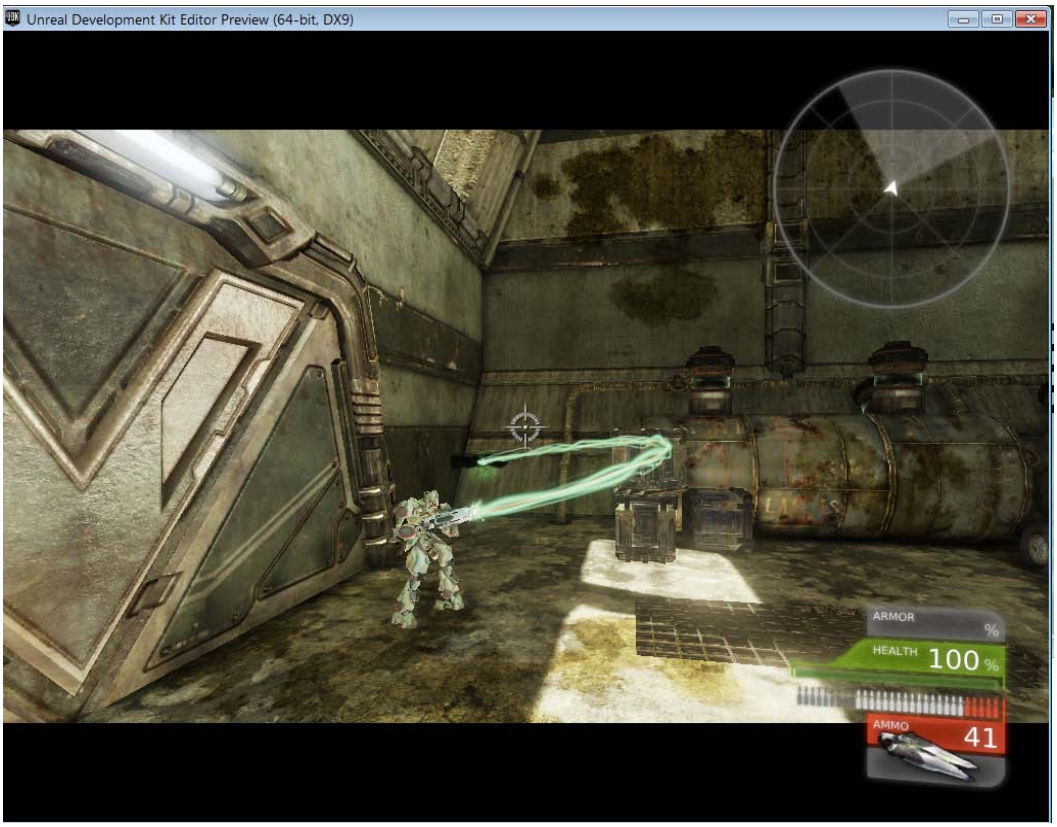
Chapter 7: Overriding Pawn Behavior

Contents

Introduction

In our last chapter we laid out the groundwork for replacing the Unreal Tournament pawn with our own, original Pawn. We replaced the pawn model, but it still *acts* like the Unreal Tournament pawn, which is a problem when we're trying to do a side-scrolling platformer.

We *can* implement a lot of quick fixes in Kismet if we so choose. We can set an external camera instead of the player's default camera, attach it to the player and the like, and a lot of small projects do take shortcuts like that. In fact, UDK Mobile employs this as its primary means of developing camera control, as it has kismet controls for things like player input. It's not that unreasonable an idea, but in UDK standard--the console and PC-oriented version of the program--these things don't function. The input nodes only pertain to Mobile input. Meanwhile hen we try to override the camera we get problems like this:



No matter what you do in Kismet, the player's gun will fire at where the camera's pointing instead of where the character is pointing. This is a standard first or third-person shooter cheat. You may *think* developers put lots of time and effort into carefully pointing gun models in such a way that they line up with where the camera's facing, but really what happens is a lot simpler.

