

Chapter 8 - Animtrees and Aim Control

Contents

Introduction	1
Implementing Animations with AnimTrees	1
Bringing it from Preview to Reality	6
Aiming	6
Making Sure UnrealScript can See our Aim Node.....	8
Aim Support in UnrealScript.....	8
Aim Input.....	8
Aim Animation	9
Weapon Aim Support.....	11
Conclusion.....	12

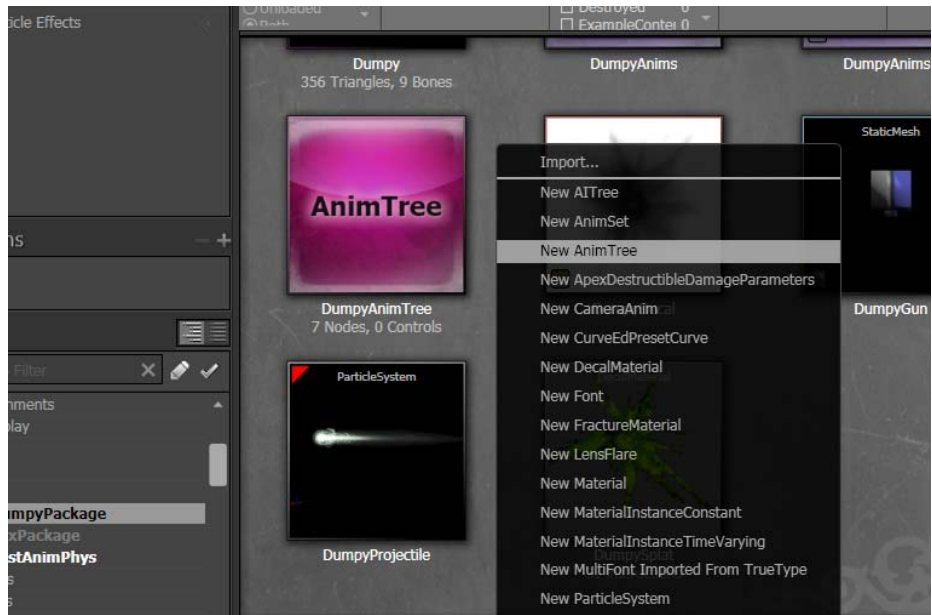
Introduction

In our last chapter we established new pawn controls and behaviors, transforming a first-person shooter effectively into a side-scrolling platformer with just a few changes, mapping our way through the Player Package as we went. Now it's time for us to tackle the animations for movement, while at the same time tackling that tricky aiming problem so that the pawn shoots only the direction we tell it to. Better than that, we're going to add in a 2D aiming system based on mouse control! This will not only serve a valuable purpose in developing our game, it will also give us an idea of how to interact with animations using code and build further on our knowledge base with custom inputs.

