# Chapter 6: Pawn Setup

## Contents



Copyright Michael Scott Prinke 2011

## Developing Original Content/Mechanics

For the following tutorials we're going to focus our efforts on developing original content and mechanics rather than depending on those put in place by the Unreal Tournament scripts provided to us. By the end of this section of the tutorials, you will produce a side-scrolling platformer with an original pawn with an original model, original movement mechanics, an original weapon, and original aiming mechanics. Once we understand how to implement these basic features it's no stretch from what we've already learned in manipulating our gametype to implement more advanced systems.

## Gametype Setup

Before we can do anything, however, we need to set up a custom gametype. Just as you did in the last section, set up a new solution and a new folder for your scripts in the Development/Src directory and alter UDKEngine.ini to account for it. Then, create a new Gametype script, but instead of extending it off of UTGame you should extend it off GameInfo. As I've said before, where UTGame contains all the information and rules necessary to create an Unreal Tournament gametype, GameInfo is as bare-bones as a gametype class can get. We'll be leaving it mostly blank for this tutorial, but this is where we're going to designate our default pawn class later on.

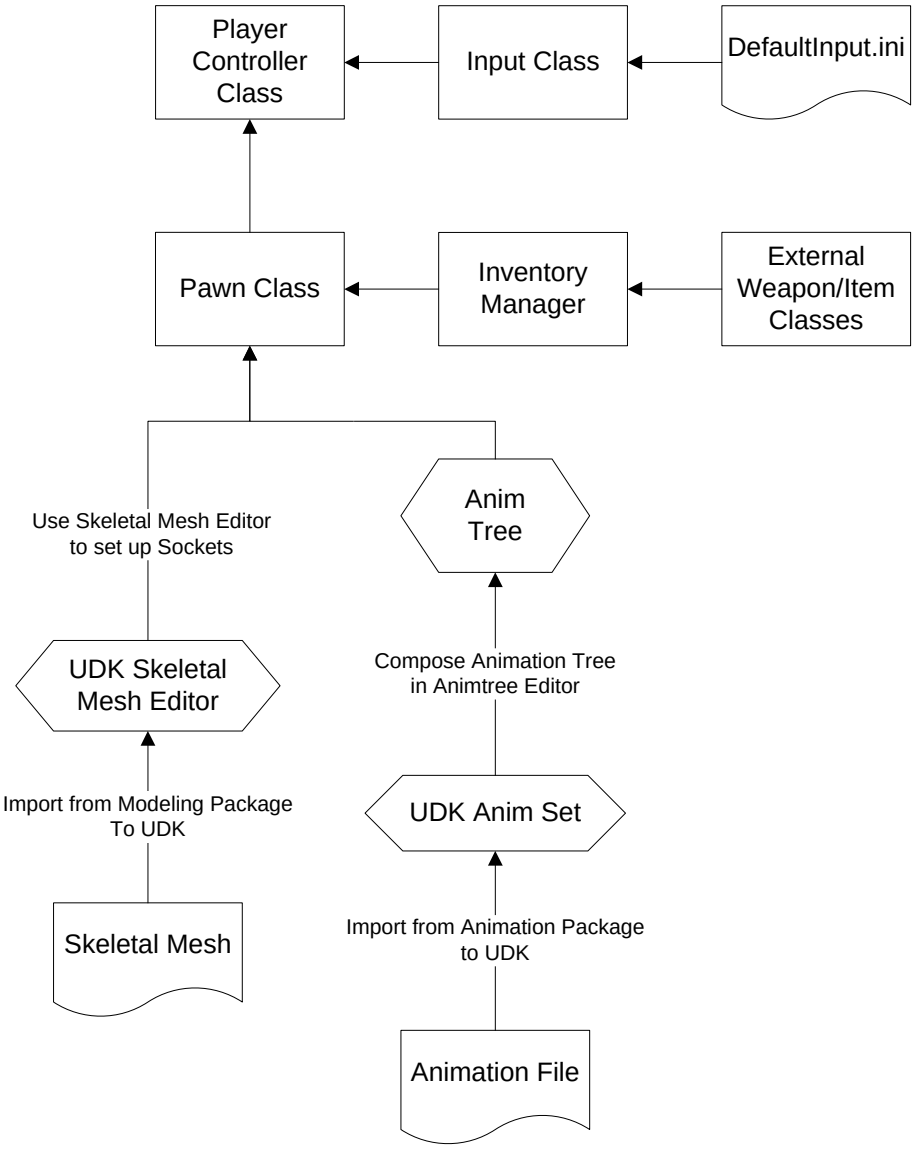
## Replacing the Pawn Model

Nothing says "lazy designer" like the Unreal Tournament bot standing in the middle of a whimsical cell-shaded forest. It's a functional pawn, it has all the support in the world for first and third-person shooting, but unless you happen to be Cliff Blezinsky the chances are pretty high that it's inappropriate for any project that you have in mind. Therefore, the first task we're going to take on is replacing this with something more appropriate to your game. We're only going to concern ourselves with getting the model into UDK and the code necessary to replace the UT pawn with our pawn for right now. The next chapter will be devoted to Unreal's animation system and complete the presentation.