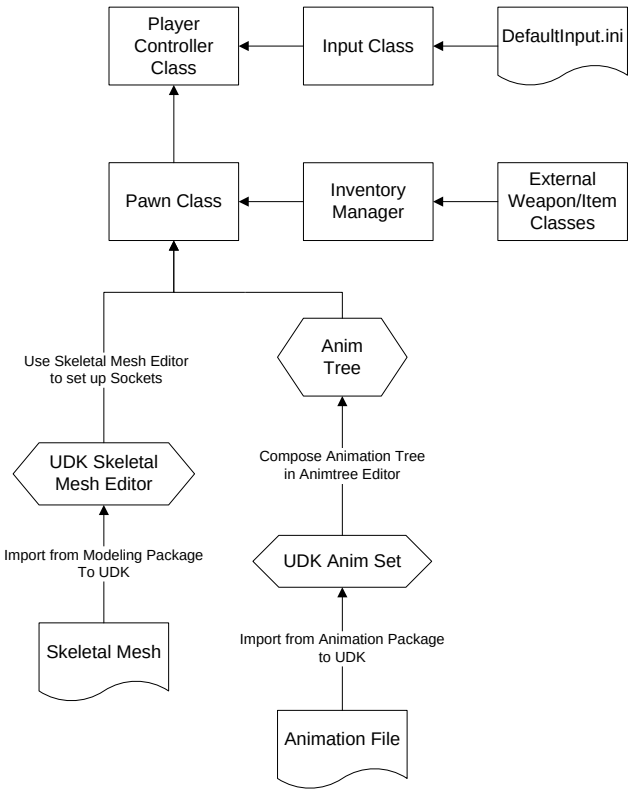
The projectile *does* fire from the end of the gun, but the gun model, like the player character, is a skeletal mesh with sockets set up showing where muzzle flashes and projectiles are supposed to emit

from. These joints/sockets can be rotated independently of where the gun is actually pointing, hence why a shot can travel in such drastically different directions. The camera, meanwhile, does a raycast to find out where the player is aiming. In other words, it draws a line from the middle of the camera forward until it hits something, and the point where that line ends is what the projectile spawner actually points at. It's simple, and it works. But, it's also *hugely* inconvenient if we're not making a shooter. Not only that, but again, we're stuck with default inputs unless we re-code them ourselves.

In this chapter we're going to override all of this, creating a new, side-scrolling camera system, fixing the player's aim so that it fires straight forward, and locking the player on a side-scrolling plane instead of maneuvering in 3D space by overriding its input.

Overriding Camera Behavior

We're going to start by changing the camera system, which is deceptively simple. Remember that flowchart from the last chapter?



You'll notice that nowhere here do I specify the camera class. While the camera *is* a part of this family tree, technically, there is no reason that we ever have to change the camera class, except if we want to change something about way the camera *displays*. If we want to make an isometric camera, add a depth of field, change the FOV angle, or anything else pertaining to the camera settings, then we'd write a new camera class. The camera class itself, however, **does not have anything to do** with how the player controls the camera. That's all actually built into the **pawn** class. So, without further adieu, let's open it up.

Pawn.uc

A single function controls how the camera works, called "CalcCamera."

```
simulated function bool CalcCamera( float fDeltaTime, out vector out_CamLoc,
out rotator out_CamRot, out float out_FOV )
{
}
```

As you can see it's a fairly hefty function that takes in a lot of parameters. The parameters we really need to pay attention to are out_CamLoc and out_CamRot, which are a vector and a rotator (basically the same as a vector, but for storing a rotation value, specifically) controlling the camera's position and rotation, respectively. Ordinarily they're set in the player's chest or behind the player's shoulder in first- and third-person perspective, respectively, but we're going to get a little bit more specific.

```
simulated function bool CalcCamera( float fDeltaTime, out vector out_CamLoc,
out rotator out_CamRot, out float out_FOV )
{
   out_CamLoc = Location;
   out_CamLoc.Y += CamOffsetY;
   out_CamLoc.Z += CamOffsetZ;
}
```

Here I'm setting out_CamLoc to "Location," which refers to the pawn's location. I then add to the Y and Z components, respectively, shifting it off to the side where we can view the pawn (Y component), and moving it slightly up so that the pawn isn't sitting dead-center on the screen. Naturally we're going to need to declare these variables at the top:

```
class PlatformerPawn extends UTPawn
      placeable;

var float CamOffsetY;
var float CamOffsetZ;
```

... and define them in DefaultProperties:

```
DefaultProperties
{
...
      CamOffsetY = 600
      CamOffsetZ = 128
...
}
```

These values are basically arbitrary; it's up to you to pick ones that work for your purposes. For the record, though, I chose 128 units upward because that's roughly the pawn's height; moved up from the pelvis, that puts the camera just a little bit over the pawn's head, keeping its feet closer to the bottom of the screen.