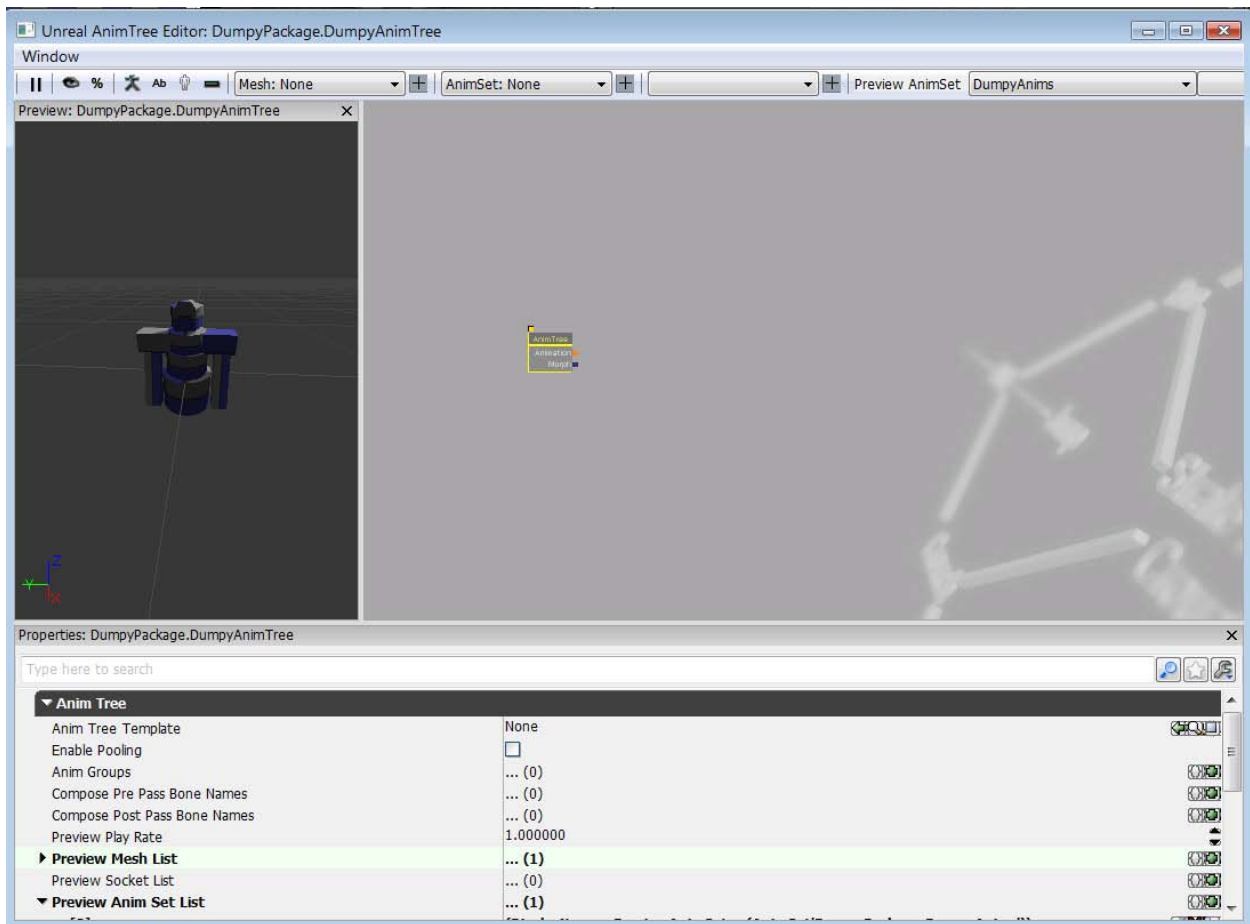
Implementing Animations with AnimTrees

Again, animation isn't the primary concern of this tutorial; it's assumed that you already know how to deal with the ActorX plugin to save out .PSA files to go with your .PSAs and create AnimSets with them. What I'm not assuming is that you know how to get those animations actually working, so we're going to jump straight to the juicy part--the AnimTree editor.

Open up the Content Browser and your Pawn package, then right click and select "New AnimTree." I've named the one for my pawn "DumpyAnimTree," of course.



Then, double-click on it. You'll get a Kismet-like editor that looks like this:



This is the AnimTree Editor, which is what controls the flow and blending of animations for your character. To get it set up, click the AnimTree node that's already there, go to Preview Mesh List and Preview Anim Set List, add one item to both of them, and then load up your character's skeletal mesh and anim set, respectively. Your pawn will appear in the preview window.