### Player Package Class Flow

What we would think of as the "player package," AKA the actions and mechanics that are available to the player, is actually divided into several different Unrealscript Classes, not too much unlike the HUD that we created in the last lesson. The practical upshot of this model, combined with Unrealscript's extensibility, is that player packages are modular and one can easily customize the parts individually to suit a project's needs. The problem is that there's a lot of "chicken and the egg" problems that arise from the interdependency of all the different scripts involved in making the simple functions that drive a pawn, IE movement, inventory, animation, et cetera, fully functional.

Copyright Michael Scott Prinke 2011




All of the classes and files that we'll be working with and their interactions with one another are displayed in the flow chart above. They are summarized as follows:

**Player Controller -** The class that acts as the "brain" of the player package, accepting input and telling the player pawn to move. This is where the majority of movement and behavior is controlled. You can think of it as a "puppeteer" class.

**Pawn -** The physical character actor, comprised of a skeletal mesh, animations, sounds, and behaviors under specific physics conditions and states.

**Input Class -** A class that specifically controls how input is routed through the player package, enabling the creation of custom input functions. While nothing stops it from communicating directly with the pawn, it's better to have it just pass input data to the player controller's functions. That way you can have the same input behave differently for any number of different pawns or different vehicles.

Copyright Michael Scott Prinke 2011

**DefaultInput.ini -** The config file that tells UDK what inputs to recognize and what functions in the Input class to route buttons to. Not only does it control what inputs UDK should recognize, but how they should be read. As a simple example, you can make a key on the keyboard simply route to a function on press, or you can have it route to two *different* functions on press *and* release of a key, respectively.

**Inventory Manager -** A separate class that concerns itself only with the storage and management of a player pawn's inventory; IE, what weapons, items, equipment, et cetera that it happens to be carrying, how it switches between them, and information on current values for things like ammunition per individual weapons. This class is assigned in the default properties of the player class.

**Skeletal Mesh -** The model of the character we'll be using. Must be rigged with a skeleton and animated externally. Animation files are saved separately. Exported from a modeling/animation package with the ActorX plugin.

**Animation File -** A separate file exported with the ActorX plugin that stores animation ranges and motions.

**Anim Set -** A reference file in UDK that refers to a fully imported anim set.

**Anim Tree -** A UDK class that controls animations via a kismet-like tree of various nodes. References things like physics states, movement directions, aim, et cetera, blending animations together smoothly and taking most of the heavy lifting out of programming. Because the anim tree automatically handles all of the basics for movement, falling, crouching, and more, we will have minimal scripting to do to actually control animations--though we can access and force anim tree nodes to behave as we wish if we really need to. We will be covering a basic way of overriding the anim tree's control of aiming behaviors as we get into the chapter on animation control.

