Advanced FPS Kit
Documentation
Loadout System / Weapon System

This document contains installation information as well as general documentation regarding the formatting of the two XML files used by the system as well as relevant changes needed to the scripts themselves.

Installation

Installing the two systems is very straight-forward. You will have a game folder, inside there is an art folder and a game folder. For most cases (if you're using the generic T3D MIT directory scheme, the "game" folder will refer to scripts and "art" will refer to "art". Copy the contents into their respective folders.

You will then need to add some new code to the engine (if you haven't done so from the install document) to ConsoleFunctions.cpp that reads:

```
DefineConsoleFunction( getEnvironmentVariable, const char*, (const char
*variableName),,
                "@brief Returns the value of the requested operating system's
environment variable.\n"
                "@param variableName Name of the environment variable\n"
                "@return String value of the requested environment
variable.\n"
                "@ingroup Console") {
    // get requested environment variable value
    const char *result = getenv(variableName);

    // verify successful operation
    if(result) {
       // success, allocate return space and return the result
       char *ret = Con::getReturnBuffer(dStrlen(result) +1);
       dStrcpy(ret, result);
       return ret;
    }
    // fail, environment variable doesn't exist
    return "";
}
```

Next, in the two PlayerLoadouts.cs file there are quite a few global variable definitions you need to read and set. I have documented the important ones below:

Client

- $LoadoutSystem::XMLPath: This must match the path from the base game folder to the added XML folder included with the pack.

- $LoadoutSystem::SaveDataPath: This is where the player's loadouts will save. NOTE: If you own MAP, and have the client account system installed, see my notes near the bottom to use saving via GUID instead.
- $LoadoutSystem::MaxClasses: This is the maximum allotted custom loadouts a player may have at once.
- $LoadoutSystem::GuiPopulated: Do not modify this variable, it is set by the scripts.
- $LoadoutSystem::FirstBook: Same as above.
- $LoadoutSystem::WHTag: This is the path to your "weapon_hud" folder located with the GUI files. (See my notes on the new folder layout below)
- $LoadoutSystem::WHTag_Gun: The hud_guns folder is located inside the weapon_hud folder, this is a reference to that folder.
- $LoadoutDefaults::PrimaryGun: This is the default starting primary weapon item in a player's class, when you start up the system for the first time all primary weapons will be this item.
- $LoadoutDefaults::SecondaryGun: Same as above, but for the secondary weapon.
- $LoadoutDefaults::Equipment: Same as above, but for equipment.

Server

- $LoadoutSystem::MaxClasses: This must match the value of $LoadoutSystem::MaxClasses in the client file.

The Directory scheme of the pack modifies the location of the weapon hud images for each of your individual guns slightly. The standard form is: art/gui/weaponHud/ and inside there are all of the images and a few others. This pack adds a "weapon ammo type" image. The two files associated will be placed in this folder and you will add a hud_guns folder where all of the weapon images will now be stored, the proper setup is provided for you in this pack. Please be sure to review $LoadoutSystem::WHTag and $LoadoutSystem::WHTag_Gun to ensure they are pointing to the correct path.

There are also a few adjustments to the WeaponItem and WeaponImage datablocks to use the new weapon system. You must change all instances of PreviewImage = 'blah.png' in the weapon item datablocks to be PreviewImage = "blah.png", otherwise the weapon hud in the top right corner will not function. For the weapon image datablocks, you need to add a ammoImage field to use the images for the string control, the individual naming convention of the files in the art folder are: bullet_Type_O/U.png so the correct syntax is: ammoImage = "Type"; For example the AssaultRifle bullet icon uses ammoImage = "AssaultRifle"; Here is a last example to show you what an updated Lurker looks like:

```
datablock ItemData(Lurker) {
   // Mission editor category
   category = "Weapon";

   // Hook into Item Weapon class hierarchy. The weapon namespace
   // provides common weapon handling functions in addition to hooks
   // into the inventory system.
   className = "Weapon";

   // Basic Item properties
   shapeFile = "art/shapes/weapons/Lurker/TP_Lurker.DAE";
   mass = 1;
   elasticity = 0.2;
   friction = 0.6;
   emap = true;

   // Dynamic properties defined by the scripts
   PreviewImage = "lurker.png"; // ←--- NOTE THE IMPORTANT CHANGE HERE
   pickUpName = "Lurker rifle";
   description = "Lurker";
   image = LurkerWeaponImage;
   reticle = "crossHair";
};


datablock ShapeBaseImageData(LurkerWeaponImage) {
   // Basic Item properties
   shapeFile = "art/shapes/weapons/Lurker/TP_Lurker.DAE";
   shapeFileFP = "art/shapes/weapons/Lurker/FP_Lurker.DAE";
   emap = true;

   imageAnimPrefix = "Rifle";
   imageAnimPrefixFP = "Rifle";

   // Specify mount point & offset for 3rd person, and eye offset
   // for first person rendering.
   mountPoint = 0;
   firstPerson = true;
   useEyeNode = true;
   animateOnServer = true;

   // When firing from a point offset from the eye, muzzle correction
   // will adjust the muzzle vector to point to the eye LOS point.
   // Since this weapon doesn't actually fire from the muzzle point,
   // we need to turn this off.
   correctMuzzleVector = true;

   // Add the WeaponImage namespace as a parent, WeaponImage namespace
   // provides some hooks into the inventory system.
```

```
class = "WeaponImage";
className = "WeaponImage";
ammoImage = "AssaultRifle";
.
.
.
```