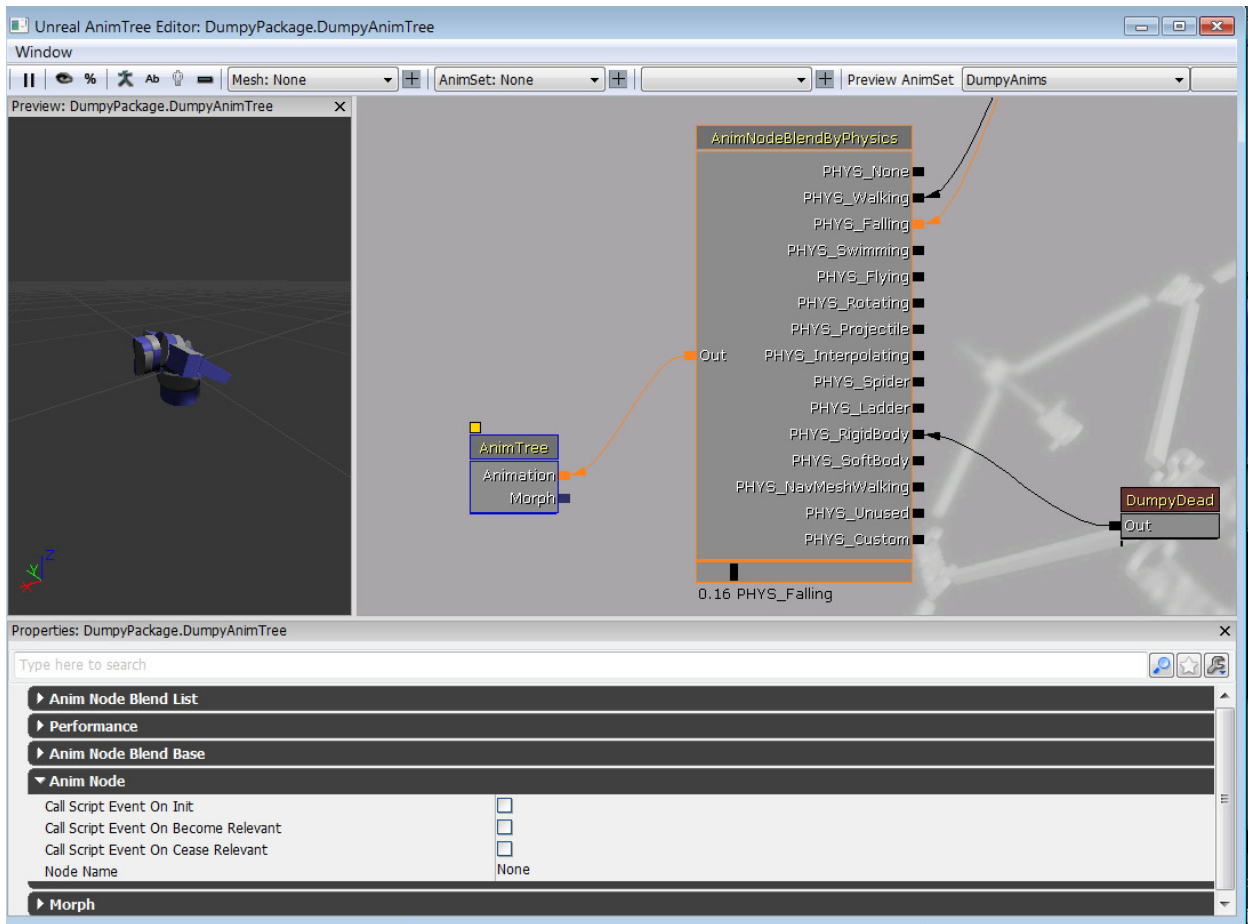
To explain how this setup works, animations "flow" from right to left blending together into one final animation that plays. It's based on nodes that are active, inactive, or blended together. A character can actually be halfway into doing one animation and halfway into doing another animation, depending on the circumstances, or it can play an animation slower based on a particular speed of movement, or a variety of other different things.

The possibilities are practically endless, and there's no way I can cover more than just the few tricks I've employed in my own Pawn class. For the best AnimTree tutorial ever, I recommend visiting UDK Central, where you'll find Wrayth's AnimTree Crash Course--a set of video tutorials that extensively covers AnimTrees, the editor, and even their relationship with UnrealScript. You'll find it here:

[Wrayth's AnimTree Crash Course](#)

In the meantime, I'll show you what I've got my character doing, and explain as best I can how it all works.

Right click, select "New Animation Node," then "New BlendBy Node," and select "AnimationBlendByPhysics." You'll get a big Flow Controller that looks like this:



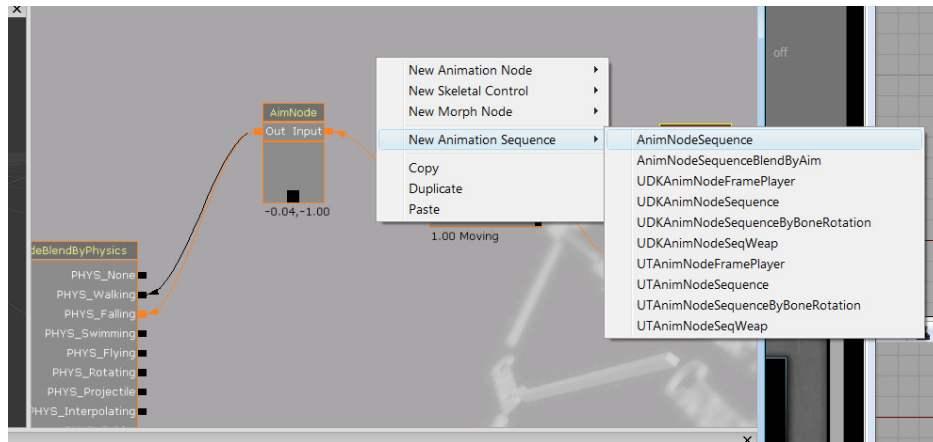
Unrealscript natively communicates with this and many of the other nodes that you see in order to create animations. In other words, it simply reads what physics state your pawn is in--walking, falling, swimming, on a ladder, et cetera--and plays animations from your AnimSet based on that. A scrubber at the bottom of the node lets us control which input we're taking at any given time so that we can preview the individual animations. Note that **physics states are not to be confused with PlayerController states** like the ones we were using before; so don't worry about whether you programmed a falling state or not. It's already in the game and already accounted for, as are walking, swimming, and ladder, which pawns naturally take on when they're on the ground, in a fluid volume, or climbing on a ladder volume, respectively.

