NW3 SDK

Nali Weapons 3 Final

1 – Introduction

The *NWCoreVIII.u* package of this mod has several main classes and features which can be used in other mods, hence being used as some sort of SDK.

So far, the first version was already used in a few mods, and were used by only looking at the package classes code itself, and hence this document has the objective to make that job easier by already explaining the main classes.

2 – Structure

Here are explained all the key classes in a very generic way to give a glimpse of what kind of features you may expect to use:

Class	Description
NWUtils	Main utilities class, with generic and global static functions used for a variety of processes: math, canvas, string manipulation, parsing, water effects, gameplay checks and others.
NWInfo	Class which holds most global configurable properties (detail, gameplay, etc...).
NaliWActor	Main abstract actor class, created for organization and flexibility purposes, as it doesn't offer any kind of specific functionality.
NaliWEffects	Main abstract effects class, created mostly for organization and flexibility purposes, and its dynamic lighting is affected depending on the detail settings.
NaliWPawn	Main abstract pawn class, with the possibility to setting up a team, a master (another pawn this one may follow, protect or other) and if it's a machine (for advanced purposes, such as the *H-Missile* and *T-Missile* of the *MultiMissile Launcher* being able to tell if this pawn is mechanical or not). If the master is a player and he reconnects, this pawn will still be able to recognize the player as its master again (if the master keeps its player name intact in the process).
NaliWeapons	Main abstract weapon class. This is the class where all the weapons extend from, and has a multitude of features make much easier the creation of any kind of weapon: - Modifiers (firerate, damage, etc...) - Optimized view offset for wide screens and hidden views (ability to render text when the weapon is hidden to inform about the current mode if needed) - Complete separation between Fire and AltFire flows (on projectiles, offsets, etc...) - Fully working ambient sound when firing - Ability to define the max hitscan range, water effects, custom crosshair and different meshes for left and right handed versions - Ability to add hand models to the weapons without affecting the model itself (resulting also in the ability to customize the hands themselves) - Ability to change the view offset during the fire animation flow to a more natural behavior - Overlays and glows on the weapon - *Zero Ping* - Custom functions to easily add more functionality (such as a separate render function to render anything on the canvas without having to replace or cast *RenderOverlays*)

Class	Description
KeyedNaliWeapon	Main abstract keyed weapon class. This class extends from *NaliWeapons*, and therefore inherits all its features, and a new one is added: - The ability to have type numbers using the NumPad or the other number keys. This class is used for the Megaton and Megaton Decoder.
NaliProjectile	Main abstract projectile class. This is the class where all projectiles extend from and has a multitude of features which make much easier the creation of any kind of projectile: - Ability to spawn trails (up to 2) and smoke (the latter can depend on the zone) - Ability to define the water hit effect - Ability to define acceleration, if can hit the instigator, if it only damages as direct hit, blast noise, directional decals and damage radius - Is affected by the weapon modifiers it was fired from - Internal easy-to-use multiple timer functions (up to 3) - Most of the online complexity ins and outs are already automatically sorted depending on your projectile settings (replication)
NaliDynColorProjectile	Main abstract colored projectile class. This class extends from *NaliProjectile*, and therefore inherits all its features, and a new one is added: - The ability to change color dynamically over time. This feature requires 3 color versions of the skins/textures: red, green and blue.
NaliNuclearProjectile	Main abstract nuclear projectile class. This class extends from *NaliProjectile*, and therefore inherits all its features, and a new one is added: - Automatic detection of this projectile as a nuclear threat.
NaliPickups	Main abstract pickup class. This is the class where all the special pickups extend from and has a multitude of features which make much easier the creation of any kind of pickup: - Ability to directly define the pickup type (health, armor, damage, invisibility or custom scripted) - Ability to set its time, charge and cumulative max charge - As an armor, it has the ability to define if it should destroy or consume other armors, or just complement them - Overlays and glows on the pickup - Player and weapon effects while using the pickup
NaliAmmo	Main abstract ammo class. This is the class where all the weapons ammo extend from and has some features which make much easier the creation of any kind of ammo: - Ability to define as super ammo and deny replacement from a mutator - Custom animation when spawning - Custom states of animation (opened or closed)
NaliFullMeshFX	Main abstract mesh class which extends from *NaliWEffects*. This class allows one to spawn mesh based effects that need the mesh fully visible independently of the player view rotation, in other words, this is a specialized class to turn around the mesh native limitation that leads it to disappear in certain angles.
NaliTrail	Main abstract trail class for projectile trail effects, with adjustable pre-pivot.
NWCarcassFX	Main abstract carcass effect class which allows the spawn of effects in gore parts, such as flames.
NWCoronaFX	Main abstract dynamic corona and lens flare class, which can be used as an effect or a projectile trail.
NWClientAuto	Class responsible to setup client keybinds automatically and manage the renderer z-buffering.
NuclearExplosions	Main abstract class of a nuclear explosion, with an extremely optimized large radius damage algorithm.
NWNukeShockFX	Main abstract class of a nuclear explosion effect, although it's largely used for minor non-nuclear effects as well, such as shaking, modifying the player's FOV dynamically, ambient sound of shock approximation, impact sound, flash and spawn custom client side effects on, after or before the hit.

Class	Description
NWDecal	Main abstract decal class, which allows specific configurations such as the longevity of the decal (while the player sees, fixed lifespan or permanent).
NWDecalGen	Main abstract decal generation class, which is should be used in situations where a notification is needed for the client to know that he has to spawn a decal somewhere.
NWWallFX	Main abstract debris generator class, which spawns debris with a texture matching the BSP surface and flat polys to provide the illusion that BSP itself is being torn apart.
NWWallFrag	Main abstract debris fragment class.
NWReplacer	Main abstract inventory replacement class, which can be used to create normal replacement mutators, arenas or others beyond that. It's main features are: - Fully configurable, from head to toe - Compatible with most weapon mutators (the non compatible ones have wacky coding 99.9% of the times), despite its radical different approach on replacement methods - Support for multiple replacements on the same item - Plenty of different ways to process the replacement by several items (fixed, random, sequential, locker) - Can be subclassed at will to create multiple replacers, and they can be stacked with other replacers (up to 64 entries per single replacement) - Conditional replacement through filters - Default weapons and items on respawn support - Set of properties on replacement - Includes optional fix for scripted pawns in Monster Hunt games (*True* by default) - Can replace in all possible situations: LMS gametypes, “loaded” cheat commands, etc... - Mod independent (so you can load and replace items from other mods without depending on them) - Customizable chargers (or item pads, place holders, whatever you wish to call them) - Customizable lockers - Keeps the weapon priorities list as intact as possible, while setting it up to provide the best automatic selection of weapons at any given time - Automatic precaching of all items to avoid freezing during the game for items not yet spawned into the map
NWHandsInfo	Main class responsible for managing the weapon hands rendering skins.
NWMutator	Main class to build plugins or extensions.
NWProfile	Main class to store, load and save profiles, in order to load several predefined settings at once.
NWKillMsgManager	Main class to manage and fix the kill messages.
NWMainModMenuInfo	Main class to store all the properties concerning an entry in the *Mod* menu, such as: - Name, title and description - Max and min size, either by pixels or percentage of space on screen - Fully supported scroll
NWMenuPageInfo	Main class to setup and manage a complete *Mod* menu window very easily, by having: - Straightforward load and save functions - All the elements are setup from the default properties (so no more mental pixel calculations and whatnot, it's just add a new entry, write what element you want, done) - Help tips support - New custom built-in elements such as color control and profile control - Inputs size adjustable to the number of characters - Ability to add more levels of menus through “Advanced buttons” - High reliability on *.int* files, providing max flexibility to either add new entries, new tabs to an existing entry or even new inputs/elements to an existing tab dynamically, without having to extend the mod itself directly - Very easy to extend for most custom purposes without any need to rewrite code

Class	Description
NaliZPEffects	Main abstract class for *Zero Ping* effects. All the effects meant to be used from a *Zero Ping* weapon should be created by extending this class.
NWZPDecalGen	Main abstract class for *Zero Ping* decal generation. All the generation of decals meant to be used from a *Zero Ping* weapon should be created by extending this class.
NWZPUT_SmokePuff	Main class for *Zero Ping* smoke effects. All the generation of smoke effects meant to be used from a *Zero Ping* weapon should be created by extending this class.
NWGoreSet	Main abstract class for everything concerning the new gore system.
NWGoreCBoard	Main abstract class to setup and manage the blood color in the gore parts.
NWBloodDecal	Main abstract blood decal class.
NWBloodyMess	Class responsible to manage all the gore system.
NWBodyPiece	Main abstract class for gore parts.
NWCordNodePiece	Main abstract class for gore parts which are composed by wire-like physics, like guts, veins, intestines, etc.
NWSkillKillsManager	Class responsible to manage the new skill kills system.

3 – Nali Weapon Overview

Every single weapon on this pack is subclassed from the class *NaliWeapons*, and although they are indeed more complex than the average weapon in the game, they have a very straightforward organization within them to facilitate greatly the build of any weapon with all the features you can see in-game.

The making of a Nali Weapon is pretty straightforward: everything starts with the 3D modeling of the weapon and hand(s) which hold the weapon, they get animated and textured, then exported to separate files so the mod can deal with the weapon model and the hands model separately.

The import process and setup is similar to any other custom weapon, and the real differences rely on the features and additional weapon settings themselves:

CustomCross is setup with the custom crosshair icon of the weapon, while ***CrossHairScale*** defines its rendering scale.

PlayerViewMeshLeft and ***PlayerViewMeshRight*** are setup with the left and right handed version of the weapon respectively (this way you don't have to code it yourself like in every other weapon).

HandPartMeshL and ***HandPartMeshR*** are setup with the respective left and right handed version of the weapon hands respectively. As you notice, both are array types with 2 possible entries each, so you can have 2 separate hands holding the weapon instead of just one, although most of the time you will only use the first entry (index 0).

BobDamping should be set to a value bigger than *1.0* if you follow the way other Nali Weapons first person view models are imported relative their scale.

FireOffset and ***AltFireOffset*** are setup with the respective fire and alt-fire offset locations. *AltFireOffset* is the new setting here, and was created exactly to distinguish the offset between fire and alt-fire (something the standard weapons do not do, which is wrong).

AnimMaxFrame has always to have the value corresponding to the following calculation:
([number of frames in Select animation] – 1) / [number of frames in Select animation]
which means that if your *Select* animation has 6 frames total, the number to put here is
(6 – 1) / 6 *= (5) / 6 = 0.8333333333333333 ≃* ***0.83333***
You must be wondering why this calculation isn't made automatically, right? Well, that's because the engine doesn't have any function or property to check the number of imported frames of any mesh, therefore this has to be calculated beforehand and set by hand.

The ***AnimMaxFrameFire*** and ***AnimMaxFrameAltFire*** are calculated in the same fashion, but relative *Fire* and *AltFire* animations respectively.

RenderOffsetSelect is the offset the weapon will move to during the *Select* animation. In some weapons this may make the weapon *Select* animation much more fluid and natural, and is the reason why the property *AnimMaxFrame* above has to be setup.

The ***RenderOffsetFire*** and ***RenderOffsetAltFire*** work in the same exact fashion, but relative *Fire* and *AltFire* animations respectively.

FireOffsetZAdjustHidden and ***AltFireOffsetZAdjustHidden*** have the same exact functionality as *FireOffset* and *AltFireOffset*, however these are only applied when the weapon rendering mode is set to hidden, as in some weapons it may be desirable to have a different offset of fire when their rendering is hidden.

FirstPersonGlowFX and ***PickupGlowFX*** are the glowing settings for the weapon, relative first person and pickup views respectively.

For the glows, you must prepare and import custom meshes with the same animation as the weapon itself (similar to the hands models), and textures. What's truly going to be used from these meshes are their vertex locations, and they define where each glow will be at, therefore they must be as low-poly as possible, and count every vertex (so if you want to have 4 glows, your model should have only 4 vertexes total at the places you want the glows to spawn).