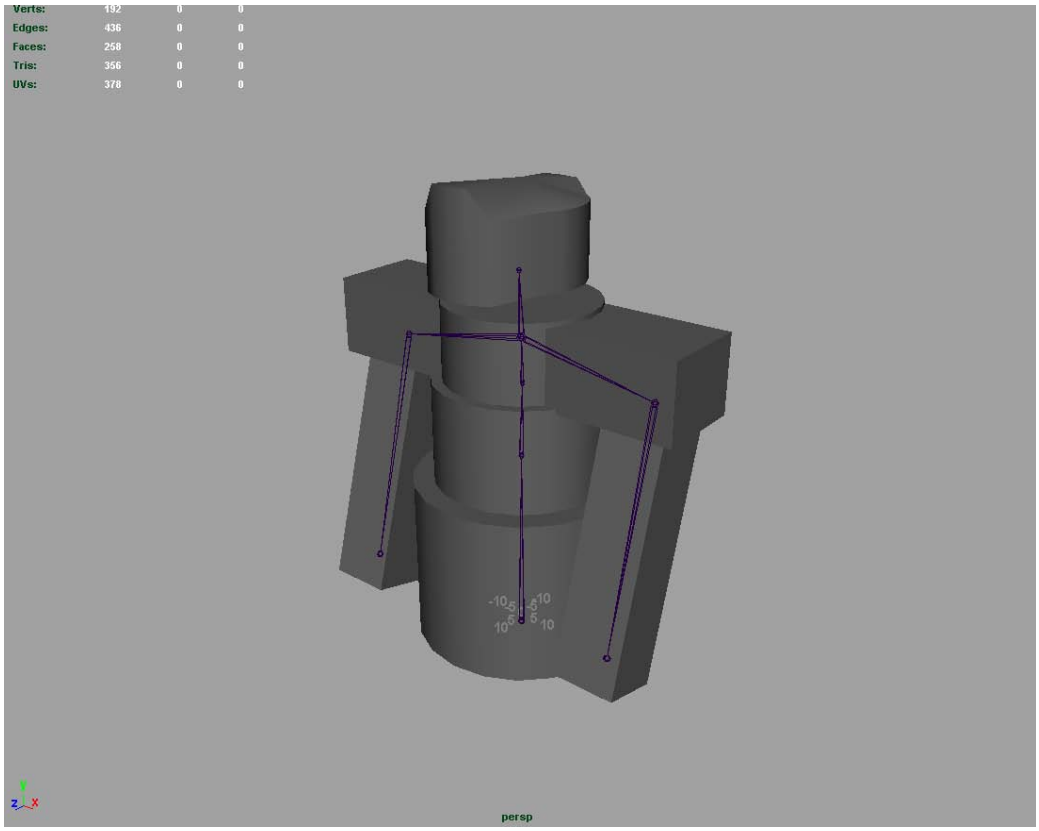
## Importing your Skeletal Mesh

As the topic of these tutorials primarily concerns itself with the processes necessary to get player package assets working with Unrealscript and not modeling or animation themselves, I will not be covering the art asset pipeline for importing a skeletal mesh or an animset in great depth, at least not for right now. More detailed tutorials on that subject can be found at the following pages on UDN:

ActorX Plugin - Exporting and Importing models from Maya and 3DSMax
Animation Import Pipeline

To summarize the process, however, you must first have a skeletal mesh rigged in Maya or Max. It doesn't necessarily need to be a fancy, soft-skinned rig with carefully painted weights; it can be just rigid objects that you've parented underneath appropriate joints.

