NST Rewind v5 Add-on Documentation

This is an Add-on for the free Network Sync Transform asset found on the Unity Asset Store.
The core is included in this asset, so there is no need to download it, but you may want to keep an eye on it for any fixes that get made over time.

Click here for the most current version of this documentation

Doxygen docs for Rewind Add-on

This documentation is in the process of being written - so if you have any questions, email me at davincarten@yahoo.com (or gmail.com).

Rewind Overview

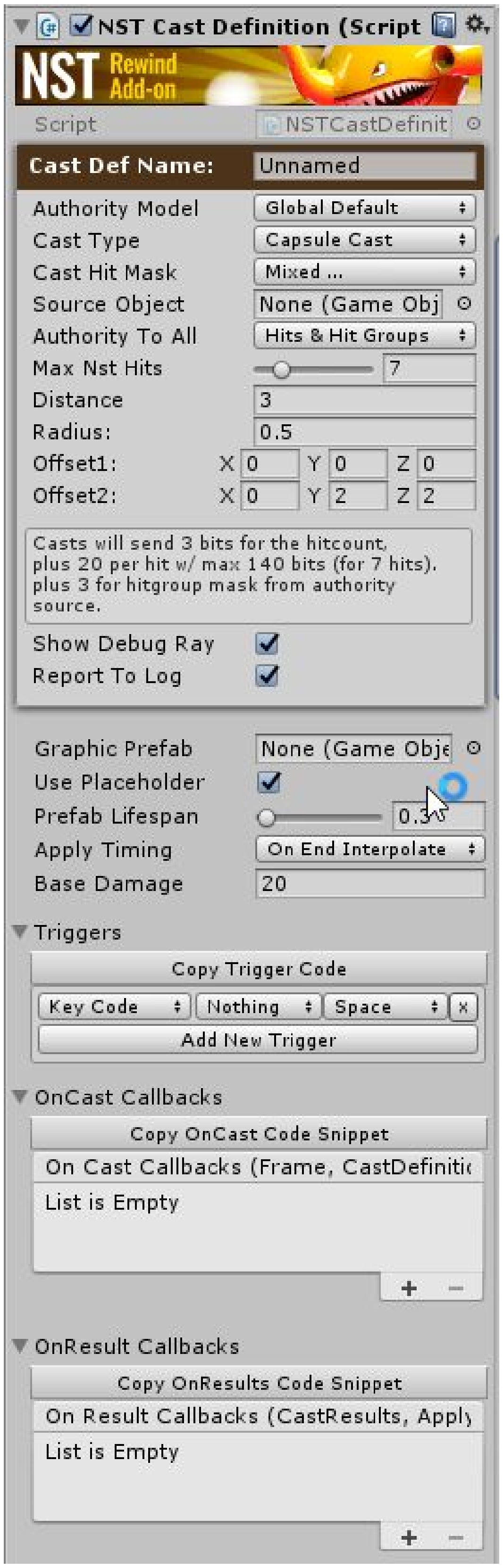
Rewind exists primarily as an anti-cheating measure. It is the act of the server rolling back time on networked objects in order to test the results of actions performed by clients. The most common example of this is for testing hitscan weapons (which this asset primarily focuses on as well). If player clients could be fully trusted, we could simply have weapon fire on clients and they would report who they hit to the server and have their damage/score registered. This however makes cheating extremely easy, as they very simply could just send messages to the server saying “I shot everyone, they are all dead” and it would have no choice but to believe them.

Server Rewind recreates shots by rewinding time by half of the RTT (Round Trip Time) to the player that is making the claim (adding in various buffer sizes and interpolation times). It then recreates the raycast (or overlap cast, spherecast, etc) from the player position/rotation at the time of the cast, and tests for hits on the rewound objects.

This kind of testing makes cheating FAR less trivial, as now to cheat you can no longer just inject the bytes needed to indicate hits, you actually have to code a cheat that tracks other players positions, accounts for obstacles, and takes over the players aiming. While still doable, this is a much more difficult task.

NST Cast Definition

This component is the easiest way (and the only way I recommend) to get going with rewind. It handles all of the communications between clients and server and exposes callbacks for your own custom reactions at each step in the process.



NST Cast Definition (Script)

NST Rewind Add-on

Script	NSTCastDefinit
Cast Def Name:	Unnamed
Authority Model	Global Default
Cast Type	Capsule Cast
Cast Hit Mask	Mixed ...
Source Object	None (Game Obj
Authority To All	Hits & Hit Groups
Max Nst Hits	7
Distance	3
Radius:	0.5
Offset1:	X 0 Y 0 Z 0
Offset2:	X 0 Y 2 Z 2

Casts will send 3 bits for the hitcount, plus 20 per hit w/ max 140 bits (for 7 hits). plus 3 for hitgroup mask from authority source.