As for their internal properties, they are as follows:

bLit – Render the glow unlit;

bRandFrame – Make the glow/particles model bRandomFrame=True;

GlowModel and *GlowModelLeft* – Right and left versions of the vertex mesh respectively;

GlowStyle – Render style (normal, translucent, etc);

GlowAmount – Render scale glow;

GlowTexScale – Render glow texture scale;

PulsingGlow – Pulsing glow amount;

GlowTex1 to *GlowTex8* – Textures making the glow (the *2* to *8* ones are only used if *bRandFrame* is set to *True*);

bRenderOnTop – Render the glow on top of everything else (first person view only);

GlowSetClass – Custom *NWeaponOverFX* class (may be desirable to have a custom glow class sometimes for a more customized behavior).

FirstPersonOverFX and ***PickupOverFX*** are similar to the *FirstPersonGlowFX* and *PickupGlowFX* settings respectively (see above), with the main difference that instead of glow textures they are mesh overlays, therefore a custom mesh is not required, but if desired, the custom mesh should contain all the same polys that the main weapon models have, but stripped down to only the ones which are really needed in the overlay (the ones which have a different and visible texture applied in the overlay), for performance purposes.

As for their internal properties, they are pretty much the same as the *FirstPersonGlowFX* and *PickupGlowFX* ones, with a few differences:

bEnviromentMode – Enable *bEnvironmentMap* in the overlay model;

OverTex0 to *OverTex7* – Overlay *MultiSkins*;

OverMainTex – Overlay *Texture*;

bCustomMesh – Has custom overlay mesh;

CustomMesh and *CustomMeshLeft* – Overlay right and left handed mesh versions respectively;

OverlModelClass – Custom *NWeaponOverFX* class (may be desirable to have a custom overlay class sometimes for a more customized behavior).

bForceHands should be set if you want the hands to be always visible on your weapon, independently of the player settings (since some weapons wouldn't make any sense visually without them, like the *Megaton* and *The Executioner* for example).

There are far more settings, but these are the ones you should worry about when planning a new Nali Weapon.

For a more complete view and explanation of settings and functions, check the ***Classes Basic Reference*** section below.

4 – Effects

One the main characteristics of this pack is the multitude of different visual effects cast when using the weapons.
Although these effects may look complex at first, many of them are actually pretty simple to perform as most of the needed complexity is already covered through code, and therefore all you have to is to import custom assets and setup as few settings in a few classes.
The four most noticeable custom effects are: dynamic coronas, BSP debris, shakes and flames on gore parts and they are described as follows:

Dynamic coronas and lens flares are subclasses of the ***NWCoronaFX*** class, where you can simply import the textures and then setup its behavior (lifespan, translucency, scaling, flickering, etc...), and thus producing small discrete lights or even huge flares (like the ones seen in the initial phase of a nuclear explosion). This class can also be attached and last as long as the actor its attached too lasts as well.

BSP debris are always subclasses of both the ***NWWallFX*** and ***NWWallFrag*** classes, being the former the debris generator, and the latter the generated fragment. In the *NWWallFX* you define the *NWWallFrag* class to be spawned, the amount and its speed behavior, while in the *NWWallFrag* class you define its scaling, lifespan, some speed properties and other small things.

The shakes, flashes, dynamic tween of the player's FOV and timed sound effects are made from subclasses of the ***NWNukeShockFX*** class, which is the most featured effects class of all the pack. This class is used heavily for the additional nuclear explosion effects (shockwave hit, ground shake,initial flash and others), and as the class name suggests, this class was created mostly for such nuclear effects. However it can and is used in a multitude of other effects, whenever is the need for a flash or ground shake.

The flames cast in the gore parts of a dying player from some weapons are done by using subclasses of the ***NWCarcassFX*** class, where the affecting radius, kind of flame and gore class to cast it to can be defined. The usage of this effect should not be abused for performance reasons.

5 – Items Replacement and Arenas

One of the strongest and most complex features of the whole pack is the replacement mutator, as it supports a vast set of features which cannot be found in any other mod of the same type.
Although the ***NWReplacer*** class (from which the main mutator and all the arenas extend from) it's internally complex, it offers simplicity for the end developer or admin by being able to have a full featured replacement for his mod.

By subclassing this class, you can create your own arena or replacement mod very easily without having to worry about a single thing, or even come up with your own gametypes (like DM-LMS, by spawning with all weapons, armor and health just like in LMS, but in a simple DM game).
Adding to this, as a developer its very very easy to modify its behavior in any way you like (as you can see in the main mutator *NWMainReplacer* class).

Also it's worth to mention that this replacement mutator is radically different in how it replaces items, fixing all the problems the standard way has.
So instead of manually writing your own replacement mutator, you can simply subclass this one and just toy with its default configurable settings to do everything you would possibly want.

For further details take a look into the *INI_NWConfig.pdf* file, in the *[NWCoreVIII.NWMainReplacer]* section, where the explanation on each setting is explained.

6 – Zero Ping

All the hitscan weapons in this pack support a custom version of *Zero Ping* in order to provide a much superior gameplay experience to players with high ping, while trying to be as secure as possible.

However, unlike other *Zero Ping* systems, the security does not rely on obfuscation and therefore the source code is open and the process of building a *ZP* weapon is very straightforward, structured and simple, as all you have to worry to make your weapon *ZP-able* are the following:

- The weapon must have the following properties set:

```
isZPWeapon=True
ZPMaxFirerate=<the weapon normal firerate>
ZPMaxFirerateAlt=<the weapon alt firerate>
bFireHitScan=<True if the normal fire is hitscan>
bAltFireHitScan=<True if the alt fire is hitscan>
AccuracyPattern(0..3)=<accuracy variation value for normal fire>
AccuracyPatternAlt(0..3)=<accuracy variation value for alt fire>
```

- In the weapon code itself, in the *ProcessTraceHit* function, always make it *simulated*, and every effect spawned from there must have it's owner set to *ZPOwner*, for example:

```
Spawn(class'NWZPUT_SmokePuff', ZPOwner,, HitLocation+HitNormal*9);
```

- All the effects spawned from there must be subclasses of any of the following classes: *NaliZPEffects* (for any effects), *NWZPDecalGen* (to generate decals) or *NWZPUT_SmokePuff* (for a generic smoke puff effect).

- All sounds, damage and other server-side only processes must have a `if(Role==ROLE_Authority)` check before.

And done, everything else you can do as if you were doing any other custom weapon without a worry in the world.

Yes, in 4 simple steps, using the right classes and settings, you can easily make your hitscan weapon a *ZP* weapon, benefiting of all the experience and security of this *ZP* system.

A good and simple *ZP* weapon example to check is the *WRE* class from the *NWWREVIII.*u package.

7 – Menus

As you surely know, this pack has tons of settings to tweak the mod to your likings and needs. However, many of these settings have to be in menus as not everyone goes through the hassle to do it directly in a configuration file, but due to the ancient and frustrating system of menus of Unreal Engine (*UWindows*), it's a real pain to do any kind of menu, even the simplest ones.

For that reason a system was developed on top of the existing one with the sole purpose of making menus very easily without all the hassle the old system requires, besides adding a few useful features to it.

This menu system consists in a very straightforward structure of setup and navigation:
- Mod menu entry;
- The entry opens a window with a set of tabs;
- Each tab opens a page on the window itself with all the settings (elements);
- Each setting triggers a *load* when opened and a *change* when modified through user input.

And adding to this:
- Each element may have a help tip when hovering on a setting;
- Each element may have an additional “advanced button” to open another window with another set of tabs and therefore another set of settings (creating depth levels).
- Everything relies heavily on *.int* files in order to expand existing tabs and pages, giving max flexibility in adding and removing any menus as needed without the need of a recompile or the creation of dependencies.

To setup a simple menu, you must first create a subclass of ***NWMainModMenuInfo***, which is the container and frame of the whole menu, where all the tabs and settings are going to be into.

It has some settings you can play with:

bUniqueMainMenu – If only 1 menu of this class can be visible at a time;

bSizableMainMenuW – If can be resized along its width;

bSizableMainMenuH – If can be resized along its height;

bCenterMainMenu – If its initial position is at the center of the screen;

MainMenuCaption – Text to appear as an entry of another menu (like the Mod menu);

MainMenuHelp – Help text to appear when hovering on the entry of this menu;

MainMenuTitle – Text to appear as the title of this menu;

bMainMenuPosPercentageX – If the settings on the X position of the menu should be read as percentage;

bMainMenuPosPercentageY – If the settings on the Y position of the menu should be read as percentage;

bMainMenuSizePercentageW – If the settings on the width of the menu should be read as percentage;

bMainMenuSizePercentageH – If the settings on the height of the menu should be read as percentage;

MainMenuPosX – Menu X position on the screen;

MainMenuPosY – Menu Y position on the screen;

MainMenuPosW – Menu width;

MainMenuPosH – Menu height;

MainMenuMinSizeW – Menu min allowed width;

MainMenuMinSizeH – Menu min allowed height;

After defining your main menu structure, you must create your tabs (menu pages within the menu window) as subclasses of ***NWMenuPageInfo***. After the new class of this is created, you need to setup the *PageTitle*, which is the text which is going to appear in the tab on the top of this menu, and also set its *ModMenuInfoClass* to the *NWMainModMenuInfo* class reference you created above.

From there, you need to setup the *SettingsList* list in the default properties in order to add settings/elements to your menu (inputs, combos, checkboxes, etc), as:

Description – Caption of the setting;

HelpTip – Help tip text (the help text that appears when you hover the mouse over a setting) ;

Type – The type of the setting, which can be any of the following: *ST_Checkbox*, *ST_Input*, *ST_IntegerInput*, *ST_FloatInput*, *ST_Slider*, *ST_Combo*, *ST_Color*, *ST_Profile* or *ST_Label* (all of them self-explanatory);

MaxChars – Max number of characters in case of a text or numeric input (it also auto resizes depending on the number of allowed characters);

BottomMargin – Number of pixels of space before the next setting (good to separate sections of settings within the same tab);

MinSliderVal – Min slider value (only used when *Type=ST_Slider)*;

MaxSliderVal – Max slider value (only used when *Type=ST_Slider)*;

SliderStep – Slider step (only used when *Type=ST_Slider)*;

SliderSize – Slider size (only used when *Type=ST_Slider)*;

SliderTrackSize – Slider track size (only used when *Type=ST_Slider)*;

SliderTrackSize – Slider track size (only used when *Type=ST_Slider)*;

ColorTex – Greyscale icon/texture to be used when *Type=ST_Color;*

ProfileClass – Profile class to affect when *Type=ST_Profile*;

hasAdvanced – Has advanced button to open another menu;

AdvancedText – Advanced button text;

AdvancedMenuInfoClass – Advanced menu class to open when the advanced button is pressed.

Example:

```
SettingsList(0)=(Description="My setting")
SettingsList(0)=(HelpTip="This is my example setting")
SettingsList(0)=(Type=ST_IntegerInput,MaxChars=6)
```

And with that you just finished the visual structure of your menu, and as you may have noticed, no pain involved.

However, you need to load and save the settings themselves through custom code (hey, there are no miracles!), but that job is also a very light and easy job compared to doing it manually as all you have to do is to rely on functions.

So, getting back to the page class itself where all your settings are added (the *NWMenuPageInfo* subclass you just created above), you need to add some code to the functions which are called whenever there's intent to load or save.

For more details on these functions, check the ***Classes Basic Reference*** section below on these classes and some examples in the already existing menus in the pack (such as the *NWServerGenericSettingsMenuPageInfo* class).

After you setup the functions, in order for the game to load up your newly created menus into the Mod menu, you must create a *.int* file with the following structure:

The Mod menu entry:

```
Object=(Name=Your_Package.Your_NWMainModMenuInfo_Subclass,Class=Class,MetaClass=UMenu.UMenuModMenuItem)
```

The tabs:

```
Object=(Name=Your_Package.Your_NWMenuPageInfo_Subclass1,Class=Class,MetaClass=NWCoreVIII.NWMenuPageInfo)
Object=(Name=Your_Package.Your_NWMenuPageInfo_Subclass2,Class=Class,MetaClass=NWCoreVIII.NWMenuPageInfo)
...
```

8 – Profiles

Another feature of this pack is the ability to save a bulk of settings as a “profile”.
To create a profile class you need to extend from ***NWProfile*** and create data structures with all the settings you want to save and load, and these structures must be declared as arrays with exactly 8 elements (the max number of profiles allowed).