This is all well and good, but if we try to play this in its current state we're going to be looking at nothing and the pawn will be offscreen--it's still pointing forward instead of *at* the pawn. So, we have to

manually adjust that rotation value. Ordinarily out_CamRot would probably be set to "Rotation," referring to the pawn's own rotation, but we want to have a little bit more control over it than that since this is a side-scroller. All we have to do is turn it 90 degrees to the left from where we've positioned it--or -90 degrees.

On Unrealscript Rotation

It's worth noting that Unrealscript does not handle actor rotation the way that Kismet does or the way that Unreal displays rotations in the editor, which is in degrees. It doesn't display them in any recognizable radians, either. Like with position, it has its own units--but they aren't arbitrary. Rather, they're based on raw *machine code* values for rotation, or something close to it; the exact details aren't important. What's important is that the numbers you put in aren't how you'd normally see rotation.

It's also not handled the same way position and scaling are in terms of naming conventions. Where we work with Location.X and Location.Y, we don't deal with Rotation.X or Rotation.Y. Those properties don't actually exist in the "Rotator" class. Rather than X, Y, and Z, we deal with Pitch, Yaw, and Roll--which, incidentally, correspond to X, Y, and Z in terms of the axis they're used to rotate an object in. Just visualize it as a spindle, pointing through the corresponding axis.

So, when we want to override our camera calculation to rotate the camera -90 degrees around the Y axis, what we *actually* have to do is:

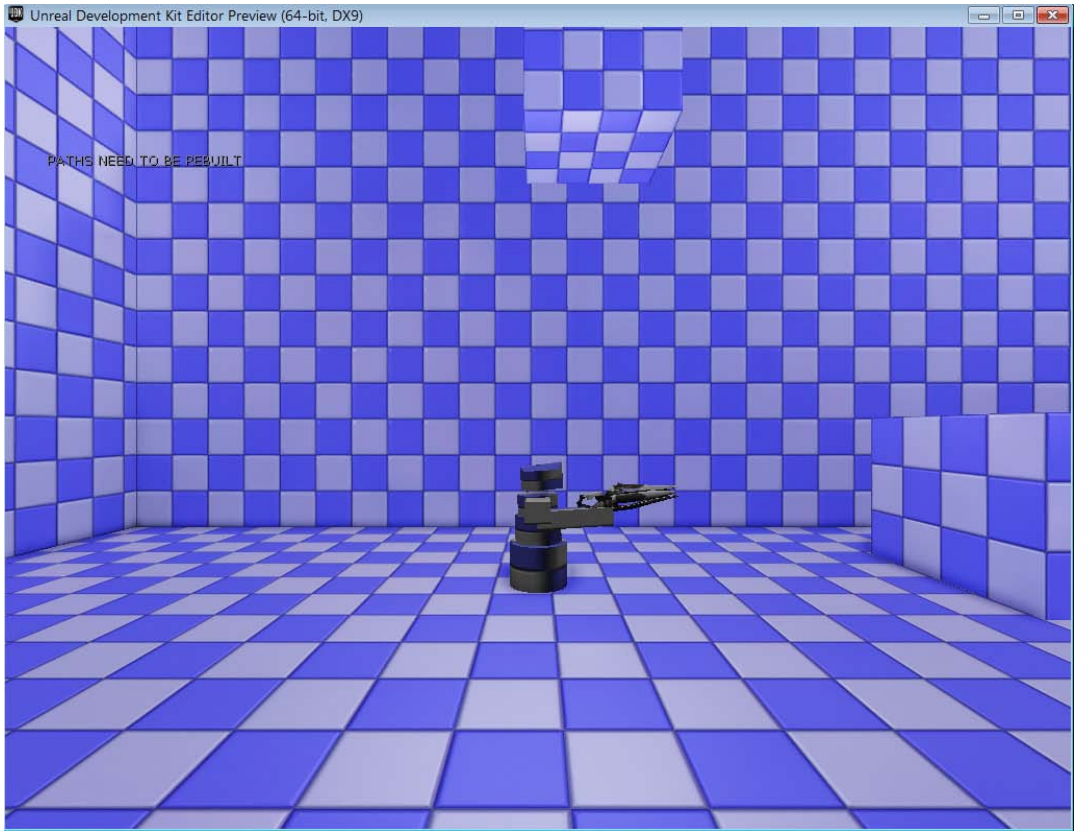
```
simulated function bool CalcCamera( float fDeltaTime, out vector out_CamLoc,
out rotator out_CamRot, out float out_FOV )
{
   out_CamLoc = Location;
   out_CamLoc.Y += CamOffsetY;
   out_CamLoc.Z += CamOffsetZ;

   out_CamRot.Pitch = 0;
   out_CamRot.Yaw = -16384;
   out_CamRot.Roll = 0;

   return true;
}
```