Show Debug Ray	✓
Report To Log	✓
Graphic Prefab	None (Game Obje
Use Placeholder	✓
Prefab Lifespan	0.3
Apply Timing	On End Interpolate
Base Damage	20

Triggers

Copy Trigger Code

Key Code | Nothing | Space

Add New Trigger

OnCast Callbacks

Copy OnCast Code Snippet

On Cast Callbacks (Frame, CastDefiniti

List is Empty

OnResult Callbacks

Copy OnResults Code Snippet

On Result Callbacks (CastResults, Apply

List is Empty

The top section of the NSTCastDefinition component is the Cast Definition itself. This is a predefined Raycast, Spherecast, Boxcast, Capsulecast, OverlapSphere, OverlapBox or OverlapCapsule. By defining this up front, network communications about casts only require a couple bits to describe a cast.

First, a player triggers the cast. They can do this by calling the **NetworkCast()** method of the NSTCastDefinition component in script, or by indicating a trigger under >Triggers.

The OnCast(Frame, CastDefinition) callback interface is fired on any root/child components that implement the INstCast interface, executing any code that needs to happen on the local client. This callback is useful for triggering graphics and such. It is preferable to put any trigger code here, rather than at the point of calling NetworkCast(), because this fires when the cast actually occurs (which depending on your settings may be milliseconds later than when the player triggered the action), and includes the exact data being sent to the server, including any position/rotation errors that result from compression.

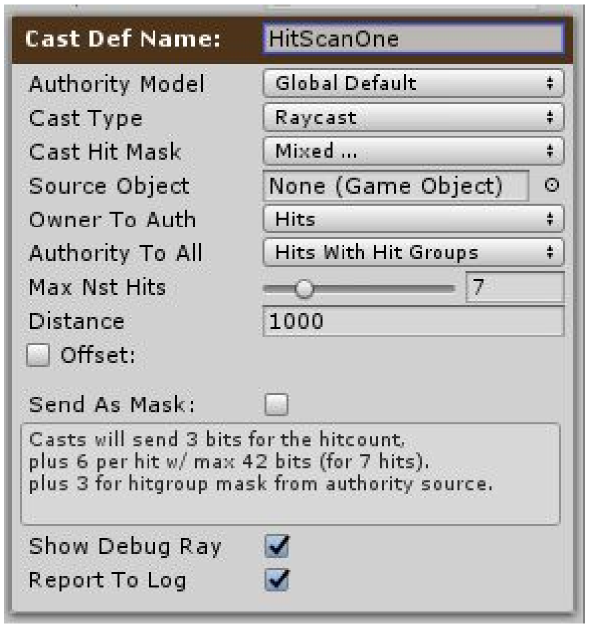
NOTE: The current implementation collects the NstIds of objects it hits. Those IDs are also passed to the server along with the CastDefinition.id. This creates a slightly larger packet, but reduces the workload for the server (as it will only rewind the objects that the client claims to have hit). Future versions will make a cast only option available.

When the **server** receives this cast command (it will be attached to an NST update) - it immediately rewinds and recreates the cast on the server, producing a CastResults object, and it fires the **OnCastResults** callback on any children with the INstCastResults interface.

Depending on the setting of “Replicate To” in the CastDefinition, the server will modify the CastResults sent out to clients accordingly. The server tests the CastResults it got from the client/player, and can send just the CastDefinition id to clients, or the resulting hits (which will produce more network traffic, so only use that setting if you actually intend to make use of that list of hits). OnCastResults passes a struct called CastResults

CastDefinition

You can use just the CastDefinition class for your own implementation of rewind tests. This class defines a cast (which can be a Raycast/Spherecast/OverlapSphere/Etc) and gives it a name/id. This definition is used by networking to invoke a cast with a very small amount of data, as it only needs to pass a few bits of data to indicate a cast event.



Cast Def Name:	HitScanOne
Authority Model	Global Default
Cast Type	Raycast
Cast Hit Mask	Mixed ...
Source Object	None (Game Object)
Owner To Auth	Hits
Authority To All	Hits With Hit Groups
Max Nst Hits	7
Distance	1000
Offset:	☐
Send As Mask:	☐
Show Debug Ray	✓
Report To Log	✓

Casts will send 3 bits for the hitcount,
plus 6 per hit w/ max 42 bits (for 7 hits).
plus 3 for hitgroup mask from authority source.