From there, *SelectedProfile* defines the default profile (from *0* to *7*) and *ProfileNames* is the list of the captions given to each respective profile to be shown in a menu control, and then you just need to use the *ProcessProfileChange* and *ProcessProfileSave* events to manage the whole thing:
ProcessProfileChange – Called whenever a profile is loaded or changed (if dynamically within a level, the *Lvl* argument is set to the current level);
ProcessProfileSave – Called whenever a profile is saved.

In both events, the argument *i* is set to the profile ID to load, change or save (from *0* to *7*).
A great example of a working profile can be seen in the class ***NWGameplayProfile***.

A profile can be connected to another through the usage of *.int* files (for example, all the weapons gameplay profiles are connected to the core gameplay profile, in order for them to receive the same events at the same time whenever the gameplay profile is loaded, changed or saved from the core profile itself).

To make this type of connection, the entry is as follows:
```
Object=(Name=MyPackage.ProfileToConnect,Class=Class,MetaClass=OtherPackage.MainProfileToConnectFrom)
```

Examples:
```
Object=(Name=NWWREVIII.WREGameplayProfile,Class=Class,MetaClass=NWCoreVIII.NWGameplayProfile)
Object=(Name=NWBoltRifleVIII.BoltRifleGameplayProfile,Class=Class,MetaClass=NWCoreVIII.NWGameplayProfile)
Object=(Name=NWGravitonVIII.GravitonGameplayProfile,Class=Class,MetaClass=NWCoreVIII.NWGameplayProfile)
```

9 – Gore

The new gore system introduced in the final version of this pack is as extensible and flexible as all the other features, however instead of affecting the gameplay, it mostly affects the graphical aspect of the pack as a whole when it comes to killing a player or another pawn in the game.

The key classes to check in case you want to see how the gore system was done are:

NWGoreSet - The parent class of all the gore. It does not do anything by itself, and serves mostly to group everything in the same tree of classes;

NWGoreCBoard - Has a set of properties and functions to define and modify the gore part depending on the blood type;

NWBloodDecal - Blood decal class;

NWBloodyMess - Self-loaded mutator which manages the entire gore system.

NWBodyPiece - Body parts composing the visual gore.

NWCordNodePiece - Special body part composed by segments and joints in such a way to create rope-like parts, like guts, intestines, anything that can stick and/or swing around.

10 – Classes Basic Reference

Here are listed basic references of all the important classes you may want to use in your any of your mods, and all of them can be found in the core package: *NWCoreVIII.u*.

Only the key functions and properties are mentioned for the sake of simplicity and a “straight to the point” approach, therefore most of the other properties and functions may not be mentioned, although the ones mentioned should give a good idea on how each class behaves, but if even so if you're still in doubt, you can always contact me and ask (see the ***Contact*** section of *NW3.pdf*).

For the properties which are marked in the code as *config* or *globalconfig*, please see the respective sections in the documentation on the configuration files (*INI_NWConfig.pdf*, *INI_NWeaponsCfg.pdf*, *INI_NWExtrasCfg.pdf* and *INI_NWNuclearCfg.pdf*).

Class: NWUtils	**Parents:** Actor > NaliWActor

Description:

This is the utilities class with all the generic functions used in other classes: math, canvas, string manipulation, gameplay and others.

All the functions of this class are *static*, and although the *simulated* modifier is not necessary in their declaration, the *simulated* keyword is meant to inform if the function is supposed to run on the client or not, therefore every *static* function without *simulated* shouldn't be called from the client, otherwise the return may not be what you're expecting.

Properties:

class ***ValidTextureClasses[8]***

- Defines all the classes to be considered as textures (since the engine is not smart enough to do itself automatically), with up to 8 entries.

class<Effects> ***WaterProjSplashClass[8]***

- Defines all the effect classes to be spawned as water zone splashes. The index defines the order of magnitude, hence *0* is the smallest and *7* is the largest splash.

class<Effects> ***SlimeProjSplashClass[8]***

- Defines all the effect classes to be spawned as slime zone splashes. The index defines the order of magnitude, hence *0* is the smallest and *7* is the largest splash.

class<Effects> ***LavaProjSplashClass[8]***

- Defines all the effect classes to be spawned as lava zone splashes. The index defines the order of magnitude, hence *0* is the smallest and *7* is the largest splash.

class<Effects> ***WaterBallisticSplashClass[4]***

- Defines all the effect classes to be spawned as water zone vertical ballistic splashes. The index defines the order of magnitude, hence *0* is the smallest and *4* is the largest vertical ballistic splash.

class<Effects> ***SlimeBallisticSplashClass[4]***

- Defines all the effect classes to be spawned as slime zone vertical ballistic splashes. The index defines the order of magnitude, hence *0* is the smallest and *4* is the largest vertical ballistic splash.

class<Effects> ***LavaBallisticSplashClass[4]***

- Defines all the effect classes to be spawned as lava zone vertical ballistic splashes. The index defines the order of magnitude, hence *0* is the smallest and *4* is the largest vertical ballistic splash.

class<NWWaterSplashRing> ***WaterProjRingClass***

- Defines the water splash ring/wave effect class.

class<NWWaterSplashRing> ***SlimeProjRingClass***

- Defines the slime splash ring/wave effect class.

class<NWWaterSplashRing> ***LavaProjRingClass***

- Defines the lava splash ring/wave effect class.

Functions:

simulated static function float aSin(*float f****)***

- Returns the arc sine of an angle ***f*** (in radians).

simulated static function float aCos(*float f****)***

- Returns the arc cosine of an angle ***f*** (in radians).

simulated static function processActorDetail(*LevelInfo Lvl, Actor A, bool bLight, float distDetail****)***

- Processes the actor **A** dynamic light and LOD depending on the ***bLight*** and ***distDetail*** arguments:
  bLight: If *True*, the actor will have dynamic light turned on, and not if otherwise;
  distDetail: Detail amount.

Lvl is always the current *Level*, and has to be passed as argument as static functions do not have direct access to their own instance members (such as *Level*).

static function bool isMonsterGame(*LevelInfo Lvl****)***

- Check if the current game is a monster game (like Monster Hunt for example).

Lvl is always the current *Level*, and has to be passed as argument as static functions do not have direct access to their own instance members (such as *Level*).

static function bool isMonsterGame(*LevelInfo Lvl****)***

- Check if the current game is a monster game (like Monster Hunt for example).

Lvl is always the current *Level*, and has to be passed as argument as static functions do not have direct access to their own instance members (such as *Level*).

static function bool isFriend(*Pawn P, LevelInfo Lvl, optional Pawn Instig, optional byte team, optional bool bNoHurtTeam, optional bool bNoHurtInstig, optional string ownerName, optional Actor src****)***

- Check if pawn ***P*** is a friend, depending on:
  Instig: The instigator of the call;
  team: The team to which the call belongs;
  bNoHurt: If can hurt teammates or not;
  bNoHurtInstig: If can hurt the instigator;
  ownerName: Instigator name;
  src: Actor source of the call.

Lvl is always the current *Level*, and has to be passed as argument as static functions do not have direct access to their own instance members (such as *Level*).

static function bool isSameHorde(*Pawn P, Pawn Instig, LevelInfo Lvl****)***

- Check if pawn ***P*** is of the same horde as pawn ***Instig*** (by “horde” is meant as a set of monsters belonging to the same group or team).

Lvl is always the current *Level*, and has to be passed as argument as static functions do not have direct access to their own instance members (such as *Level*).

simulated static function bool isTeamMember(*Pawn PSource, Actor A****)***

- Check if pawn ***PSource*** is of the same team as actor ***A***.

simulated static function byte getTeam(*Actor A****)***

- Get actor ***A*** team number.

simulated static function bool isValidTarget(*Actor A, optional bool ignoreStationaryPawn****)***

- Check if actor ***A*** is a valid enemy target.
ignoreStationaryPawn is passed as *True* if stationary pawns aren't supposed to be considered valid targets.

static function bool processFiredHealth(*int hAmount, Pawn P, Pawn ObjInstig, optional bool noHealth****)***

- Process pawn ***P*** health incrementation by ***hAmount*** points from a health modified weapon belonging to the pawn ***ObjInstig***.
Returns *True* if the pawn ***P*** is a valid pawn to increment health relative pawn ***ObjInstig***, and *False* if otherwise.
When ***noHealth*** is passed as *True*, pawn ***P*** won't receive any health even if it's valid to, and is therefore used when only the return value of this function is intended in its usage.

static function byte IdentifyTeam(*LevelInfo Lvl, Actor A, out ControlPoint CPTeam, out PlayerStart PSTeam, out FortStandard FSTeam, out byte bHaveFort****)***

- Get the team number based on the actor ***A*** location in a map in a team based gametype.
CPTeam is set as the closest ***ControlPoint*** in case the gametype is *Domination*.
PSTeam and ***FSTeam*** are set as the closest ***PlayerStart*** and ***FortStandard*** respectively in case the gametype is *Assault*.
bHaveFort is set to *1* in case ***FSTeam*** is closer than ***PSTeam***.
Lvl is always the current *Level*, and has to be passed as argument as static functions do not have direct access to their own instance members (such as *Level*).

static function byte GetCurrentTeam(*Actor A, LevelInfo Lvl, byte defaultColor, byte PTeam, optional ControlPoint CPTeam, optional PlayerStart PSTeam, optional FortStandard FSTeam, optional byte bHaveFort****)***

- Get the team number based on the actor ***A*** location in a map and in the current team ***Pteam*** and in the following data: passed *ControlPoint* ***CPTeam***, passed *PlayerStart* ***PSTeam***, *FortStandard* ***FSTeam*** and if ***FSTeam*** is prioritized over ***PSTeam*** by passing ***bHaveFort*** as *1*, in a team based gametype.
defaultColor is the desired default team color index if no specific team is returned.
Lvl is always the current *Level*, and has to be passed as argument as static functions do not have direct access to their own instance members (such as *Level*).

static function InitializeRes(*NRessurectFX NRes, Actor A, LevelInfo Lvl, bool bEnableTeamColor, byte defaultColor, byte Pteam, optional ControlPoint CPTeam, optional PlayerStart PSTeam, optional FortStandard FSTeam, optional byte bHaveFort****)***

- Initalize the color of the ressurection effect ***NRes*** based on the actor ***A*** location in a map and in the current team ***Pteam*** and in the following data: passed *ControlPoint* ***CPTeam***, passed *PlayerStart* ***PSTeam***, *FortStandard* ***FSTeam*** and if ***FSTeam*** is prioritized over ***PSTeam*** by passing ***bHaveFort*** as *1*, in a team based gametype.
bEnableTeamColor indicates if the color should be team based.
defaultColor is the desired default team color index if no specific team is detected or if ***bEnableTeamColor*** is *False*.
Lvl is always the current *Level*, and has to be passed as argument as static functions do not have direct access to their own instance members (such as *Level*).

static function bool isAllowedToKick(*Pawn Inst, Pawn P, LevelInfo Lvl, int Kickback, optional byte savedTeam, optional string ownerName, optional actor src****)***

- Check if pawn ***Inst*** is allowed to kickback pawn ***P***, based on the ***Kickback*** value, and optionally the team ***savedTeam***, instigator player name ***ownerName*** and the actor source ***src*** of the kickback.
Lvl is always the current *Level*, and has to be passed as argument as static functions do not have direct access to their own instance members (such as *Level*).

simulated static function bool isMonster(*Pawn P****)***

- Check if pawn ***P*** is a monster.

static function bool isDMGame(*GameInfo GI****)***

- Check if gametype ***GI*** is a death match.

simulated static function bool isValidBot(*Pawn P****)***

- Check if pawn ***P*** is a valid bot-like pawn.

static function bool isElegibleBotAI(*Pawn P, optional bool ignoreNovice, optional float minSkill, optional float maxSkill****)***

- Check if pawn ***P*** has enough intelligence to do some sort of AI operation based on:
  ignoreNovice: If *True*, ignore the fact that the bot may be a novice;
  minSkill: Min bot skill;
  maxSkill: Max bot skill.

simulated static function rotator rTurn(*rotator rHeading, rotator rTurnAngle****)***

- Get rotator result of a rotation of ***rHeader*** over ***rTurnAngle*** (all credits go to *UnrealWiki* for this particular function).

simulated static function vector HUDObjectToWorld(*Actor TargetOther, PlayerPawn PPOwner, float HX, float HY, float ScreenWidth, float ScreenHeight, float DistFromScreen****)***