In other words, you don't have to code anything at all to get most of your animations to work. Need a jump animation? Plug your Jump animations into PHYS_Falling; the pawn will automatically take them on. Need walking, running, and idling? Plug them into PHYS_Walking. Want to animate what happens when your character dies? Do what I've done here and put your Death Animation through PHYS_Rigidbody, which is what dead pawns take on. It's all done already.

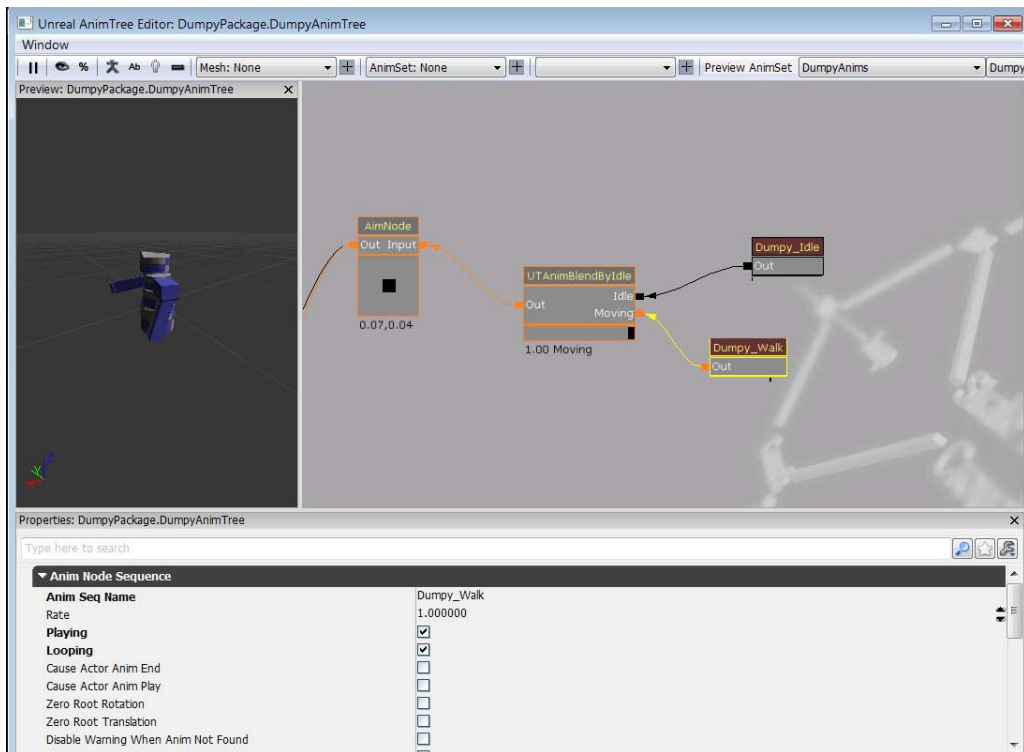
The things you *do* need to worry about coding tend to come down to custom actions. If we want a sword or some other weapon that's not a gun, or if we want a complex ranged weapon like a bow, where we necessarily want to see a character knocking and drawing an arrow, then we have to create

some new logic for how it works. For the purposes of *our* game, though, and for basic actions that are industry standards, *including* double-jumps, UDK has everything mostly figured out.

To add a specific animation from your anim set, right click, go to "New Animation Sequence" and add "AnimNodeSequence." You'll get a blank node.



If you re-name one of these AnimNodeSequences to a name that matches one of your AnimSet's animations--in my case I'm picking "Dumpy_Walk" and "Dumpy_Idle"--then it will automatically load up the animation from the preview Anim Set that we selected earlier.



For animations that you want to see previewed, check "Playing" in the properties at the bottom. For ones you want to loop, check "Looping" and they will loop based on the logic you put into your AnimTree.

Next, create another new node, this time a "BlendBy" node again, and select "UTAnimBlendByIdle." Here I've linked the "Idle" input to my idling animation and my "Moving" input to my Walking animation. I can layer even more nodes on top of this to separate out speed, direction, etc., but this will do fine for my purposes. Next, if you link it back into "PHYS_Walking" and scrub the Physics blender to display it, you should see a preview of your animation in action. Scrub the BlendbyIdle node between Idle and Walking to see the change.