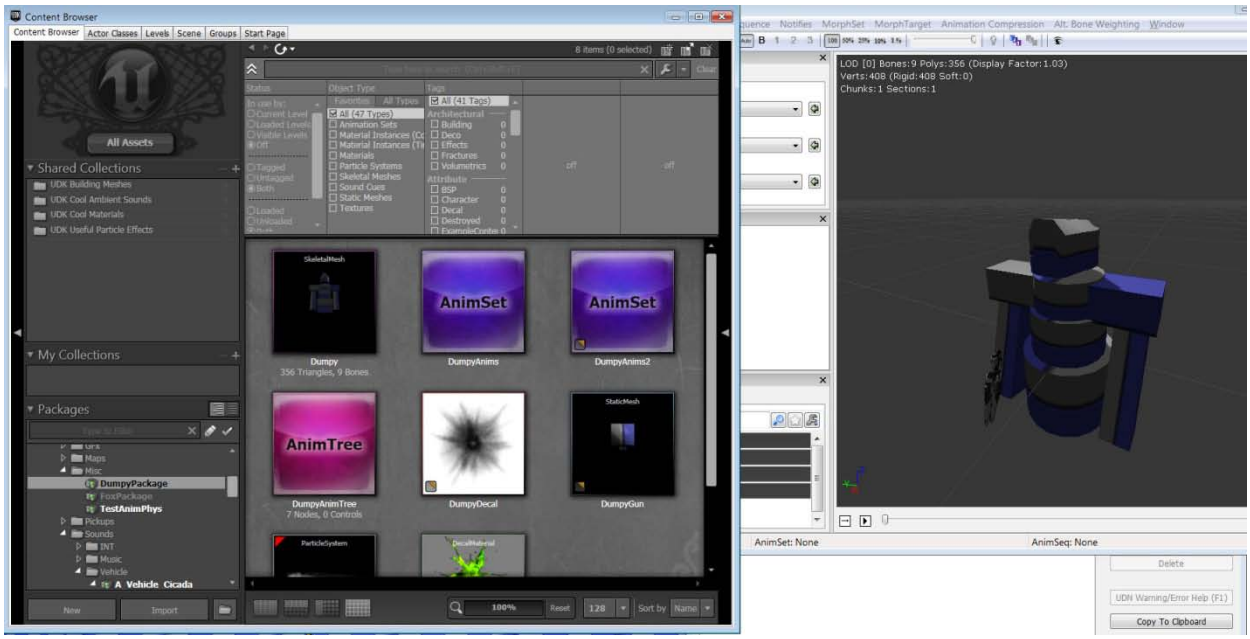
Copyright Michael Scott Prinke 2011




Yes, even a dumpy-looking robot like *this* with not even a remote resemblance to the Unreal Tournament pawn will work. You can make your skeleton and your character's animations look like anything you want, even if all you *are* making is an Unreal Tournament skin. You'll be seeing why in a little while, but as long as the animtree has all its bases covered, equivalent animations for the Unreal Tournament pawn's animations exist, and we set up the pawn's sockets properly so that it can hold weapons in the right places, our pawn can easily do everything that Epic's pawn can do and even play alongside it in a game of deathmatch, with minimal scripting.

But we're getting way ahead of ourselves. Simply use ActorX to export your mesh with "AXmain," and then import it into UDK. As with the UI from before, I advise giving the pawn its own package so that you don't have to load up an entire map's worth of assets whenever you want to get at it.

Copyright Michael Scott Prinke 2011



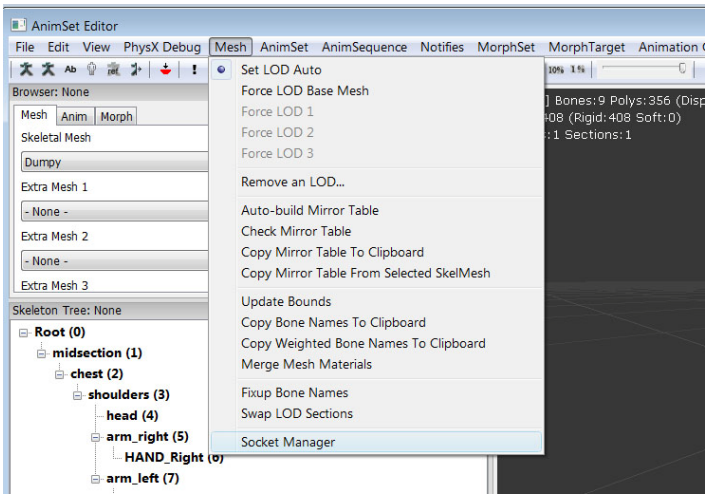

Pictured above: the package and my skeletal mesh, imported and displayed in the skeletal mesh editor.

## Setting up Sockets on your Skeletal Mesh

The step that I *will* be covering is the socket editor, because sockets are vital to gameplay functionality.