- Get vector with the location of the actor ***TargetOther*** in the level from canvas coordinates, given the player ***PPOwner*** as the rendering target, ***HX*** and ***HY*** as the X and Y coordinates, ***ScreenWidth*** and ***ScreenHeight*** as the max width and height of the screen respectively (game resolution), and ***DistFromScreen*** as the distance from the screen the object should be rendered at.

simulated static function bool LocToCanvas(*out vector OutXY, vector Loc, vector ViewOrigin, rotator ViewRot, canvas Canvas, optional bool returnPrecision****)***

- Get as ***OutXY*** the X,Y coordinates in the canvas given a level location ***Loc***.
This function returns *True* if the render location has a valid X,Y coordinate at all (visible), and takes the following additional arguments:
  ViewOrigin: View origin location;
  ViewRot: View rotation;
  Canvas: The canvas to render at;
  returnPrecision: If *False*, it returns the ***OutXY*** parameters rounded to the nearest integer.

simulated static function bool ActorToCanvas(*out vector OutXY, actor CTarget, canvas Canvas, optional bool bConsiderCollisionHeight, optional bool returnPrecision****)***

- Get as ***OutXY*** the X,Y coordinates in the canvas given a level location ***Loc***.
This function returns *True* if the render location has a valid X,Y coordinate at all (visible), and takes the following additional arguments:
  Canvas: The canvas to render at.
  bConsiderCollisionHeight: Return ***OutXY*** relative the top location of the actor (actor location + collision height);
  returnPrecision: If *False*, it returns the ***OutXY*** parameters rounded to the nearest integer.

simulated static function byte getResolutionFontCoef(*canvas Canvas, optional byte maxCoef****)***

- Get the proper font size coefficient given the ***Canvas*** to render at.
The return varies between *0* and *5* (inclusive), as *0* the smallest font and *5* as the largest.
If ***maxCoef*** is passed with a value smaller than *5* and higher than *0*, the font size returned with be limited to that max value.

simulated static function int getFrameRateLevel(*LevelInfo Lvl, float Delta, optional bool returnExcessLevel****)***

- Get the framerate level based on the frame ***Delta*** time.
If *0*, it means that the framerate is OK.
If between *+1* and *+5*, then it means that the game has still room to process more things without loosing a noticeable framerate (being *+1* as “a bit of room” and *+5* as “lots of room”).
If between *-1* and *-5*, then it means that the game is loosing performance (being *-1* as “loosing a bit of performance” and *-5* “lag hell”).
returnExcessLevel is passed as *False* if any result value above *0* should be returned as *0*.
Lvl is always the current *Level*, and has to be passed as argument as static functions do not have direct access to their own instance members (such as *Level*).

simulated static function float getFrameRateBasedLODBias(*LevelInfo Lvl, float Delta, float curLODBias****)***

- Get the LOD value based on the frame rate, depending on the frame *Delta* time and the current ***LODBias*** as ***curLODBias***.
Lvl is always the current *Level*, and has to be passed as argument as static functions do not have direct access to their own instance members (such as *Level*).

simulated static function int getSign(*float n****)***

- Return *1* if ***n****>0*, *-1* if ***n****<0* or *0* if ***n****=0*.

simulated static function string getValueFromSettingsString(*coerce string varString, string strSettings****)***

- Get the value of the property ***varString*** in the list of settings ***strSettings*** which is formatted as *"property=value;"* (no spaces).

Example:

getValueFromSettingsString("property2", "property1=abc;property2=523;");
returns: "*523*"

simulated static function bool hasValueFromStringList(*string dataStr, int index, optional out string indexData****)***

- Check if the list of values ***dataStr*** has the value corresponding to the position ***index*** (starting in 0) which is formatted as *"value1,value2,value3"* (no spaces) and puts that value into ***indexData***.

Example:

hasValueFromStringList("553,stuff,someval,abc", 2, indexData)
returns: *True* and ***indexData*** assumes the value *"someval"*.

simulated static function bool hasPropertyFromStringList(*string dataStr, int index, optional out string indexProperty, optional out string indexData****)***

- Check if the list of values ***dataStr*** has the property corresponding to the position ***index*** (starting in 0) which is formatted as *"property1=value1;property2=value2;property3=value3;"* (no spaces) and puts the property name into ***indexProperty*** and the value of that property into ***indexData***.

Example:

hasPropertyFromStringList("prop1=500;prop2=ert;prop3=stu;", 1, indexProperty, indexData)
returns: *True* and ***indexProperty*** assumes the value *"prop2"* and ***indexData*** assumes the value *"ert"*.

simulated static function processStrSplit(*string splitStr, string srcStr, out string outStr1, out string outStr2****)***

- Splits the string ***srcStr*** into ***outStr1*** and ***outStr2*** by using the first occurrence of ***splitStr*** as the separator.

Example:

processStrSplit("!", "foo!bar", outStr1, outStr2)
result: *outStr1* assumes the value *"foo"* and ***outStr2*** assumes the value *"bar"*.

simulated static function string ReplaceStr(*coerce string Replace, coerce string With, coerce string Text, optional int maxReplaces****)***

- Replaces all occurrences of the string ***Replace*** by ***With*** in ***Text***.

If ***maxReplaces*** is passed with as a number greater than zero, the number of replaced occurrences is limited to this number.

simulated static function bool StrMatch(*string sA, string sB, optional bool bCaseSensitive****)***

- Check if string ***sA*** matches string ***sB***.

The ***sA*** argument supports 2 wildcards: * and *?*:

* - Match with any number of any characters or none at all;
? - Match with any 1 character.

If ***bCaseSensitive*** is passed as *True*, the match will be case-sensitive.

Examples:

StrMatch("stuff", "STUFF", False); : returns *True*
StrMatch("stuff", "STUFF", True); : returns *False*
StrMatch("stuff", "STAFF", False); : returns *False*
StrMatch("st?ff", "STAFF", False); : returns *True*
StrMatch("st", "STAFF", False);* : returns *True*
StrMatch("?st", "STAFF", False);* : returns *False*
*StrMatch("st*f", "STAFF", False);* : returns *True*
*StrMatch("st*x", "STAFF", False);* : returns *False*

static function bool isLevelGametype(*string GM, LevelInfo Lvl****)***

- Checks if the string ***GM*** refers to a valid gametype.

Lvl is always the current *Level*, and has to be passed as argument as static functions do not have direct access to their own instance members (such as *Level*).

static function bool isInTeam(*string T, Pawn P****)***

- Checks if the pawn ***P*** is in the same team ***T***.
T can be the team number or verbose (*"0"* and *"Red"* are both recognized as Red Team).

simulated static function texture loadTexture(*string texStr****)***

- Get texture from the string ***texStr***.

simulated static function bool hasInventory(*Pawn P, name InvName, optional bool isClient****)***

- Check if pawn ***P*** has an inventory which class name or one of its parents is called ***InvName***.
isClient indicates if the call is being made from the client instead of the server.

simulated static function Inventory getInventory(*Pawn P, name InvName, optional bool isClient****)***

- Get inventory which class name or one of its parents is called ***InvName*** from the pawn ***P***.
isClient indicates if the call is being made from the client instead of the server.

simulated static function texture getInvisibleTexture(*ERenderStyle TexStyle****)***

- Get the most suitable invisible texture depending on the render style ***TexStyle***.

simulated static function NWMutator getNWMutator(L*evelInfo Lvl, optional bool bClient****)***

- Get the the first loaded NWMutator.
bClient indicates if the call is being made from the client instead of the server.
Lvl is always the current *Level*, and has to be passed as argument as static functions do not have direct access to their own instance members (such as *Level*).

simulated static function SpawnWaterSplash(*Actor A, byte WaterSplashType, bool inWater, ZoneInfo WZone, optional float WaterRingSize, optional bool bBallistic, optional vector curLoc, optional vector olderLoc, optional bool isProcessed, optional Actor AOwner****)***

- Spawn the appropriate water splash effect depending on the actor ***A*** it's being spawned from, and:
WaterSplashType: The water splash effect type;
inWater: If it just entered the water zone;
WZone: The affected water zone;
WaterRingSize: Wave size;
bBallistic: If it's from an hitscan shot;
curLoc: The current hit or actor ***A*** location;
olderLoc: The hit origin or actor ***A*** location in the last Tick;
isProcessed: If all the extra processing to get the best location to spawn the effect can be ignored (in cases where this was already made somehow or if the precision is good enough already);
AOwner: The owner/instigator of the water effect (mostly used for *ZeroPing*).

simulated static function SpawnHitscanWaterSplash(*Actor A, byte WaterSplashType, ZoneInfo WZone, bool bBallistic, vector realLoc, optional Actor AOwner****)***

- A stripped down version of ***SpawnWaterSplash(...)*** function described above meant to be used for hitscan only.

Class: NWInfo	**Parents:** Actor > Info

Description:

This is the class which defines most generic settings of the whole mod, such as detail, quality, performance, toggle of specific features and weapon respawn visual settings.

Properties: For the complete list of properties, check the *[NWCoreVIII.NWInfo]* section of the *INI_NWConfig.pdf* file.

Functions:

static function string getZHackBindOptCommand(`byte i` ***)***

- Returns the string ***KeyBindCommand*** corresponding to the index ***i*** from the ***ZhackBindOptions*** list.

static function byte getZHackBindOptAction(`byte i` ***)***

- Returns the byte ***KeyBindAction*** corresponding to the index ***i*** from the ***ZhackBindOptions*** list.

static function byte getCarcassFXSettings()

- Returns the byte ***CarcassFX***.

static function ECarcassFX GetCarcassFXFromByte(`byte n` ***)***

- Returns the ***ECarcassFX*** enumeration corresponding to the byte ***n***.

static function EAmmoBehaviour GetAmmoBehaviourFromByte(`byte n` ***)***

- Returns the ***EAmmoBehaviour*** enumeration corresponding to the byte ***n***.

static function ERessurectColor GetRessurectColorFromByte(`byte n` ***)***

- Returns the ***ERessurectColor*** enumeration corresponding to the byte ***n***.

Class: NaliWEffects	**Parents:** Actor > Effects

Description:

This is the effects main class, from which most effects (such as explosions for example) are made from.

Properties:

sound ***EffectSound1***
- Effect sound.

float ***SndVol***
- Effect sound volume.

float ***SndRadius***
- Effect sound radius.

Functions:

function MakeSound()
- Play ***EffectSound1***.

Class: NaliWPawn	**Parents:** Actor > Pawn > StationaryPawn

Description:

This is the main pawn class which has the ability to associate players as its master, set a team and remember a player as its master even after said player's reconnect to the server.

Properties:

byte ***MyTeam***
- Pawn team number.

bool ***bMachine***
- This pawn is a machine (for distinction when locking T-Missiles or other machine specific features).

bool ***bIsTeamElement***
- This pawn belongs to a team.

Pawn ***MasterPawn***
- The master of this pawn.

Functions:

function setMaster(*Pawn P****)***
- Set ***P*** as the master of this pawn.

function bool isSameTeam(*byte nTeam****)***
- Returns *True* if the ***nTeam*** corresponds to this pawn team.

function bool isFriend(*actor A****)***
- Returns *True* if the actor ***A*** is a friend.

function SetTeam(*int TeamNum****)***
- Sets this pawn team to ***TeamNum***.

function bool SameTeamAs(*int TeamNum****)***
- Similar to the function ***isSameTeam(...)*** above, but it takes the value of ***bIsTeamElement*** into consideration.

Class: NaliWeapons	**Parents:** Actor > Inventory > Weapon > TournamentWeapon

Description:

This is the main weapon class for all the weapons of this pack, with several features, such as: modifiers, more flexibility, easy to build with less code, *Zero Ping*, glows and overlays, complete separation between fire and alt-fire modes, and more.

Some properties are also explained in the ***Nali Weapon Overview*** and ***Zero Ping*** sections of this document.