Bringing it from Preview to Reality

Unfortunately, this editor will *only* show you a preview of what it looks like. To actually assign the AnimTree to the pawn, we have to open up the pawn's script once again.

PlatformPawn.uc

DefaultProperties

```
{
    Begin Object class=SkeletalMeshComponent Name=SkeletalMeshComponent0
        SkeletalMesh=SkeletalMesh 'DumpyPackage.Dumpy'
        bHasPhysicsAssetInstance=True
        bAcceptsLights=true
        AnimSets(0)=AnimSet 'DumpyPackage.DumpyAnims'
        AnimTreeTemplate=AnimTree 'DumpyPackage.DumpyAnimTree'
    End Object
    Mesh=SkeletalMeshComponent0
    Components.Add(SkeletalMeshComponent0)
```

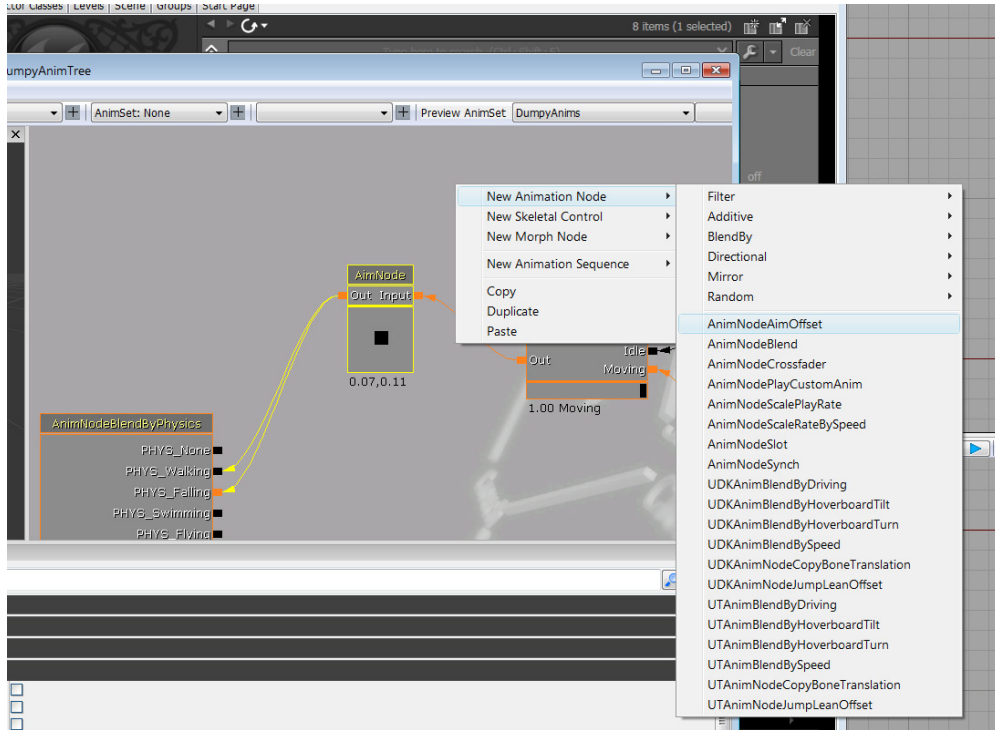
...

This is the Skeletal Mesh component we added to the Pawn before, except we've added in two new lines--one to add an AnimSet I imported called "DumpyAnims" to him, officially assigning animations to the pawn, and one to designate the AnimTree I just made as the AnimTree whose logic the pawn should take on when it starts moving around and taking physics states.