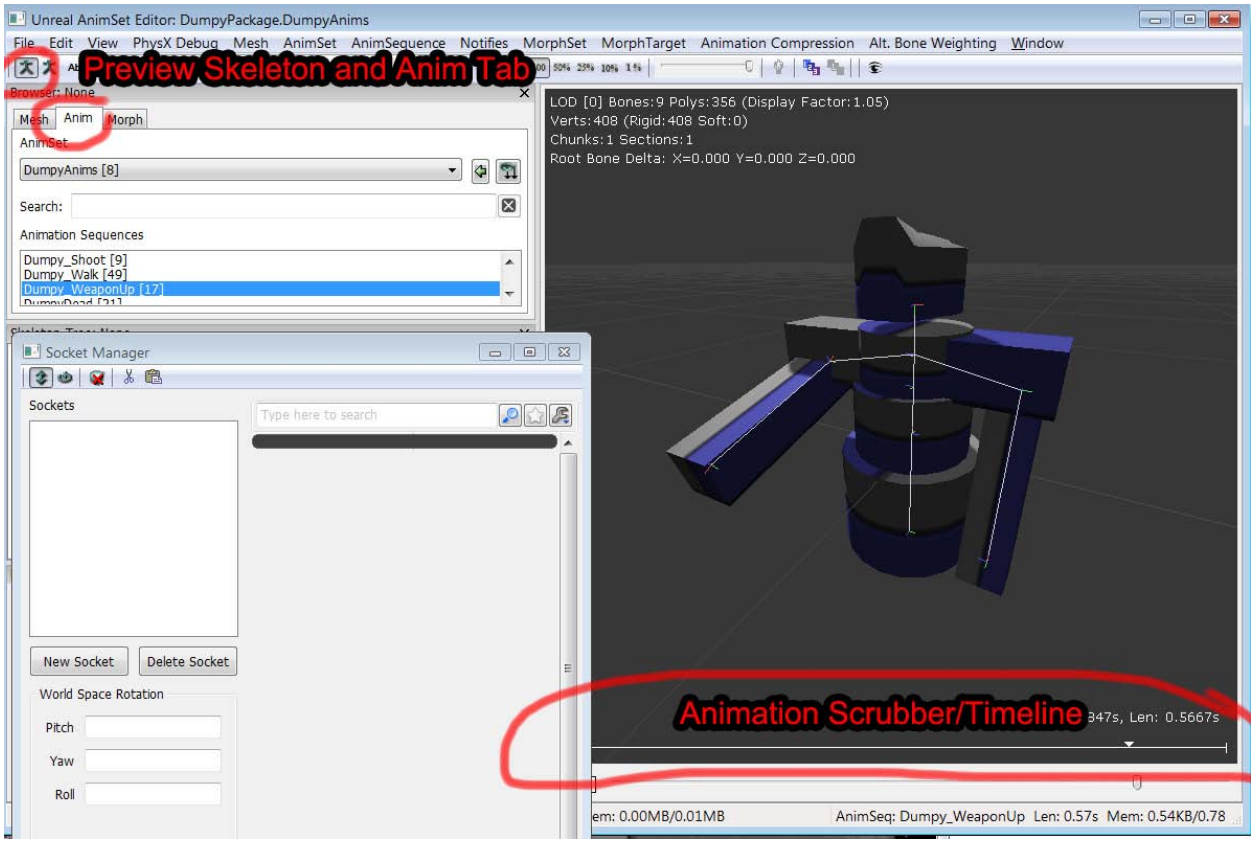
To explain the concept of "sockets" in UDK: ever wonder how Unreal Tournament knows how or where to put guns in a character's hands? Or how a game knows where to put the fumes or exhaust from a jetpack? Or, for that matter, how it knows what end of a gun the bullets come out of? The answer is sockets. These are essentially control points that are bound to a character's skeleton, and you set them up and name them in the skeletal mesh editor. From there, you can reference sockets by name in Unrealscript, telling the game to attach objects and emitters and spawn actors at the points you placed these sockets at.

In the skeletal mesh editor click "Mesh," then scroll all the way to the bottom where it says "socket manager," as displayed below.

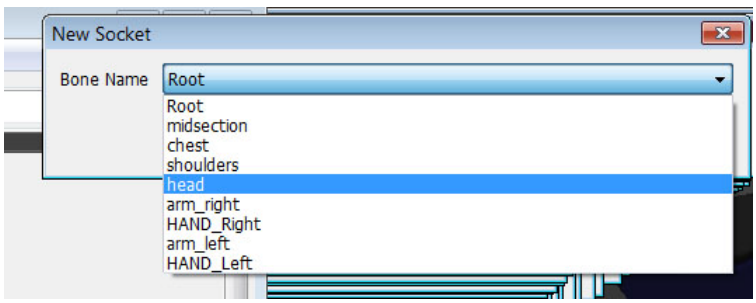



Copyright Michael Scott Prinke 2011

This will open up a window specially designed for the placement of sockets. While you're at it, you'll want to click the stick figure icon in the upper left corner to display your skeleton and go over to the "Anim" tab if you've already got an Animset imported into UDK. I will note that this has **nothing to do** with actually assigning your pawn's Animset to your pawn; that happens either in code or in matinee. This exists **only** for preview purposes, and that's what we're using it for here--previewing what it looks like when the pawn is in an animation for carrying a weapon. You can use the animation scrubber at the bottom of the mesh preview window to show what the animation looks like. Here I've got it midway through an animation where it raises its arm, but anything that gives you a good idea of how it'll actually look for the pawn to have a weapon in its hand will work just fine.