To actually use the new weapon system in your game, simply execute the two files named playerLoadouts.cs in their respective folders, I've coded up the file to execute all of the needed other scripts, guis, and even to package in the necessary script overloads it needs to properly function.

You will need to add on your own however:

- A Gui button somewhere to load up the class creator. The command of this button will read: Canvas.pushDialog(ClassCreator);
- Keybinds to the client controls to use the Equipment slot: moveMap.bind(keyboard, e, useEquipment); and to call up the in game class selector: moveMap.bind(keyboard, m, callLoadoutGUI);

XML Documentation

There are two .xml files included in this pack. ItemRequirements.xml and ClientLoadoutSelector.xml, both of these files directly control the class creator system and the GUI.

ItemRequirements.xml is based in the client-side function function checkItemUse(%item); and it validates if a client is allowed to use a certain item selection. You will modify checkItemUse to add individual cases to the system on your own, however a sample is provided (but is commented) for you to build from. The actual XML file itself has this format:

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<LOS>
  <Item name="Lurker">none</Item>
  <Item name="Ryder">none</Item>
  <Item name="ProxMine">none</Item>
  <Item name="Sentry">none</Item>
</LOS>
```

The format itself is very self-explanatory. The name attribute is the WeaponItem datablock name, and the element field is the matching case in the switch() statement located in checkItemUse.

ClientLoadoutSelector.xml is the file that actually populates the class creator gui. You can define individual weapon groups and item sections in this xml file. It is important to pay attention to the naming convention here or it won't work.

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<LOS>
  <Class category="PrimaryWeapons" IGN="Primary Weapons">
    <SubCategory category="AssaultRifles" IGN="Assault Rifles">
      <Item IGN="Lurker Assault Rifle">Lurker</Item>
    </SubCategory>
  </Class>
  <Class category="SecondaryWeapons" IGN="Secondary Weapons">
    <SubCategory category="Pistols" IGN="Pistols">
      <Item IGN="Ryder">Ryder</Item>
    </SubCategory>
  </Class>
  <Class category="Equipment" IGN="Equipment">
    <Item IGN="Proximity Mine">ProxMine</Item>
    <Item IGN="Deployable Sentry">DeployableTurret</Item>
  </Class>
</LOS>
```

<Class> defines the top group in the GUI, and individual weapon categories will fit in the class. The attribute category is used directly in script, and while not too important to keep track of, can help you along if you choose to modify the gui in any way. The IGN attribute is the display name of the tab in the creator gui.

<SubCategory> is an individual category of items located in the <Class>, it follows the same scheme of attributes as <Class> so name them accordingly. No two elements may share the same category tag, however IGN may match if you please.

<Item> is the individual items within a <SubCategory> within a <Class>, IGN is the display name on the buttons in the GUI and can be whatever you please. The element of the <Item> must match the item name, and for the image of the item, the image name must also match this Item name, for example: Item: Lurker must have Image: Lurker.png and so on.

Additional documentation regarding this file is available at the top of the file if you need it.

MAP Owners Read

If you own the Multiplayer Assembly Package, you can adjust your save path on the weapon loadout saver to use GUID based locales, in which case an individual person on one computer can have more than one save file (if they have multiple game accounts). To make use of this, delete the global variable $LoadoutSystem::SaveDataPath at the top of the clientside playerLoadouts.cs and make the following changes in the file:

Find function ClassCreator::saveLoadout(%this) and make this change:

Remove the definition of %file and instead use the following two lines:

```
%playerGUID = $ConnStore::guid;
%file = GetUserDataPath() @ "saveData/"@%playerGUID@"/MultiLoadouts.xml";
```

And then, near the bottom in function loadPlayerLoadouts(), remove the %file line and instead replace it with these two lines:

```
%playerGUID = $ConnStore::guid;
%file = GetUserDataPath() @ "saveData/"@%playerGUID@"/MultiLoadouts.xml";
```

This will adjust the save path of player loadouts to include the player's GUID. If you have an older version of MAP (the one that uses MainMenuGui and MainMenuALGui) you will need to change MainMenuGui to MainMenuALGui in the package to prevent the call from loadPlayerLoadouts() from receiving a blank GUID. If you have the newer version of MAP, this should not be a problem.

```
package afpsk_UpdatedSystem_Client {
  .
  .
  .
  function MainMenuGui::onWake(%this) {
    Parent::onWake(%this);
    loadPlayerLoadouts();
  }
  .
};
```