Properties:

bool ***denyReplacement***
- Never replace this weapon (deny the replacement mutator intent to do so).

float ***FireRateChange***
- Firerate multiplier.

bool ***bInfinity***
- Infinite ammo.

int ***KickBack***
- Damage extra kickback amount.

float ***MoreDamage***
- Damage multiplier.

bool ***HealthGiver***
- Give health instead of damage (to teammates only, enemies will still get normal damage instead).

float ***Splasher***
- Range multiplier (for splash damage).

bool ***bTheOne***
- Enable ability to carry 3 modifiers at the same time.

float ***FireRateMult1***
- Base fire mode firerate multiplier.

float ***FireRateMult2***
- Base alt-fire mode firerate multiplier.

float ***DamageMult1***
- Base fire mode damage multiplier.

float ***DamageMult2***
- Base alt-fire mode damage multiplier.

float ***WidePlayerViewOffset***
- ***PlayerViewOffset*** value when rendering in a wide screen with a FOV of 100 or above.

bool ***bMegaWeapon***
- Flag this weapon as a super weapon (a weapon of the same class as the Redeemer for example).

bool ***bForceWeaponStay***
- Force weapon to stay.

bool ***bNeverWeaponStay***
- Force weapon to never stay.

bool ***bGameWeaponStay***
- Force weapon to follow the game settings to either stay or not.

bool ***bForceTranslucentCrossHair***
- Force the crosshair to be rendered as translucent.

vector ***AltFireOffset***
- Holds the same purpose of ***FireOffset***, but for the alt-fire alone, which means that ***FireOffset*** is also fire mode exclusive.

float ***FireOffsetXAdjustHidden***
- ***FireOffset*** X axis adjustment value when the weapon is set as hidden in the first person view.

float ***FireOffsetZAdjustHidden***
- ***FireOffset*** Z axis adjustment value when the weapon is set as hidden in the first person view.

float ***AltFireOffsetXAdjustHidden***
- ***AltFireOffset*** X axis adjustment value when the weapon is set as hidden in the first person view.

float ***AltFireOffsetZAdjustHidden***
- ***AltFireOffset*** Z axis adjustment value when the weapon is set as hidden in the first person view.

sound ***FiringAmbSound***
- Ambient sound during fire mode.

sound ***AltFiringAmbSound***
- Ambient sound during alt-fire mode.

byte ***FiringSndVol***
- Ambient sound volume during fire mode.

byte ***AltFiringSndVol***
- Ambient sound volume during alt-fire mode.

byte ***FiringSndPitch***
- Ambient sound pitch during fire mode.

byte ***AltFiringSndPitch***

- Ambient sound pitch during alt-fire mode.

byte ***FiringSndRadius***

- Ambient sound radius during fire mode.

byte ***AltFiringSndRadius***

- Ambient sound radius during alt-fire mode.

bool ***bHighFireRate***

- Flag to indicate that the fire mode has a high firerate (this way, the Fast modifier will affect the weapon damage instead of its already high firerate).

bool ***bAltHighFireRate***

- Flag to indicate that the alt-fire mode has a high firerate (this way, the Fast modifier will affect the weapon damage instead of its already high firerate).

texture ***CustomCross***

- Custom crosshair texture.

float ***CrossHairScale***

- Custom crosshair texture rendering scale.

mesh ***PlayerViewMeshLeft***

- First person view left handed mesh to render.

mesh ***PlayerViewMeshRight***

- First person view right handed mesh to render.

mesh ***HandPartMeshL[2]***

- First person view left handed hand models.

mesh ***HandPartMeshR[2]***

- First person view right handed hand models.

bool ***bForceHands***

- Force the rendering of hands in first person view independently from the player settings on the weapon hands display.

byte ***HandsBaseFatness***

- Hands rendering normal fatness value.

vector ***RenderOffsetSelect***

- Max rendering offset during the *Select* animation.

float ***AnimMaxFrame***

- ***AnimFrame*** value which marks the end of the *Select* animation, to be used in the correct calculation of the rendering offset during such animation.

vector ***RenderOffsetFire***

- Max rendering offset during the *Fire* animation.

vector ***RenderOffsetAltFire***

- Max rendering offset during the *AltFire* animation.

float ***AnimMaxFrameFire***

- ***AnimFrame*** value which marks the end of the *Fire* animation, to be used in the correct calculation of the rendering offset during such animation.

float ***AnimMaxFrameAltFire***

- ***AnimFrame*** value which marks the end of the *AltFire* animation, to be used in the correct calculation of the rendering offset during such animation.

vector ***CenterPlayerViewOffset***

- First person rendering offset when set to render at the center.

vector ***WideCenterPlayerViewOffset***

- First person rendering offset when set to render at the center in wide screens with a FOV value of 100 or higher.

bool ***bInstantHitWaterFX***

- Enable special water effects when hitscan hits a water zone.

bool ***bBallisticWaterFX***

- Enable the alternate type of water effects (which is more vertical based).

byte ***WaterSplashType***

- Water effects type.

GlowSet ***FirstPersonGlowFX[8]***

- First person view glow effects (for more info, check the ***Nali Weapon Overview*** from this document).

GlowSet ***PickupGlowFX[8]***

- Pickup view glow effects (for more info, check the ***Nali Weapon Overview*** from this document).

OverlModel ***FirstPersonOverFX[3]***

- First person view overlay effects (for more info, check the ***Nali Weapon Overview*** from this document).

OverlModel ***PickupOverFX[3]***

- Pickup view overlay effects (for more info, check the ***Nali Weapon Overview*** from this document).

texture ***HandSkin***

- Default hands skin texture.

float ***RessurectionTime***

- Respawn effect duration time.

sound ***RessurectSound1***
- Respawn effect first sound.

sound ***RessurectSound2***
- Respawn effect last sound.

string ***NoAmmoMsgString***
- Message to display when the fire mode has insufficient ammo to be used.

string ***NoAltAmmoMsgString***
- Message to display when the alt-fire mode has insufficient ammo to be used.

bool ***bRenderOptionsOnHiddenWeapon***
- Render specific options when the weapon is hidden in first person view.

bool ***bRenderCustomOnHiddenWeapon***
- Call the rendering functions ***getRenderCustomCanvasColor()*** and ***getRenderCustomCanvasText()*** to return the color and text to be rendering when the weapon is hidden in first person view.

NROptions ***renderOptions[****16****]***
- Rendering options when the weapon is hidden in first person view, where ***optionText*** is the text and ***optionColor*** the color of the text.

byte ***DeployAIPriority***
- Priority of this deployable weapon (for AI purposes).

bool ***cannotRespawn***
- Remove the ability for this weapon to respawn.

bool ***bRenderHandsOnly***
- Flag so only hands are rendering and not the weapon itself.

bool ***isZPWeapon***
- Enable *Zero Ping* for this weapon.

float ***ZPMaxFirerate***
- Max firerate for *Zero Ping* fire mode.

float ***ZPMaxFirerateAlt***
- Max firerate for *Zero Ping* alt-fire mode.

bool ***bFireHitScan***
- The fire mode is hitscan.

bool ***bAltFireHitScan***
- The alt-fire mode is hitscan.

Functions:

function ResetModifiers(*NaliWeapons Copy****)***

- Remove all modifiers from the weapon ***Copy***.

function AddAuxMutator(*Class<Mutator> MutClass****)***

- Loads up the mutator of the class ***MutClass*** if it's not loaded already.

simulated function PlayNoAmmoFiring()

- Event called whenever the weapon has no ammo left for its fire mode, but has still ammo to have the weapon up.

simulated function PlayNoAmmoAltFiring()

- Event called whenever the weapon has no ammo left for its alt-fire mode, but has still ammo to have the weapon up.

simulated function bool CheckAmmo(*bool bAltFire****)***

- Returns *True* if the weapon has enough ammo to use the fire mode passed as ***bAltFire***.

simulated function bool isInFireState(*optional bool bAltFire****)***

- Returns *True* if the weapon is in a fire mode passed as ***bAltFire***.

function ActivateTheOne()

- Activate The One modifier.

function GiveSpecificModifier(*NaliWeapons Copy, byte type****)***

- Give a specific modifer of the type ***type*** to the weapon ***Copy***, taking into account if the weapon is a super weapon.

function GiveModifierType(*NaliWeapons Copy, int type****)***

- Give a specific modifer of the type ***type*** to the weapon ***Copy***, regardless if it's a super weapon or not.

simulated function GetCrosshairCoords(*out float posX, out float posY, canvas Canvas, texture Icon, float Scale****)***

- Get the crosshair coordinates in the ***Canvas*** depending on the weapon handedness, level gametype and the rendering scale ***Scale***.
Icon is the crosshair texture, and the return values are set in the reference arguments ***posX*** and ***posY***.

simulated function color GetCrosshairColor(*PlayerPawn P****)***

- Get the crosshair color depending on the player ***P*** settings.

simulated function PrePostRenderOther(*canvas Canvas, float Scale****)***

- Event called before the rendering in ***Canvas*** in the ***PostRender(...)*** function takes place.
Scale is the horizontal screen scale relative the resolution of 640x480.

simulated function PostPostRenderOther(*canvas Canvas, float Scale****)***

- Event called after the rendering in ***Canvas*** in the ***PostRender(...)*** function takes place.
Scale is the horizontal screen scale relative the resolution of 640x480.

function Projectile AltProjectileFire(*class<projectile> ProjClass, float ProjSpeed, bool bWarn****)***

- Alt-fire version of ***ProjectileFire(...)***.

function float GetDamageMult(*optional bool bAlt****)***

- Get damage multiplier depending on the fire mode ***bAlt***.

simulated function float GetFirerateMult(*optional bool bAlt****)***

- Get firerate multiplier depending on the fire mode ***bAlt***.

function bool useAmmoToFire(*optional bool bAltFire****)***

- Returns *True* if the weapon should use ammo depending on the fire mode ***bAltFire***.

simulated function ProcessOther(*Actor Other, Vector HitLocation, Vector HitNormal, Vector StartTrace****)***

- Event called as result of the hitscan made during the weapon fire.

simulated function ProcessWaterFX(*Actor Other, vector HitLoc, vector StartLoc****)***

- Function to spawn water effects at the location ***HitLoc***, being ***Other*** the *Actor* responsible for it and ***StartLoc*** the origin of the hit.

simulated function vector CalcNewDrawOffset()

- Function which completely replaces the standard ***CalcDrawOffset()***.

simulated function byte getRenderOptionIndex()

- Get render option index when ***bRenderOptionsOnHiddenWeapon=True***.

simulated function string getRenderCustomCanvasText()

- Get render option custom text when ***bRenderCustomOnHiddenWeapon=True***.

simulated function color getRenderCustomCanvasColor()

- Get render option custom color when ***bRenderCustomOnHiddenWeapon=True***.

simulated function RenderOther(*canvas Canvas, vector Loc, rotator Rot****)***

- Event called from ***RenderOverlays(...)*** to be able to process further custom rendering as needed.
Loc and ***Rot*** are the location and rotation the weapon is being rendered with.

function bool isHeadShotDmg(*Actor Other, vector HitLocation****)***

- Returns *True* if the damage made to ***Other*** in ***HitLocation*** is a headshot.

function bool giveFiredHealth(*Actor Other, float dmg****)***

- Returns *True* if ***Other*** should receive health relative the damage amount of ***dmg***.

function bool checkDeployPriority(*Actor A****)***

- Returns *True* if this weapon has deployment priority over all others from ***A*** inventory list.

Class: KeyedNaliWeapon	**Parents:** Actor > Inventory > Weapon > TournamentWeapon > NaliWeapons

Description:

This is an abstract *NaliWeapons* subclass, which adds one extra special feature: the ability to make a weapon based in numeric key pressing. Example: Megaton.

This class works like a normal weapon when picked up, but once the fire or alt-fire button is pressed, it can enter into the key pressing mode instead to type codes, passwords, times, whatever you want your special weapon (or tool) to do.

Once within the key pressing mode, there can be different stages or states of key pressing. Example: Megaton has 2 states: the time state and the password state.

Since the average player doesn't know how to handle a weapon like this, another feature are the default and extended help texts which are rendered on the screen in each state, to guide the player through on using this kind of tool or weapon.