Cast Def Name - This is the lookup name for easily referencing this cast definition.

Authority Model - This is a placeholder for future mixed server/client authority models. I recommend leaving this alone as its meaning and implementation is still being worked out.

Cast Type - The type of test this definition will use

Cast Hit Mask - Which physics layers in the scene this test ‘sees’. Ideally you will only want to include layers that NST objects will use to avoid wasted CPU power testing layers that will never contain an NST that can be hit.

Source Object - which root/child of the prefab is considered the source where the test (raycast/overlapSphere/etc) originates. If left blank, the object this NSTCastDefinition component is on is used.

OwnerToAuth - sets how much hit information from the owner is sent to the server (only applicable if we are not running as OwnerAuthority).

AuthorityToAll - sets how much hit information is sent along with CastDefinition events from the authority to all clients.

Max Nst Hits - The maximum number of hits that can be sent with a cast event. Reducing this number to the bare minimum required can shave a bit or two off of your network traffic on cast events (by reducing the bits needed for the hitcount). Even all the way up is at most going to cost you a couple of bits of data per weapon fire. So if you are uncertain about this - leave it high at like 15 or 31.

(Various Cast/Overlap specific values) - Depending on which Cast Type is selected, various fields will appear here. These will be the values used by the rayCast, capsuleCast, overlapSphere, etc - such as distance, offset, radius.

Send As Mask - If you expect your tests to hit multiple other NST objects often, it may be more efficient to send the hits using a hitmask. Using a mask has a fixed size bitmask. How big this mask will be is determined by the global Header Settings value for BitsForNstId - specifically the resulting Max NST Objects is the number of bits used. This is pretty advanced stuff, so I just recommend leaving the default values and not exploding your head trying to micro-optimize this stuff.

CastResults

The callback interfaces used by NSTRewindEngine to notify components of outgoing or incoming cast events pass a reference to a class called CastResults. This class contains the CastDefinition Id that identifies which cast definition on the NST object was used for this cast, as well resulting hits (if the NSTCastDefinition settings indicate that it should be included).

CastResults returns two lists.

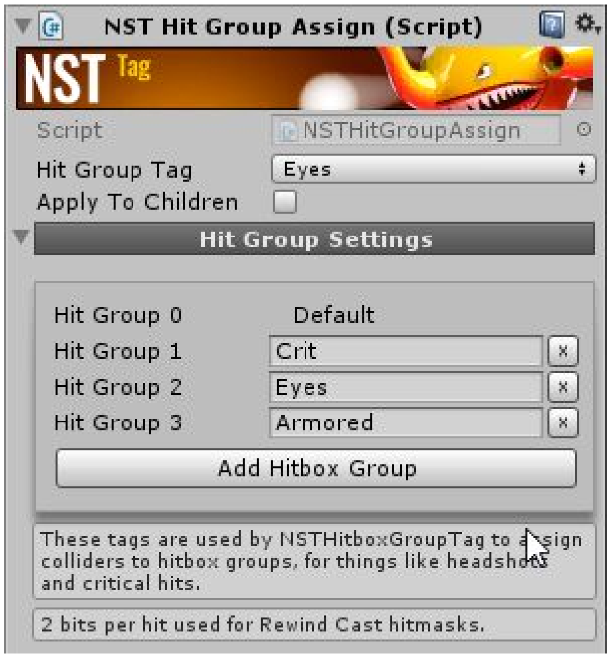
All NST objects that were hit by this cast:
`public List<NetworkSyncTransform>` **`nsts;`**

```
Corresponding hitmasks for each NST, indicating which hit groups were touched:
public List<int> hitGroupMasks;
```

Hit Groups

You can designate various colliders on your NST object to hit groups with the **NST Hit Group Assign** component. This allows you to detect things like crits and headshots.

Put this component on any child objects of an NST networked prefab to designate the collider(s) on that child to a group, or to designate any colliders on children of that object to a group.



NST Hit Group Assign (Script)

Script: NSTHitGroupAssign
Hit Group Tag: Eyes
Apply To Children: ☐

Hit Group Settings

Hit Group 0: Default
Hit Group 1: Crit
Hit Group 2: Eyes
Hit Group 3: Armored

Add Hitbox Group

These tags are used by NSTHitboxGroupTag to assign colliders to hitbox groups, for things like headshots and critical hits.