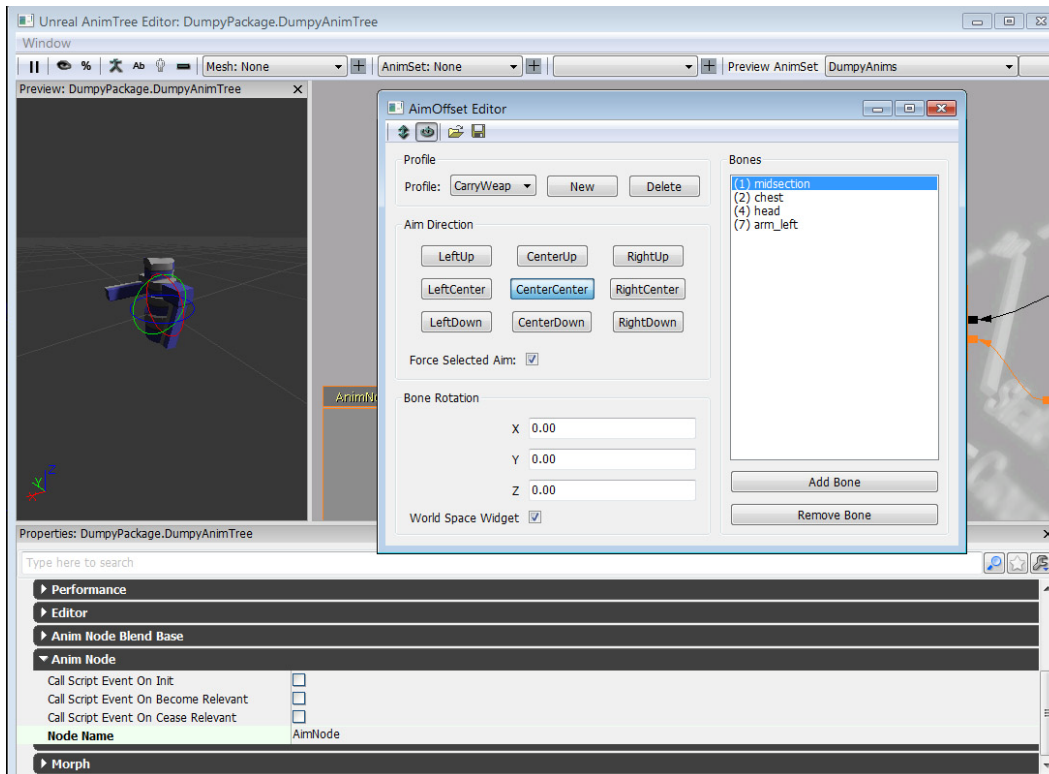
Now when I play my game, I'll have the pawn animating between walking and idling when I move and stop--and I didn't have to do more than write two lines of code to do it. *It really is that simple!*

Aiming

But we're about to make it more complicated! Break the link from your BlendByIdle to PHYS_Walking. Right click, create a new Animation node, and this time go for the first one on the list--AnimNodeAimOffset. Plug this in between BlendByPhysics and BlendByIdle.



This node is very special in that it allows you to control the way that the player character aims, letting it look up, down, left, and right while also playing walking and idling animations seamlessly. Double-click on it, and you'll get a special AimOffset editor.



This editor allows you to set specific poses to bones, manually rotating them up, down, and to the sides for up to nine specific positions: straight forward, left, right, up, down, and everywhere in between. In our case we're going to concern ourselves only with three of these: CenterUp, CenterDown, and CenterCenter. CenterUp is looking straight forward and straight up, CenterDown is straight forward and straight down, and CenterCenter is just looking straight forward. As the "AimNodeOffset" value changes, it will blend between the poses that you set for these positions.

Before you can set any of these, you'll need to create an AimOffset Profile. These Profiles pertain to specific "postures" that the character takes on; crouching, unarmed, one-handed weapons, and two-handed weapons are a few good examples of different kinds of postures you can set for AimOffset Profiles. I just have a general one for carrying a weapon, as this pawn is very simple. When you save it, you'll save it as an external file; I recommend placing it in the "UDKGame/Content/Misc" folder.

Clicking the "Add Bone" button will ask you to choose a bone to add from the skeletal mesh's skeleton. Here I've chosen everything in my pawn's torso plus its arm. It's a little bit of a pain, but I simply selected the CenterUp, CenterCenter, and CenterDown poses, respectively, and manually rotated each of those bones in the preview until I got results I wanted that seemed like they realistically pointed straight up and straight down.

Once you have these set as you like them, you'll be able to return to the AnimTree Editor and move the square in the "AimNodeOffset" node to preview your changes.

Making Sure UnrealScript can See our Aim Node

There's one last change we have to make in order to make sure this sucker works. In the properties at the bottom of the screen, scroll down to "Anim Node," open it up, and change the "Node Name" to "AimNode," just like in the figure pictured above. When we use UnrealScript we're going to be searching for the node by name, and this is the name Unreal looks for by default.

Aim Support in UnrealScript

Aim Input

PlatformerController.uc (PlayerWalking State, PlayerMove Function)

First thing's first. We need to actually *get* mouse input before we can have the pawn respond to it. Because we've overridden practically everything the Pawn can do in the Walking state, it's not taking the mouse to change aim. From looking at the DefaultInput.ini, we can tell that all we need is to get PlayerInput.aMouseY for the up-and-down motion of the mouse.