Properties:

name ***NumberAnimIn[****10****]***
- Animation names for when the buttons are being pressed in. The index is the corresponding number.

name ***NumberAnimOut[****10****]***
- Animation names for when the buttons were already pressed and are moving outwards. The index is the corresponding number.

name ***EnterKeyModeAnim***
- Animation for when entering key pressing mode.

name ***LeaveKeyModeAnim***
- Animation for when leaving key pressing mode.

name ***NumberFinishAnimIn***
- Animation for when pressing in the number buttons.

name ***NumberFinishAnimOut***
- Animation for when leaving the last button pressed.

float ***NumberPressRate***
- Animation rate when pressing number buttons.

float ***EnterKeyModeRate***
- Animation rate when entering key pressing mode.

float ***LeaveKeyModeRate***
- Animation rate when leaving key pressing mode.

float ***KeysTweenTime***
- Tween time when moving from one number button to another.

byte ***KeysBufferSize***
- The number of digits.

byte ***KeyStatesAmount***
- The number key pressing states.

sound ***KeyPressSnd***
- Sound to play when a key number is pressed.

sound ***KeyPressFinishSnd***
- Sound to play when key stops being pressed.

bool ***bFireAsKeyStarter***
- Use the fire click to enter the key pressing mode.

bool ***bAltFireAsKeyStarter***
- Use the alt-fire click to enter the key pressing mode.

bool ***bFireAsKeyEnder***
- Use the fire click to leave the key pressing mode.

bool ***bAltFireAsKeyEnder***
- Use the alt-fire click to leave the key pressing mode.

bool ***bLeaveKeyModeOnceSet***
- Automatically leave the key pressing mode once the digits were fully entered in all the needed states.

string ***defaultKeysHelpText***
- Default help text.

color ***defaultKeysHelpColor***
- Default help text color.

Functions:

function WeaponKeyPressed(*byte k, byte bufferPos****)***
- Event called whenever a key is pressed, where ***k*** is the number of the pressed key, and ***bufferPos*** is current position of that key in the buffer.

function KeyStateChange(*byte newState****)***
- Event called whenever the key pressing state changes to a new one, being ***newState*** the new state.

function KeyModeToggled(*bool isKeyMode****)***
- Event called whenever the key pressing mode is entered or left, being ***isKeyMode*** the flag which indicates which one is it.

simulated function KeyClientTick(*float Delta****)***
- Client side ***Tick(****float Delta****)*** event which only works during the key pressing mode.

simulated function string GetKeysHelpText()
- Get custom help text.

Class: NaliProjectile	**Parents:** Actor > Projectile

Description:

This is the main projectile class, which is easier to setup and work with to do normal projectiles, as all the common projectile features (smoke generation, trails, water effects, zone-dependent speed, extended damage settings, etc...) and the ins and outs of replication in each one are already entirely covered, eliminating the need of having to write hundreds or even thousands of lines of code to do fairly simple projectiles which behave in a similar fashion of the other 90% of the existing ones, while giving room for new custom features effortlessly.

Properties:

class<Actor> ***TrailClass1***
- Primary trail actor class.

class<Actor> ***TrailClass2***
- Secondary trail actor class.

vector ***TrailOffset1***
- Primary trail actor offset.

vector ***TrailOffset2***
- Secondary trail actor offset.

bool ***bSpawnServerTrail***
- Spawn the trail server-side.

bool ***enableSmokeGen***
- Enable smoke generation.

bool ***bSmokeGenUnderWater***
- Enable smoke generation underwater.

class<effects> ***SmokeClass***
- Smoke effect class.

class<effects> ***UnderWaterSmokeClass***
- Underwater smoke effect class.

float ***SmokeGenRateMax***
- Smoke generation max rate.

float ***SmokeGenRateMin***
- Smoke generation min rate.

float ***SmokeXOffset***
- Smoke generation offset relative the projectile direction (in the X axis).

float ***UnderWaterSmokeGenRateMax***
- Underwater smoke generation max rate.

float ***UnderWaterSmokeGenRateMin***
- Underwater smoke generation min rate.

float ***UnderWaterSmokeXOffset***
- Underwater smoke generation offset relative the projectile direction (in the X axis).

bool ***bClientOnlySmokeGen***
- Spawn the smoke effects client-side only.

bool ***bWaterHitFX***
- Spawn water effects when entering or leaving a water zone.

float ***WaterFXScale***
- Water effect scale.

float ***WaterSpeedScale***
- Water zone entering speed multiplier.

byte ***WaterSplashType***
- Water effect type.

float ***WaterWaveSize***
- Water wave size.

float ***ProjAccel***
- Acceleration.

bool ***CanHitInstigator***
- Can hit and damage the instigator.

float ***HitInstigatorTimeOut***
- Delay to enable any possible hit or damage against the instigator (only when ***CanHitInstigator=True*** and ***Lifespan>0.0***).

bool ***bDirectHit***
- Damage only the actor this projectile hits and disable splash damage.

float ***ExplosionNoise***
- Amount of noise generated on hit (for AI awareness of the projectile).

bool ***bDirDecal***
- Generate a directional decal on hit rather than normal one.

float ***DmgRadius***
- Damage radius on hit.

bool ***bNeverHurtInstigator***
- Never hurt instigator.

bool ***bNoHurtTeam***
- Never hurt teammates.

bool ***bDirectionalBlow***
- Process the hit momentum into a single direction (the one of the projectile itself).

float ***TimeOut1***
- Time to call the event ***TimedOut1()***.

float ***TimeOut2***
- Time to call the event ***TimedOut2()***.

float ***TimeOut3***
- Time to call the event ***TimedOut3()***.

bool ***bRepeating1***
- Make ***TimedOut1()*** a repeatable event.

bool ***bRepeating2***
- Make ***TimedOut2()*** a repeatable event.

bool ***bRepeating3***
- Make ***TimedOut3()*** a repeatable event.

class<Effects> ***TrailingClass***
- Trailing effect class.

float ***TrailingSize***
- Trailing effect size (on the X axis).

vector ***TrailingSpawnOffset***
- Trailing effect starting offset.

bool ***bReverseTrailingPoint***
- Reverse the direction and way the trailing effect is spawned.

Functions:

simulated function SetTimeOut1(*float Tout, optional bool bRepeat****)***
- Set a timeout which calls the event ***TimedOut1()*** after ***Tout*** seconds and which keeps on repeating if ***bRepeat*** is set to *True*.

simulated function SetTimeOut2(*float Tout, optional bool bRepeat****)***
- Set a timeout which calls the event ***TimedOut2()*** after ***Tout*** seconds and which keeps on repeating if ***bRepeat*** is set to *True*.

simulated function SetTimeOut3(*float Tout, optional bool bRepeat****)***

- Set a timeout which calls the event ***TimedOut3()*** after ***Tout*** seconds and which keeps on repeating if ***bRepeat*** is set to *True*.

simulated function ExplodeX(*vector HitLocation, vector HitNormal, optional actor A****)***

- Event called whenever the projectile hits something.

function ExplodeOnWall(*vector HitNormal, actor Wall****)***

- Event called whenever the projectile hits a BSP surface.

function bool ProcessHurtRadiusVictim(*Actor Victim****)***

- Returns *True* if this ***Victim*** cannot be damaged.

function PostProcessVictim(*Actor Victim****)***

- Event called for any ***Victim*** which was found to be in the range of damage of this projectile.

Class: NaliDynColorProjectile	**Parents:** Actor > Projectile > NaliProjectile

Description:

This is a *NaliProjectile* subclass with the feature of being able to change color dynamically over time.

Properties:

ColorProj ***ProjectileColor[****8****]***

- Projectile color settings, with up to 8 entries for 8 possible color transitions over time, and has the following properties:
  bUseThisColor: Use this color entry;
  ProjColor: Projectile color;
  bFadeToNext: Fade to the next color entry;
  FadeTime: Fade time to the next color;
  LifeTime: Lifespan of this color entry;

texture ***RedCompTex[****9****]***

- Red channel of the projectile texture. If the projectile is a sprite, only index 0 is used, if is a mesh, each index corresponds to a ***MultiSkins*** entry, except index 8 which corresponds to ***Texture***.

texture ***GreenCompTex[****9****]***

- Green channel of the projectile texture. If the projectile is a sprite, only index 0 is used, if is a mesh, each index corresponds to a ***MultiSkins*** entry, except index 8 which corresponds to ***Texture***.

texture ***BlueCompTex[****9****]***

- Blue channel of the projectile texture. If the projectile is a sprite, only index 0 is used, if is a mesh, each index corresponds to a ***MultiSkins*** entry, except index 8 which corresponds to ***Texture***.

Functions: This class has no functions of its own which worth to be mentioned in this document.

Class: NaliNuclearProjectile	**Parents:** Actor > Projectile > NaliProjectile

Description:

This is a *NaliProjectile* subclass with the feature of being recognized as a nuclear device.

Properties:

byte ***NuclearLevel***
- Nuclear level of the nuke (*0* to *5*).

float ***CriticalImpactDist***
- Max distance from impact to be considered critical.

float ***DeathImpactDist***
- Max distance from impact to be considered a death zone.

float ***DangerousImpactDist***
- Max distance from impact to be considered dangerous.

color ***defaultInfoColor***
- Default nuclear threat text color.

color ***warningInfoColor***
- Default nuclear threat warning color.

string ***defaultInfoText***
- Default nuclear threat text.

bool ***denyNuclearWarning***
- Remain undetected as a nuclear threat.

Functions:

function int getImpactArea(*Actor A****)***
- Get danger classification based on the impact area and the location of ***A***. This value varies between *0* (critical) and *3* (safe), while *-1* means that the classification is unknown.

simulated function string getNukeInfoText()
- Get nuclear threat text.

simulated function color getNukeInfoColor()
- Get nuclear threat text color.

Class: NaliPickups	**Parents:** Actor > Inventory > Pickup > TournamentPickup

Description:

This is the main pickup class, from which health, armor, damage and any other kinds of pickups can be made.
The amount of different properties and the straigh to the point existent functions, make this class suitable to do any kind of pickup you desire effortlessly.

Properties:

bool ***denyReplacement***

- Never replace this pickup (deny the replacement mutator intent to do so).

byte ***PickupPriority***

- Priority over other pickups (for effects, armor, etc...).

EPickupType ***PickupType***

- Pickup type:
  PCK_Health: Health pickup;
  PCK_Armor: Armor pickup;
  PCK_ExtraDamage: Damage boost pickup;
  PCK_Invisibility: Invisibility cloaking pickup;
  PCK_Other: Custom pickup (custom code).

bool ***bTemporaryPickup***

- Pickup with a lifespan defined by its ***Charge***.

bool ***bCumulativeCharge***

- When a pickup of the same type is picked up, add its ***Charge*** to the already existing one in the players possession.

int ***MaxCharge***

- Max value of ***Charge***.

float ***ChargeDecreaseRate***

- Decrease rate of ***Charge***.

bool ***bBoots***

- Show the boots pickup icon when using this pickup.

bool ***bSuperHealth***

- Add health on top of the player's default amount (like health vials).

int ***HealthAmount***

- Amount of health to give.

float ***DamageMult***

- Weapon damage multiplier.

byte ***PawnVisibility***

- Pawn visibility (for AI purposes).

bool ***bSuperArmor***

- Make bots to not look for this armor if they're wearing some kind of defense relic (for AI purposes).

bool ***bConsumeOtherArmors***

- Consume and destroy other armors when picking up this one (only armor with lower priority is affected).

bool ***bShield***

- Show the shield pickup icon when using this pickup.

bool ***bThighs***

- Show the thigh pads pickup icon when using this pickup.

class<Actor> ***PickupFXClasses[16]***

- Pickup effects classes to be spawned when this pickup is visible (example: dynamic coronas, particles, any effect at all).

bool ***bWeaponFX***

- Enable weapon effects while using this pickup.

bool ***bOverlayWeaponFX***

- Enable weapon overlay effects while using this pickup.

bool ***bThirdPersonWeaponFX***

- Enable weapon third person view effects while using this pickup.

bool ***bWeaponAffector***

- Set this pickup to be the current weapon affector.

bool ***bNullifyOtherWeaponOverlayerFX***

- Disable other weapon effects from other pickups of lower priority while this one is active.

ERenderStyle ***WeaponFXStyle***

- Weapon effect rendering style.

ERenderStyle ***WeaponFXOverlayStyle***

- Weapon overlay effect rendering style.