2 bits per hit used for Rewind Cast hitmasks.

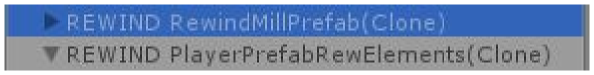
*Note: The **Hit Group Settings** scriptable object is appended to the bottom of the component to make it easy to change the global groups without having to go dig for the settings. Changes to Hit Group Settings in one place will change it everywhere, as there is only on Hit Group Settings singleton in a project.*

CastResults contains a list of masks indicating which of these hit groups were touched by the cast, one for each NST returned as a hit. So for each:
castResults.nsts[i]

there is a corresponding hitmask of which hit groups were impacted in
castResults.hitGroupMasks[i]

Rewind Ghosts

When you hit play on a scene that makes use of the rewind Add-on, you may notice inactive objects with the prefix “REWIND”...

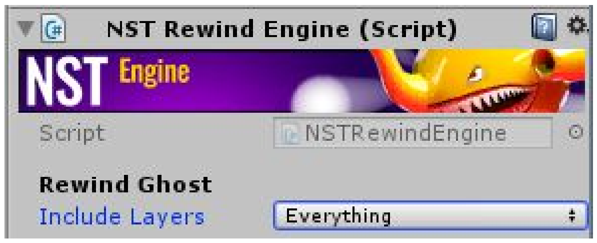


▶ REWIND RewindMillPrefab(Clone)
▼ REWIND PlayerPrefabRewElements(Clone)

These are the Rewind Ghosts of networked NST objects. They are very lightweight simplified skeletons of NST objects used specifically for rewinding. Rather than rewinding the actual game objects (which runs the risk of creating conflicts with user code, physics and future features - I have chosen to create dedicated ghost objects for all rewinding (and possibly for Server Authority over movement code in the future).

Ghost Colliders

The main function of Rewind Ghosts is for testing raycast and overlap casts, which uses colliders. These ghosts replicate their colliders from the source networked object. By default ALL colliders are cloned, but that may not be desired. For example you may have some colliders on objects for special uses, and they aren’t meant to be used as hitboxes. There is a drop list on the NSTRewindEngine component that lets you select which physics layers are meant to act as hitboxes - and any collider whose gameobject does not belong to one of those layers - will not be cloned to the ghost.

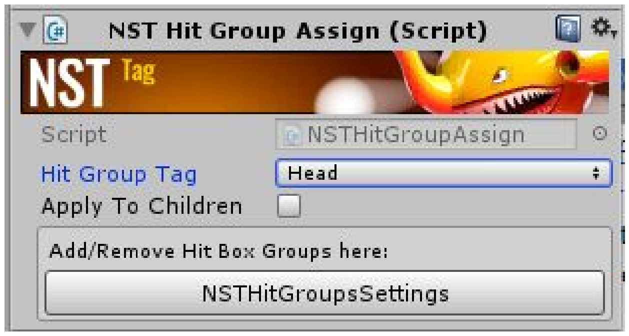


NST Rewind Engine (Script)

Script NSTRewindEngine

Rewind Ghost
Include Layers Everything

You may also make use of the NSTHitGroupAssign component to specify which hit group colliders on a gameobject should belong to. For example by placing this on the head object(s) of a player, you can indicate that that hits on those colliders should be flagged as headshots.



NST Hit Group Assign (Script)

Script NSTHitGroupAssign
Hit Group Tag Head
Apply To Children ☐

Add/Remove Hit Box Groups here:
NSTHitGroupsSettings

GhostEngine

The GhostEngine is a static class that keeps track of all NSTGhost objects and Ghost related settings in a project.

```
public static List<NSTGhost> ghosts;
```

You can enumerate all ghosts with a for or foreach loop using:
`GhostEngine.ghosts[index]`

A reference to the NSTGhost associated with a NST can be referenced with:

On older versions of NST the reference is stored in the NSTRewindEngine component that is automatically added to the root of the object:
`nstRewindEngine.ghostGO.GetComponent<NSTGhost>()`

On newest version of NST it now keeps that reference by the NSTGhost component rather than the ghost GameObject and can be referenced with:
`nstRewindEngine.ghost`
or
`(nst.iRewindEngine as NSTRewindEngine).ghost`

```
public static int rewindLayer;
public static int rewindMask;
```

If you want to run your own tests against the Ghosts, you access their physics layer number and/or mask with the static:

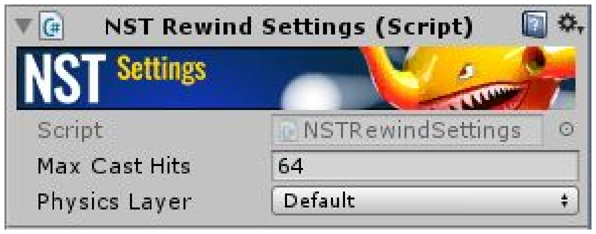
`GhostEngine.rewindLayer`
`GhostEngine.rewindMask`

(Note: these are public fields and not properties, so do be careful not to change the values in your code.)