Inside the "PlayerMove" function of the PlayerWalking state, we'll move down to just below the "jump" process and just above the "NewAccel.Y=0" assignment.

```
PlatformerPawn(Pawn).MouseAimUpdater(PlayerInput.aMouseY);
```

This is a custom function we're about to write; "Pawn" refers to the pawn belonging to this Controller. We're casting it to the type "PlatformerPawn" in order to insure that we can get PlatformerPawn

functions, as we're about to write one. We haven't written "MouseAimUpdater" yet, but that's where we're going to send the up-and-down mouse input.

PlatformerPawn.uc

```
class PlatformerPawn extends UTPawn
placeable;

var float CamOffsetDistance, MouseLookAim;
...

function MouseAimUpdater(float MouseYInput)
{
    MouseLookAim += MouseYInput;
}
```

Now let's write that function. To get useful mouse input we have to do something a little odd. MouseYInput, in this case, doesn't actually refer to a mouse *position*, but a mouse *change* in position. We need to convert that *to* a position, and for that purpose we've made a new variable called MouseLookAim, and we're going to increment the Mouse Input we're getting on top of that. Now any time we move the mouse up or down, that value will change, giving us the value we'll need to produce the rotation we want to set the player's aim to.

Aim Animation

In order to make the aiming animation work, we have to add support for the aiming profile we created. "SetWeapAnimType" is the function where this occurs. Since I only have *one*, mine looks like this:

```
simulated function SetWeapAnimType(EWeapAnimType AnimType){
if (AimNode!=none)
    {
        AimNode.SetActiveProfileByName('CarryWeap');
    }
}
```

We also need to assign the aim node and any other important nodes; thus, we look at the "PostInitAnimTree" function, which happens--as you can probably guess--just after the pawn and its animations are initialized.

```
simulated event PostInitAnimTree(SkeletalMeshComponent SkeletalMeshComp){
if (SkeletalMeshComp==Mesh)
    {
        AimNode=AnimNodeAimOffset(Mesh.FindAnimNode('AimNode'));
    }
}
```

You can find any and all named animation nodes using the "FindAnimNode" function like this, meaning that you aren't tied down to just regular nodes--custom blend nodes that you put together in the Anim Tree Editor can also be accessed this way.

We want the pawn's aiming animation to respond to our input, so we're going to do another addition:

```
function MouseAimUpdater(float MouseYInput)
{
MouseLookAim += MouseYInput;
AimNode.AngleOffset.Y -= (MouseYInput/16000)
}
```

If you recall, "AimNode" is the name that Unrealscript looks for to find a pawn's aiming node, and it separates values into two axes--X and Y. Thus, AimNode.AngleOffset.Y is the value we're playing with in order to make it aim up and down; we are literally accessing and overriding those properties manually. This will be the extent to which we have to code any animation.

Just like with MouseLookAim, we're incrementing it--but it's a value of -1 (straight down, or 90 degrees downward) to 1 (straight up, or 90 degrees upward). What we get from the mouse is more akin to the binary data that we use for rotations, with 16000 being 90 degrees downward and -16000 being 90 degrees upward, roughly. By dividing this value by 16000, we produce something on the scale that our anim node can deal with.

Now we need to put some control in here, because we don't want the pawn to try to aim back over its head. So, we're going to add in some simple flow control. We don't want the MouseLookAim to update *unless* it's somewhere between 90 and -90 degrees, so we add this:

```
if (MouseLookAim < 16000 && MouseLookAim > -16000)
{
    MouseLookAim+=MouseYInput;
    AimNode.AngleOffset.Y -= (MouseYInput/16000);
}
```

Now the MouseLookAim variable and our pawn will only try to rotate *between* a straight up and straight down position. If we test this, though, it *will* cause the pawn to lock in place when it hits either of those extremes and never move again. So, let's add a couple of additional clauses...

```
if (MouseLookAim < 16000 && MouseLookAim > -16000)
{
    MouseLookAim+=MouseYInput;
    AimNode.AngleOffset.Y -= (MouseYInput/16000);
}
else
    if (MouseLookAim >= 16000 && MouseYInput <0)
    {
        MouseLookAim+=MouseYInput;
        AimNode.AngleOffset.Y -= (MouseYInput/16000);
    }
else
    if (MouseLookAim <=-16000 && MouseYInput >0)
    {
        MouseLookAim+=MouseYInput;
        AimNode.AngleOffset.Y -= (MouseYInput/16000);
    }
```

These extra if/else statements enable us to override it by saying that, if we somehow pass our aiming threshold--or meet it--we can keep moving the player's aim *as long as we're moving in the opposite*

direction of our threshold. So, if we're locked straight up, as long as the input we get is negative instead of positive, we can still move; and vice-versa.