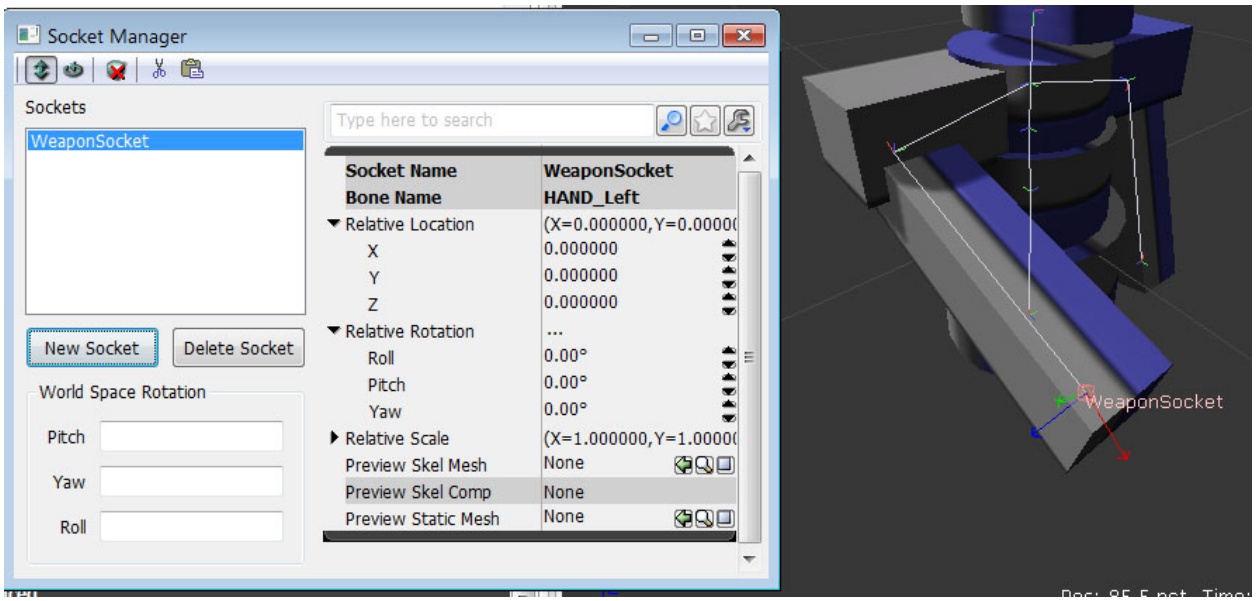



Now, click the "New Socket" button. You'll be asked to pick a bone to assign the socket to. When you select a bone, that socket will be attached to it; in other words, when that bone or any of its parents rotates, the socket will rotate accordingly.

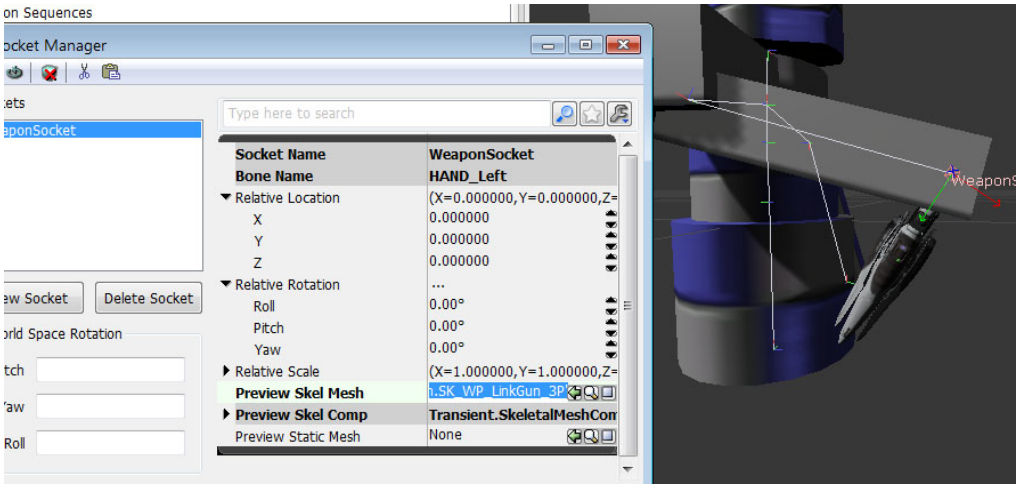



Copyright Michael Scott Prinke 2011

In my case I'm going to pick HAND_Left, a joint at the end of the skeletal mesh's arm. Click "OK," and you'll be asked to name the socket. **Make it something distinct**. Don't use general terms like "Weapon." Intuitive though they may be, it's easy to end up in confusing conflicts with class names. Since "Weapon" is already the name of an Unrealscript class, I'll want to call this something like "WeaponSocket." Click "OK" again, and the socket will be added right on top of the bone, in the form of a pink sphere, and it will appear in the Socket Manager's list.