ERenderStyle ***WeaponFXThirdPersonStyle***

- Weapon third person effect rendering style.

bool ***bTeamBasedWeaponFX***

- Make the weapon effect texture be team based.

texture ***WeaponFXTex[5]***

- Weapon effect texture. If the effect is team based, indexes *0* to *3* correspond to the team (being *4* the neutral one), otherwise only index *0* is used.

texture ***WeaponFXOverlayTex[5]***

- Weapon overlay effect texture. If the effect is team based, indexes *0* to *3* correspond to the team (being *4* the neutral one), otherwise only index *0* is used.

texture ***WeaponFXThirdPersonTex[5]***

- Weapon third person effect texture. If the effect is team based, indexes *0* to *3* correspond to the team (being *4* the neutral one), otherwise only index *0* is used.

bool ***bWeaponFXEnviroMap***

- Make the weapon effect to be environment mapped.

bool ***bWeaponFXOverlayEnviroMap***

- Make the weapon overlay effect to be environment mapped.

bool ***bWeaponFXThirdPersonEnviroMap***

- Make the third person weapon effect to be environment mapped.

byte ***WeaponFXOverlayExtraFatness***

- Weapon overlay effect extra fatness.

bool ***bWeaponFXUnlit***

- Make the weapon effect unlit.

bool ***bWeaponFXOverlayUnlit***

- Make the weapon overlay effect unlit.

bool ***bWeaponFXThirdPersonUnlit***

- Make the weapon third person effect unlit.

float ***WeaponFXGlow***

- Weapon effect glow amount.

float ***WeaponFXOverlayGlow***

- Weapon overlay effect glow amount.

float ***WeaponFXThirdPersonGlow***

- Weapon third person effect glow amount.

class<NWeaponOverFX> ***WeaponFXOverlayClass***

- Weapon overlay effect class.

bool ***bHandsFX***

- Enable weapon hands effects while using this pickup.

bool ***bHandsOverlayFX***

- Enable weapon hands overlay effects while using this pickup.

bool ***bNullifyOtherHandsOverlayerFX***

- Disable other weapon hands effects from other pickups of lower priority while this one is active.

ERenderStyle ***HandsFXStyle***

- Weapon hands effect rendering style.

ERenderStyle ***HandsFXOverlayStyle***

- Weapon hands overlay effect rendering style.

bool ***bTeamBasedHandsFX***

- Make the weapon hands effect texture be team based.

texture ***HandsFXTex[5]***

- Weapon hands effect texture. If the effect is team based, indexes *0* to *3* correspond to the team (being *4* the neutral one), otherwise only index *0* is used.

texture ***HandsFXOverlayTex[5]***

- Weapon hands overlay effect texture. If the effect is team based, indexes *0* to *3* correspond to the team (being *4* the neutral one), otherwise only index *0* is used.

bool ***bHandsFXEnviroMap***

- Make the weapon hands effect to be environment mapped.

bool ***bHandsFXOverlayEnviroMap***

- Make the weapon hands overlay effect to be environment mapped.

byte ***HandsFXOverlayExtraFatness***

- Weapon hands overlay effect extra fatness.

bool ***bHandsFXUnlit***

- Make the weapon hands effect unlit.

bool ***bHandsFXOverlayUnlit***

- Make the weapon hands overlay effect unlit.

float ***HandsFXGlow***

- Weapon hands effect glow amount.

float ***HandsFXOverlayGlow***

- Weapon hands overlay effect glow amount.

class<NWeaponOverFX> ***HandsFXOverlayClass***

- Weapon hands overlay effect class.

bool ***bPawnFX***

- Enable player effects while using this pickup.

bool ***bPawnOverlayerFX***

- Enable player overlay effects while using this pickup.

bool ***bNullifyOtherPawnOverlayerFX***

- Disable other player effects from other pickups of lower priority while this one is active.

class<NaliPickupPawnOV> ***FXPawnOVClass***

- Player overlay effect class.

ERenderStyle ***PawnFXStyle***

- Player effect rendering style.

ERenderStyle ***PawnFXOverlayStyle***

- Player overlay effect rendering style.

bool ***bTeamBasedPawnFX***

- Make the player effect texture be team based.

texture ***PawnFXTex[5]***

- Player effect texture. If the effect is team based, indexes *0* to *3* correspond to the team (being *4* the neutral one), otherwise only index *0* is used.

texture ***PawnFXOverlayTex[5]***

- Player overlay effect texture. If the effect is team based, indexes *0* to *3* correspond to the team (being *4* the neutral one), otherwise only index *0* is used.

bool ***bPawnFXEnviroMap***

- Make the player effect to be environment mapped.

bool ***bPawnFXOverlayEnviroMap***

- Make the player overlay effect to be environment mapped.

byte ***PawnFXOverlayExtraFatness***

- Player overlay effect extra fatness.

bool ***bPawnFXUnlit***

- Make the player effect unlit.

bool ***bPawnFXOverlayUnlit***

- Make the player overlay effect unlit.

float ***PawnFXGlow***

- Player effect glow amount.

float ***PawnFXOverlayGlow***

- Player overlay effect glow amount.

byte ***PawnFXAmbientGlow***

- Player overlay effect ambient glow.

bool ***bPawnFXLight***

- Enable player dynamic light effect.

byte ***PawnFXLightHue***

- Player dynamic light effect hue.

byte ***PawnFXLightSaturation***

- Player dynamic light effect saturation.

byte ***PawnFXLightBrightness***
- Player dynamic light effect brightness.

byte ***PawnFXLightRadius***
- Player dynamic light effect radius.

ELightType ***PawnFXLightType***
- Player dynamic light effect type.

ELightEffect ***PawnFXLightEffect***
- Player dynamic light effect behavior.

byte ***PawnFXLightPeriod***
- Player dynamic light effect period.

byte ***PawnFXLightPhase***
- Player dynamic light effect phase.

byte ***PawnFXLightCone***
- Player dynamic light effect cone.

bool ***bForceRotatingPickupOnReplace***
- Force rotating pickup if replaced.

float ***ChargerScale***
- Pickup charger scale.

float ***PlacementZOffset***
- Z-axis offset on the placement of the pickup relative the charger location.

Functions:

function PrePickupInit()
- Event called before all the processing relative the pickup of the pickup takes place.

function PostPickupInit()
- Event called after all the processing relative the pickup of the pickup takes place.

function ChargeDecreaseEvent()
- Event called whenever the ***Charge*** is decremented.

function bool PickupExpired()
- Event called whenever the ***Charge*** reaches zero.

function SetupExtras()
- Event called to setup any extra properties or do any extra processing on the pickup of the pickup.

function ResetExtras(*optional bool bResetOnly****)***

- Event called to reset any extra properties or do any reset processing once the pickup expires or gets destroyed.

bResetOnly is passed as True if the event was called with the intent to just reset and not getting destroyed or expired.

function ResetExtras(*optional bool bResetOnly****)***

- Event called to reset any extra properties or do any reset processing once the pickup expires or gets destroyed.

Class: NaliAmmo	**Parents:** Actor > Inventory > Pickup > Ammo > TournamentAmmo

Description:

This is the ammo main class of the pack. It doesn't bring much to the table besides the ability of flagging as a super weapon ammo and support to have an opening animation.

Properties:

bool ***bMegaAmmo***

- Flag this as a super weapon ammo.

bool ***denyReplacement***

- Never replace this ammo (deny the replacement mutator intent to do so).

name ***OpenedAnimSeq***

- Opened animation sequence.

name ***ClosedAnimSeq***

- Closed animation sequence.

name ***AmmoAnimSeq***

- Opening animation sequence.

float ***AmmoAnimRate***

- Opening animation rate.

float ***AmmoAnimTime***

- Opening animation time.

sound ***AmmoAnimSound***

- Opening animation sound.

sound ***EndAnimSound***

- Opening animation finish sound.

Functions: This class has no functions of its own which worth to be mentioned in this document.

Class: NaliFullMeshFX	**Parents:** Actor > Effects > NaliWEffects

Description:

This is a special *NaliWEffects* subclass, directed exclusively to mesh based effects, and with the features of being able to be fully rendered no matter the angle of view, or/and being rendered always pointed to the player (like a sprite).

To avoid weird lighting glitches, the mesh should be rendered as unlit.

Properties:

bool ***bEnableFullMeshView***
- Enable full mesh rendering.

float ***RadiusView***
- Radius within the mesh is fully rendered.

bool ***bAffectByDrawScale***
- Make ***RadiusView*** be affected by the mesh rendering scale.

bool ***bDirectionalMesh***
- Enable directional mesh rendering.

Functions: This class has no functions of its own which worth to be mentioned in this document.

Class: NaliTrail	**Parents:** Actor > Effects

Description:

This is the main projectiles trail class.

Properties:

vector ***PrePivotRel***
- Trail rendering offset.

bool ***bReplicatePrePivotRel***
- Replicate the ***PrePivotRel*** value to the client if spawned in the server.

bool ***UpdateInClientOnly***
- Update the trail position in the client only.

Functions: This class has no functions of its own which worth to be mentioned in this document.

Class: NWCarcassFX	**Parents:** Actor > Effects > BulletImpact > UT_WallHit > UT_HeavyWallHitEffect

Description:

This class is used to spawn effects in the gibs and carcasses of dead players.

Properties:

float ***CarcassRadiusCheck***

- Radius check of gibs and carcasses.

name ***ValidCarcassTypes[8]***

- Valid gibs and carcasses class names to check for.

bool ***bSplashEffected***

- Have the radius affected by the Splasher modifier from the instigator weapon or projectile.

EFXSplashType ***SplashType***

- Effect splash type (for gameplay and detail settings):
  SPLX_Precise: Precise effect (amount of gibs affected: low);
  SPLX_Moderate: Moderate effect (amount of gibs affected: normal);
  SPLX_Splash: Moderate effect (amount of gibs affected: high).

Functions:

simulated function ExecuteCarcass(*Carcass c, optional byte chosenIndex****)***

- Event called for any gib or carcass ***c*** found within the effect radius.
chosenIndex indicates the entry of ***ValidCarcassTypes*** detected.

Class: NWCoronaFX	**Parents:** Actor > NaliWActor

Description:

This is the main dynamic corona effect class, with the features of being able to also render a lens flare effect, being attached to any object and change over time (flicker, fade, etc...).

Properties:

float ***MaxDistance***
- Max distance from which the corona is visible.

float ***StartScaleTime***
- Scale up time.

float ***EndScaleTime***
- Scale down time.

float ***FadeInTime***
- Fade in time.

float ***FadeOutTime***
- Fade out time.

float ***EndScaleCoef***
- Scale when starting to scale down.

float ***StartScaleCoef***
- Scale when finishing to scale up.

texture ***CoronaSprite***
- Corona texture.

float ***MaxCoronaSize***
- Corona size when rendered at its max distance.

float ***MinCoronaSize***
- Corona size when rendered up close.

float ***CGlow***
- Corona glow.

float ***CGlowMax***
- Corona max glow (for flickering effect).

float ***CGlowMin***
- Corona min glow (for flickering effect).

float ***DScaleCoefMax***

- Corona max scale multiplier (for flickering effect).

float ***DScaleCoefMin***

- Corona min scale multiplier (for flickering effect).

bool ***enableLensFlare***

- Enable lens flare effect.

bool ***bFadeOutOnView***

- Fade out the lens flare depending on the view angle.

bool ***bFixedOverAbsRayLength***

- Enable fixed lens flare max length.

float ***AbsRayLength***

- Fixed lens flare max length.

NWLensFlare ***LensFlareParts[****16****]***

- Lens flare parts:
  bEnableLens: Enable this lens flare entry to be rendered;
  DistanceCoefCutOff: Distance percentage this lens flare is rendered at;
  LensSprite: Lens flare texture;
  MaxLensSize: Max lens flare size;
  MinLensSize: Min lens flare size;
  bAffectByCorona: Affect scaling and glow by the main corona;
  LensGlow: Lens flare glow.

Functions: This class has no functions of its own which worth to be mentioned in this document.

Class: NuclearExplosions	**Parents:** Actor > NaliWActor

Description:

This is the main nuclear explosion class, which features plenty of properties and functions to setup to make a proper nuclear damaging effect.

Properties:

class<Light> ***DynamicLightClass***
- Dynamic light class.

sound ***NuclearDistSnd***
- Nuclear explosion distant sound.

float ***NuclearDistSndVol***
- Nuclear explosion distant sound volume.

float ***NuclearDistSndRadius***
- Nuclear explosion distant sound radius.

class<Actor> ***NuclearCoronaClass***
- Corona class.

float ***ShockRadius***
- Shockwave max radius.

float ***NucleusRadius***
- Nucleus max radius.

float ***ShockTime***
- Shockwave duration.

float ***NucleusTime***
- Nucleus duration.

int ***ShockMomentum***
- Shockwave max kickback.

int ***NucleusMomentum***
- Nucleus max kickback.

name ***ShockDmgType***
- Shockwave damage type.

name ***NucleusDmgType***
- Nucleus damage type.

bool ***bGrowingNucleus***
- Enable growing nucleus.

float ***MinNucleusRad***
- Min nucleus radius.

class<NWShockwave> ***ShockwaveFXClass***
- Shockwave effect class.

bool ***bUpdateGroundBreaking***
- Enable ***UpdateGroundBreaking(****float Delta****)*** event.

bool ***bUpdateTickedEvents***
- Enable ***UpdateTickedEvents(****float Delta****)*** event.

