a bit of pointless work if no one can see the debug log. Turn it off for your production build.

Enable Net Simulation
Jacks up the data coming out of clients to simulate jacked up internet connections.
TURN THIS OFF FOR RELEASES. SERIOUSLY. I'll automate that later, but for now you will need to.
Clumsy is a superior way to test for distressed internet conditions anyway, I recommend using this if you are on the PC.
https://jagt.github.io/clumsy/

- **Latency Simulation**
  Milliseconds of induced latency from clients.

- **Packet Loss Simulation**
  0 = no packet loss
  100 = full packet loss

- **Jitter Simulation**
  X milliseconds of latency will randomly be added to each packet (in addition to the Latency Sim ms). Making this number greater than the current (Time.fixedDeltaTime * Send Every X Rate) value will give you out of order packets (which is something you probably want to test for). If that math seems heavy, don't bother, just crank it up a bit until you see that packets are coming in out of sequence. Crank too much and you will exceed the NSTs ability to clean up the mess (it's only human).

NetworkSyncTransform

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. 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)

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.

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.

Rewind Hit Layers
If "Use Rewind Colliders" is enabled in NSTSettings, all gameobjects with NST on them will create a simplistic clone of themselves in the scene using only dummy objects and colliders. By default it will copy ALL of your colliders and apply them to the rewind hit test object. You may want to disable some layers from these tests, leaving only the layers that are actually meant to be used as a hitbox.

Root Position, Position Elements, and Rotation Elements

The Root Position is a special position element. It expect the object to not be a child and to move around inside of the map area defined by NSTMapBounds. Position Elements on the other hand operate on the localposition of the object, and are designed to work for child objects. Rotation elements can be set to either local or global space for their rotations.

Root Position Sync (Root Position only)
Disable this for objects that do not move around in the world. Such as if an object only rotates, or if the object is a child of another object. For child objects add a Position Element to the NST and sync position using that.

Include Axis (Position and Rotation)
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 Culling (Position and Rotation)
This reduces data by only sending the events of this type.

- *Always* - no culling.
- *Changes* - will send updates if there are no changes in position/rotation.
- *Events* - will send updates when a custom message or teleport occurs.

Keyframe Rate (Position and Rotation)
How often in seconds a keyframe will be sent. This is a full update and will override any other culling settings (such as upperbits culling).

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

Extrapolation (Position and Rotation)
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.

Cull Upper Bits (Root Position only)
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 (Root Position only)
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.

Position Element Specifics

All position elements are moved by their transform.localposition. This is primarily meant to handle child objects of the root object. The root object is best handled with the root position sync.

Compression Type
None - uncompressed floats - this is just for testing, DON'T send these over the network for real please.
Half Float - These are here just for the sake of completeness. These have a fixed size of 16 bits per axis.
Local Range - This is what you really will want to use. Once you set your ranges and resolution correctly you will have TINY data rates.

Axis Ranges
Enable/disable each axis here. If Local Range is the selected compression (and it should be) you will see three fields per axis.
Min - smallest value expected - in standard units
Max - largest value expected - in standard units
Res - resolution in terms of x subdivisions per unit. For example 100 results in a precision of .01 units.

Rotation Element Specifics

You can sync up to 32 different object rotations per NST (please don't try to test that though, that's absurd). Uses for this would be one rotation for the root Tank, Spaceship, Submarine or Person, and other rotations for child objects like turrets, heads, arms or anything else attached to the root. Each individual rotation can be of a Quaternion type (full rotation) or a Euler type with any combination of the X, Y and Z axis being synced. Each axis can be restricted to a range as well to further reduce network traffic. This is useful for objects that have a limited range of motion, such as a player who can rotate 360° on the y axis, but only 180° on the x axis.

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.

- **Euler Compression** may be able to get reductions in bandwidth by using various Euler combinations. These allow you to selectively set bit depths and ranges for each of the 3 axis or rotation.

Axis Ranges
The Euler compression types allow for ranges to be limited, allowing for more data reduction.

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.

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 be called on the player with local authority, or on the server. Calls on other clients will be ignored.

If you only have one local NST object and can't be bother keeping track of the NST instance, you can simply call the static version:

```
NetworkSyncTransform.TeleportLclPlayer(pos);
```

Note: Support for pushing rotations with teleport in the works. Since multiple rotations are supported this will require a little more work than just adding rotation.

Rewind

Rewind can be accessed on the server simply by calling:

```
emotitron.Network.NST.RewindFrame rewframe = myNST.Rewind(float rewindTime);
```

The rewind struct consists of :

```
public int packetid;
public float endTime;
public Vector3 rootPosition;
public List<Vector3> positionsElements;
public List<Quaternion> rotationElements;
```

The Rewind engine also has some methods (which I will be expanding on over time) added for testing against rewound hitboxes. These are very simple to use, but getting the correct NST or NetworkConnection to pass to them will require some attention to detail. Remember that what you are passing to these is the CLIENT THAT WE ARE REWINDING FOR NOT THE NST THAT IS BEING REWOUND. This means in the case of an FPS we want the NST/Conn of the shooter that has told the server “I shot this guy”. Make sure you are passing this an object which that player has authority over, or the connectionToClient for that player, and not some other player by accident. For convenience, all of the Custom Message Events carry the connection of the authority owner of their NST/NetworkIdentity. In most cases this is what you will be rewinding against (since only the owner with authority should be making any claims about what that NST has done).

Rewind Test Methods

```
targetNST.TestHitscanAgainstRewind(NetworkSyncTransform rewindForNST, Ray ray)
```

Where *rewindForNST* is an NST the player we are rewinding for *hasAuthority* over, or

```
targetNST.TestHitscanAgainstRewind(NetworkConnection rewindForConn, Ray ray)
```

Where *rewindForConn* is the connection to the player we are rewinding for.

One way to get the correct NST is to use:

```
NetworkConnection conn = NSTofThePlayerMakingAClaim.NI.clientAuthorityOwner;
```