Weapon Aim Support

Finally, we're going to fix the pawn's aiming rotation so that its weapon *only* fires forward. The function responsible for this is called "GetBaseAimRotation()." It's one of the default Pawn's more straightforward functions, fairly easy to understand--and we're about to make it easier.

```
simulated singular event Rotator GetBaseAimRotation()
{
    local rotator    POVRot, tempRot;

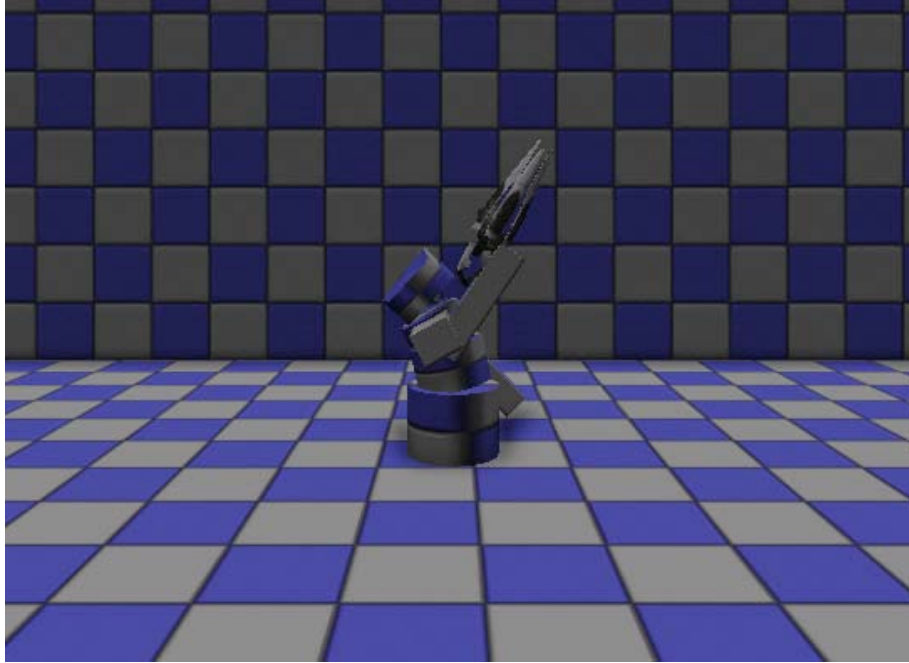
    tempRot = Rotation;
    tempRot.Pitch = 0;
    SetRotation(tempRot);
    POVRot = Rotation;
    POVRot.Pitch = MouseLookAim;

    return POVRot;
}
```

To explain what we're doing...

tempRot is the rotation that the pawn will physically be taking on. We don't want it to be able to rotate up and down, so we've locked its Pitch to 0. Otherwise, the Controller forces the pawn to take rotations for us, so we hardly need to worry about it and just use "SetRotation" to make it keep the rotation we gave it before. Probably we can take some information out of this step, but it's important to be sure behaviors work the way we want them to at this early stage.

POVRot is the rotation that our character's weapon will take on when we try to aim it. Normally acquiring the value for POVRot entails a lot of raycasting, but we're lopping a lot out of the equation by simply setting it to the pawn's rotation value. In other words, it will **always point straight forward**, the way that the pawn is pointing... with one exception, that being that we set its Pitch value--IE, its rotation around the pawn's X-axis, or up-and-down rotation--to the "MouseLookAim" value that our MouseAimUpdater function is giving us. "MouseAimUpdater" already handles most of the work for us, so we can consider our work done! Build the project and start up the game and, like magic, your pawn will now aim based on your mouse input.



Conclusion

We've now overridden both the pawn's aim and the pawn's animation with script. From here it's no small stretch to try some even crazier things. We can easily extend this from mouse control to dual analog control, for instance, and use the up-and-down axis of the left analog stick (`aForward`) to control aim instead of the mouse. The possibilities are limited only by your imagination and design knowhow.

We have one last important topic to cover, though--namely, the generation of interactive ingame assets. Specifically, I'm talking about weapons. The Link Gun is all well and good, but Unreal weapons have a great deal of overhead--and for a Contra-style platformer, we need something a bit simpler and more stylistically appropriate to our game. Plus, this will be a good opportunity for us to start figuring out how to dissociate our classes from Unreal Tournament classes and have more original content. The next chapter, then, will be devoted to custom weapons and inventory.