Functions:

function SpawnShockwave()
- Spawn shockwave effect.

simulated function SetTimeOut(*float time, byte i, optional bool isClient****)***
- Set a timeout call after ***time*** seconds.
i is the ID of the timeout function to call (from *0* to *7*, for ***TriggerTimeOut0()*** to ***TriggerTimeOut7()*** respectively).
isClient indicates if the call is made client-side or server-side.

simulated function UpdateGroundBreaking(*float Delta****)***
- Client-side only version of ***Tick(****float Delta****)***.

function UpdateTickedEvents(*float Delta****)***
- Server-side only version of ***Tick(****float Delta****)***.

function bool SkipThisActor(*actor A****)***
- Returns *True* if ***A*** should be skipped from any kind of damage or effect during the blast.

function SpawnCorona()
- Spawn corona effect.

simulated function SpawnLight()
- Spawn dynamic light effect.

function SpawnExtraFX(*Actor A, vector Dir****)***
- Spawn extra effects from the hit actor ***A*** towards the direction ***Dir***.

Class: NWNukeShockFX	**Parents:** Actor > NaliWActor

Description:

This is the nuclear effects main class, featuring: shake system, dynamic player FOV effect, muffling, dynamic ambient sound, hit sound effect, flash and possibility to spawn custom effects during the hit.

Although it was designed mostly of nuclear effects, it can and is used in many non-nuclear effects due to its shaking system.

Properties:

NukeFXStruct ***NukeFX[8]***

- List of effects:
  - *bActive*: Activate this effect;
  - *bDistanceBased*: Distance based effect;
  - *TimeDelay*: Delay until the effect activates;
  - *TimeDuration*: Duration of the effect;
  - *DistOffsetMin*: Distance min offset;
  - *DistOffsetMax*: Distance max offset;
  - *MinMufflingDist*: Min distance for the effect to get muffled;
  - *BlurNoise*: Shake noise;
  - *Shake*: Shake effect;
  - *ShakeRate*: Shake effect rate;
  - *bRisingShake*: Rising shake with time/distance effect;
  - *bMuffledShake*: Enable shake muffling;
  - *bFadeShakeOnDistance*: Make shake weaker on distance;
  - *ShakeDistance*: Max distance to be affected by shaking;
  - *AmbSound*: Ambient sound;
  - *StartVolume*: Ambient sound starting volume;
  - *EndVolume*: Ambient sound ending volume;
  - *StartPitch*: Ambient sound starting pitch;
  - *EndPitch*: Ambient sound ending pitch;
  - *bMuffledAmb*: Enable ambient sound muffling;
  - *NegMinTimeDelay*: Time subtraction from the main time duration for the ambient sound to finish;
  - *HitSound*: Hit sound;
  - *MufHitSound*: Hit sound muffled variation;
  - *bMuffledHit*: Enable hit sound muffling;
  - *bInterruptOtherSnd*: Interrupt the play of other sounds when the hit sound plays;
  - *FlashAmount*: Flash effect amount;
  - *bMuffledFlash*: Enable flash muffling;
  - *FlashScale*: Flash effect scale;
  - *bAffectFOV*: Affect player's FOV dynamically;
  - *FOVDistortion*: Amount of player's FOV change;
  - *FOVDistortionType*: Algorithm in the calculation of the player's FOV change;
  - *FOVRiseFactor*: Percentage of the FOV change complete duration to reach its peak;
  - *bMuffledFOV*: Enable player's FOV muffling;
  - *bSpawnFX*: Spawn custom effects through the ***SpawnFX(...)*** function;
  - *bMuffledSpawnFX*: Enable custom effects muffling.

float ***FullTime***

- Effect full time duration.

float ***FullSize***

- Effect full size/radius.

bool ***bAutoLifeSpan***

- Setup the effect lifespan automatically based on the inner effects duration.

Functions:

simulated function SpawnFX(*byte i, vector Loc, rotator Rot****)***

- Custom effect event call, where ***i*** is the index of the ***NukeFX*** origin.
Loc and ***Rot*** are the base location and rotation of the effect respectively.

Class: NWWallFX	**Parents:** Actor > Effects > BulletImpact > UT_WallHit > UT_HeavyWallHitEffect

Description:

This is the BSP debris generator main class.

Properties: This class has no properties of its own which worth to be mentioned in this document.

Functions:

simulated function InitPlayFX()
- Event called to generate the BSP debris.

Class: NWWallFrag	**Parents:** Actor > Projectile

Description:

This is the BSP debris main class, which is generally spawned from an instance of *NWWallFX* class.

Properties:

float ***InitMinForce***
- Initial debris min speed.

float ***InitMaxForce***
- Initial debris max speed.

float ***MinDrawScale***
- Debris min rendering scale.

float ***MaxDrawScale***
- Debris max rendering scale.

float ***LifeSpanVariation***
- Extra randomized lifespan.

float ***MaxCoveredDebriRadius***
- Max radius which the debris cover visually on any rotation (for collision purposes).

int ***WaterFXDif***
- Water effect type difference.

bool ***bAlwaysHeavy***
- Make debris always “heavy”.

Functions:

simulated function CalcVelocity(*vector Momentum, float ExplosionSize****)***

- Setup speed based on the passed ***Momentum*** and ***ExplosionSize***.

Class: NWMainModMenuInfo	**Parents:** Actor > Info

Description:

This is the menu window info main class, where through simple properties alone one can setup its size, position and basic resize controling.

Properties:

bool ***bUniqueMainMenu***

- Only one menu of this class can be visible at a time.

bool ***bSizableMainMenuW***

- Can be resized along its width.

bool ***bSizableMainMenuH***

- Can be resized along its height.

bool ***bCenterMainMenu***

- Initial position at the center of the screen.

string ***MainMenuCaption***

- Text to appear as an entry of another menu (like the Mod menu).

string ***MainMenuHelp***

- Help text to appear when hovering on the entry of this menu.

string ***MainMenuTitle***

- Text to appear as the title of this menu.

bool ***bMainMenuPosPercentageX***

- Read the ***MainMenuPosX*** setting as percentage.

bool ***bMainMenuPosPercentageY***

- Read the ***MainMenuPosY*** setting as percentage.

bool ***bMainMenuSizePercentageW***

- Read the ***MainMenuPosW*** setting as percentage.

bool ***bMainMenuSizePercentageH***
- Read the ***MainMenuPosH*** setting as percentage.

float ***MainMenuPosX***
- Menu X position.

float ***MainMenuPosY***
- Menu Y position.

float ***MainMenuPosW***
- Menu width.

float ***MainMenuPosH***
- Menu height.

float ***MainMenuMinSizeW***
- Menu min allowed width.

float ***MainMenuMinSizeH***
- Menu min allowed height.

Functions: This class has no functions of its own.

Class: NWMenuPageInfo	**Parents:** Actor > Info

Description:

This is the menu page/tab info main class, where through simple properties and simple events one can build an entire menu with all kinds of inputs: text, numeric, color, lists, sliders, etc, without the hassle of having to build it all through hardcoding.

Properties:

class<NWMainModMenuInfo> ***ModMenuInfoClass***

- Menu windows main class this menu page should be loaded into.

string ***PageTitle***

- Menu page title (text that appears in the page tab).

NPageSettings ***SettingsList[****256****]***

- Menu page settings/elements list:
  Description: Caption of the setting;
  HelpTip: Help tip text (the help text that appears when you hover the mouse over a setting);
  Type: The type of the setting, which can be any of the following:
    ST_Checkbox – Check box:
    ST_Input – Normal text;
    ST_IntegerInput – Integer number;
    ST_FloatInput – Float number;
    ST_Slider – Slider;
    ST_Combo – Combo box/list;
    ST_Color – Color picker;
    ST_Profile – Profile load/save;
    ST_Label – Text label;

  MaxChars: Max number of characters in case of a text or numeric input;
  BottomMargin: Number of pixels of space before the next setting;
  MinSliderVal: Min slider value (only used when *Type=ST_Slider*);
  MaxSliderVal: Max slider value (only used when *Type=ST_Slider*);
  SliderStep: Slider step (only used when *Type=ST_Slider*);
  SliderSize: Slider size (only used when *Type=ST_Slider*);
  SliderTrackSize: Slider track size (only used when *Type=ST_Slider*);
  SliderTrackSize: Slider track size (only used when *Type=ST_Slider*);
  ColorTex: Greyscale icon/texture to be used when *Type=ST_Color*;
  ProfileClass: Profile class to affect when *Type=ST_Profile*;
  hasAdvanced: Has advanced button to open another menu;
  AdvancedText: Advanced button text;
  AdvancedMenuInfoClass: Advanced menu class to open when the advanced button is pressed.

Functions:

static function string GetDefaultValue(*byte i, optional byte advIndex****)***

- Event called during the page load to get the default value (as string) for the setting with the index ***i*** in the ***SettingsList***.
advIndex is set if the current menu windows is the result of an advanced button call, and is therefore the index of the ***SettingsList*** of the previous menu setting it was called from.

static function string ProcessSettingsChange(*PlayerPawn P, byte i, string val, optional byte advIndex****)***

- Event called whenever the setting with the index ***i*** in the ***SettingsList*** is changed in some way through player input from ***P***.
val is the new value of the setting, and ***advIndex*** is set if the current menu windows is the result of an advanced button call, and is therefore the index of the ***SettingsList*** of the previous menu setting it was called from.
The setting changed will set its value with the return value of this function, therefore if is desired to left the changed value intact, ***val*** should be returned.

static function color GetDefaultColor(*byte i, optional byte advIndex****)***

- Event called during the page load to get the default color value for the setting of the type *ST_Color* with the index ***i*** in the ***SettingsList***.
advIndex is set if the current menu windows is the result of an advanced button call, and is therefore the index of the ***SettingsList*** of the previous menu setting it was called from.

static function color ProcessSettingsChangeColor(*PlayerPawn P, byte i, color C, optional byte advIndex****)***

- Event called whenever the setting of the type *ST_Color* with the index ***i*** in the ***SettingsList*** is changed in some way through player input from ***P***.
C is the new value of the setting, and ***advIndex*** is set if the current menu windows is the result of an advanced button call, and is therefore the index of the ***SettingsList*** of the previous menu setting it was called from.
The setting changed will set its value with the return value of this function, therefore if is desired to left the changed value intact, ***C*** should be returned.

static function GetComboDefaultValues(*byte i, out string val1, out string val2, optional byte advIndex****)***

- Event called during the page load to get the default values as ***val1*** and ***val2*** for the setting of the type *ST_Combo* with the index ***i*** in the ***SettingsList***.
val1 is the label of the list entry, and ***val2*** the true value of that entry.
advIndex is set if the current menu windows is the result of an advanced button call, and is therefore the index of the ***SettingsList*** of the previous menu setting it was called from.

static function bool LoadComboList(*byte i, byte listIndex, out string val1, out string val2, optional byte advIndex****)***

- Event called during the page load to load the set of entries for the combo list corresponding to the setting of the type *ST_Combo* with the index ***i*** in the ***SettingsList***.
val1 is the label of the list entry, and ***val2*** the true value of that entry.
advIndex is set if the current menu windows is the result of an advanced button call, and is therefore the index of the ***SettingsList*** of the previous menu setting it was called from.

static function string ProcessSettingsChangeCombo(*PlayerPawn P, byte i, string val2, optional byte advIndex****)***

- Event called whenever the setting of the type *ST_Combo* with the index ***i*** in the ***SettingsList*** is changed in some way through player input from ***P***.
val2 is the new value of the setting, and ***advIndex*** is set if the current menu windows is the result of an advanced button call, and is therefore the index of the ***SettingsList*** of the previous menu setting it was called from.
The setting changed will set its value with the return value of this function, therefore if is desired to left the changed value intact, ***val2*** should be returned.