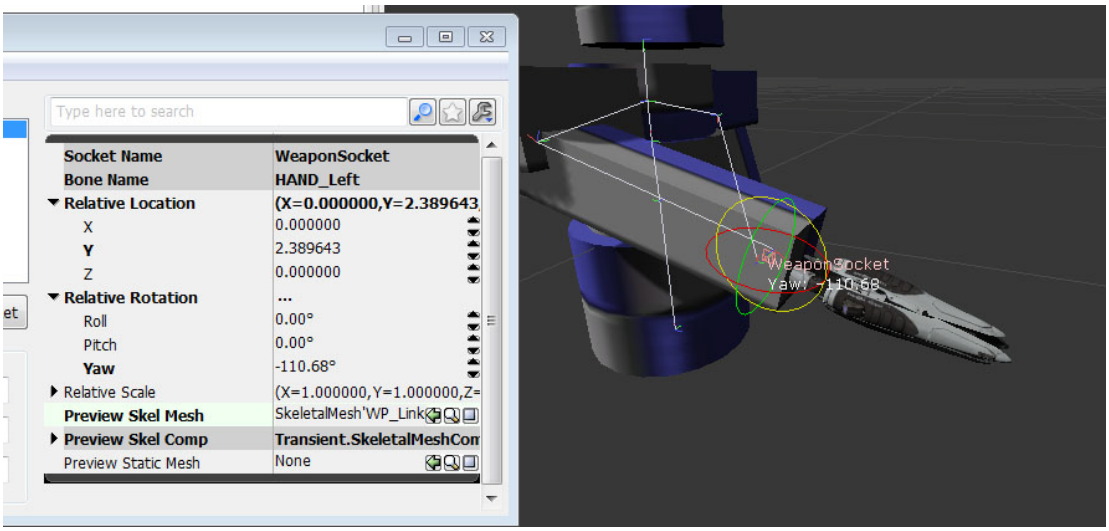


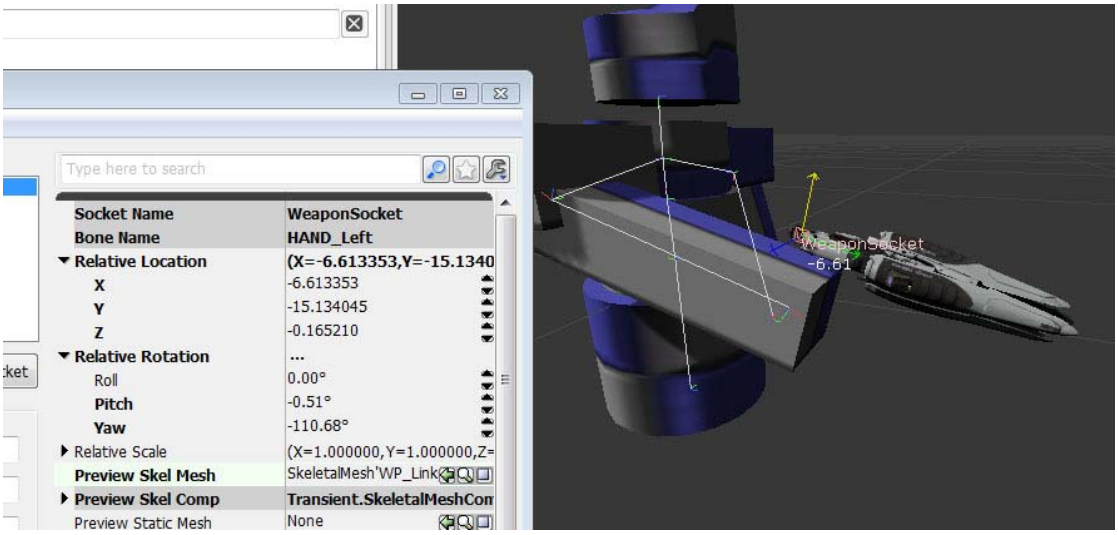

We can now edit the socket. Just as you can within UDK's level editor, you can use the space bar to cycle between a set of gizmos that will let you move and rotate the socket you have selected. However, we have no way of knowing just based on what we see here whether we have our socket oriented properly. Note the field that says "Preview Skel Mesh." Here we can designate a skeletal mesh to stick in the socket, just as we would see it ingame. As a rule of thumb for when you create your own weapons and equipment, note that the skeletal mesh's origin or pivot point will be snapped directly to our socket's center.