This is the physics layer all ghosts have their colliders assigned to, so this is the layer you will want to use to mask your collision, raycasts and overlap checks.

NST Rewind Settings

Set some global settings here.



NST Rewind Settings (Script)

Script: NSTRewindSettings

Max Cast Hits: 64

Physics Layer: Default

Max Cast Hits
Sets the size of the non-alloc arrays for raycast/overlap operations in rewind. Make sure this value is greater than the most hits you would expect from any raycast, including ALL colliders on the included cast layers, not just NST objects.

Physics Layer
This is the physics layer that will be reserved for rewind hit tests. No scene objects can be using this layer. Leave as 'default' and it will automatically find the first unused (unnamed) physics layer. There is a risk here that some other Asset Store assets might be doing the same thing and will try to make use of the same layer - in that case you will want to actually make a dedicated physics layer for this and set it yourself.

NSTRewindEngine

This is the core rewind engine component that automatically is added to any NST objects when the Rewind Add-on exists in a Unity project. It creates the ghost objects at startup, and contains all of the basic methods used by rewind.

The most core method in the NSTRewindEngine is deceptively simple.

```
public HistoryFrame Rewind(float t, bool applyToGhost = true)
```

> **HistoryFrame** is a placeholder class that contains the state of the root object and any NSTTransformElement children being synced, at the time specified. Currently it just holds the root, and is a stub for future use. You can ignore this currently.
>
> **t** is the amount of time to rewind. Passing a value of .1 would result in rewinding 100ms. Passing a value of 1 would rewind 1 second, and so on.
>
> **applyToGhost** indicates of the ghost objects (see above in their own section) will have their transforms updated to the returned rewind values. I strongly recommend using the ghosts, as they are created and synced here for you to save you a LOT of work. This is optional, because there are other uses for rewind which I have not provided same code for, such as creating replays or actually rewinding time in-game, and and not just using Rewind for server confirmation of hit client claims.

This is deceptively simple, because getting the amount of time to rewind is actually NOT simple.

There is a method to automatically come up with the time value, just by supplying an NST as an argument.

```
public HistoryFrame RewindWithNst(
   NetworkSyncTransform playerNst,
   bool applyToGhost = true,
   bool includeServerBuffer = true,
   bool includeServerInterpolation = true
)
```

There is a dense little block of code inside of this implementation that comes up with a best guess at a rewind time value by summing up the Round-Trip Time between the Server and the Client(Player) that owns the supplied playerNst with the frame buffer and interpolation times

> **HistoryFrame** is a placeholder class that contains the state of the root object and any NSTTransformElement children being synced, at the time specified. Currently it just holds the root, and is a stub for future use. You can ignore this currently.
>
> **playerNst** is the instance of the networked object (this is all server only) that belongs to the player we are trying to rewind for. In most games this would be the player object for the

‘shooter’, as we need to rewind things on the server to get a best guess at how things looked on that players machine at the time of the event we are testing.

applyToGhost *same as above*

includeServerBuffer indicates if the rewind should factor in the frame buffer. This should be true if the event (such as weapon fire) is applied when the frame containing the event comes off of the frame buffer and is applied to the NST object. It is possible to process incoming updates on the server for things like weapon fire immediately when they arrive - before the frame is even added to the frame buffer, and if we are testing them immediately rather than when they come off the frame buffer - our rewind test should not factor in the size of the frame buffer. This is pretty advanced tweaking, so If this is confusing just stick with the default true value.

includeServerInterpolation indicates if we are factoring in the latency caused by interpolation (since interpolated objects are lerping toward the current position). This is used to account for the difference between having events like weapon fire apply normally at the end of interpolation (keeping the event in sync with the position/rotation), OR if we are processing our event at the start of interpolation rather than at the end. This is pretty advanced tweaking, so if this is confusing just stick with the default true value.

After running the Rewind methods, you can do your collision, raycast, or overlap tests against the associated NSTGhost of the NSTRewindEngine that was rewound.