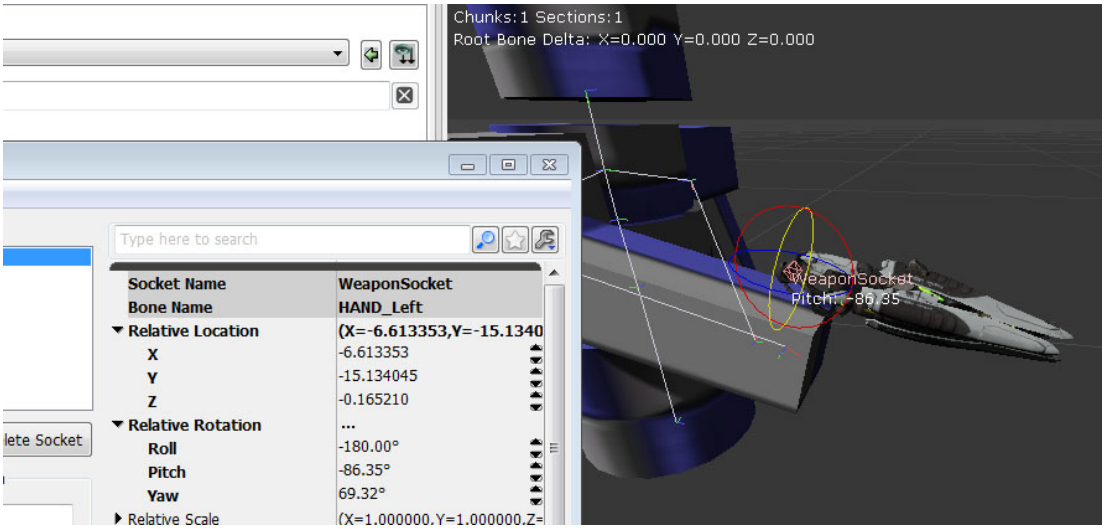

Copyright Michael Scott Prinke 2011

As you can see, the weapon socket is oriented all wrong. It's penetrating through the character's arm and pointing down at the ground; we want it facing forward. So, with a few quick adjustments using the transform gizmos in the skeletal mesh preview window...




Copyright Michael Scott Prinke 2011




... we now have our weapon socket oriented properly. If we want more precise adjustments, we can edit the "Relative Rotation" and "Relative Location" fields manually, of course, but the visual editor makes it easy to quickly get things into place.

## Assigning the Pawn in Unrealscript

With our skeletal mesh imported and our sockets set up, we've covered everything necessary to get our pawn accepting weapons and showing up in code. We have a lot more to do before the pawn is fully functional, but let's get him into the game and get the player controlling it.

We've already created our custom GameInfo class, which I've named PlatformGame.uc. In that class, we'll need to do two things in order to make our pawn playable:

**PlatformGame.uc**

```
DefaultProperties
{
	PlayerControllerClass = class'PlatformerController'

	DefaultPawnClass= class'PlatformerPawn'

	bDelayedStart=false

	Name="PlatformGame"
}
```

The obvious things we need to do are assign our new pawn class and player controller classes (we'll get to actually making them in a moment), but there's a property we have to change as well: bDelayedStart. If you don't set this to false, the player will be stuck in "spectator mode," waiting for the game to "begin" as if it were a game of Unreal Tournament and the pre-game countdown hasn't ended yet. By setting bDelayedStart to false, we bypass this and spawn the player right away.

Copyright Michael Scott Prinke 2011

**PlatformerPawn.uc**

Next, let's code together our pawn. In my case I'm going to extend it off of UTPawn, just to get it working quickly.

```
class PlatformerPawn extends UTPawn
	placeable;

DefaultProperties
{
	Begin Object class=SkeletalMeshComponent Name=SkeletalMeshComponent0
		SkeletalMesh=SkeletalMesh'DumpyPackage.Dumpy'
		bAcceptsLights=true
		End Object
	Mesh=SkeletalMeshComponent0
	Components.Add(SkeletalMeshComponent0)

	WeaponSocket = PlatformWeapSocket
	WeaponSocket2 = PlatformWeapSocket

}
```

This code in the DefaultProperties section is what will allow us to add the skeletal mesh component, and will override the UTPawn's existing skeletal mesh. Like with other asset references, you have to specify the path to the pawn inside the package you created for it. In this case my package is called "DumpyPackage" and the mesh is simply named "Dumpy." I've also assigned two of UTPawn's sockets to take the names of my own, custom sockets instead of the original socket names. That way, when I pick up a weapon I'll be able to use it.

**PlatformerController.uc**

```
class PlatformerController extends UTPlayerController;

DefaultProperties
{
	bBehindView=true
	bForceBehindView=true
	bMouseControlEnabled=true
}
```

Again, we don't actually need to change much. I'm just telling the controller to force behind view so that we get a third-person camera instead of a first-person camera.

## Conclusion

Save your scripts, boot Unreal up, build a quick custom map, set a player start, and switch to your gametype, and your new model should now be there in place of the Unreal Tournament pawn. It won't have animations, but the weapons should attach to your socket and be operational. They won't point in the right direction, but they'll *fire* in the right direction. We'll have more on why *that* is later on when we change the pawn to side-scrolling behavior. In the meantime, we've got a brand new pawn mesh implemented and have laid a lot of groundwork.

Copyright Michael Scott Prinke 2011