Pitch and Roll (X and Z) rotations are kept static at 0. The Yaw, however, which is rotating around the Y axis, is set to -16384, with 16384 being the equivalent of 90 degrees in raw rotational numbers. 65536 would be a full 360 degree turn, 32768 would be 180 degrees, et cetera. Just think of it in terms of fractions of these numbers and you should be able to get most of the rotations that you need. We will need to remember this for when we manipulate our pawn's *own* rotation later on.

Last but not least, though, we need to put in "return true" here, for no other reason than that this function has to return a value to tell the game whether the camera is working or not. Now that we have our camera overridden, we can build and see how it looks.



Now it's starting to look like a real side-scroller. Unfortunately this won't alleviate the aiming problem--the pawn will *still* fire at the background since aim is still associated with the camera, and we also haven't changed the way that the pawn moves. Not only is it not side-scrolling, it's also not terribly well-suited to platforming. We'll move on to the movement problem first, since it's relatively easy to take care of.

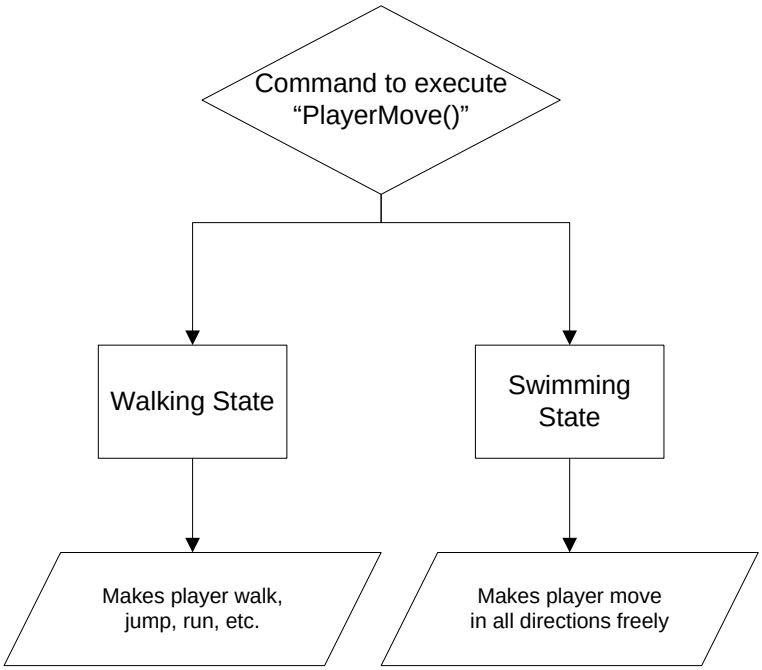
Changing the Movement Controls

On States

Now that we're getting into the movement controls and the Player Controller class, it's time to talk about "states." Unrealscript is a scripting language that natively supports state-based coding, which is the foundation of all gaming. Essentially characters, actors, and AI enter various "states," literally different states of either being or *thinking*. Common states in AI include attacking, moving towards a target, or other individual actions. Common states in terms of what *we're* doing are purely physical modes of movement; walking, swimming, flying, and driving.

Ordinarily state-based coding would involve lots of switch statements or nested if/else statements, but in Unrealscript all you need to do is declare a state the same way you'd declare a function, then place functions inside of it. Between different states you can re-define the *same* functions to change the way that they behave when they're called under specific conditions. For instance, the "PlayerMove" function

under the Walking state controls how the player runs and jumps, but under the Swimming state it controls how the player swims.



Platformercontroller.uc

We're going to start with re-defining the PlayerWalking state. Because we're writing the entire state, any functions that we *don't* write here that are in the original UTPawn class are going to be completely erased, leaving only the exact functionality that we want. I've copied the declaration of the PlayerWalking state from UTPawn.

```
class PlatformerController extends UTPlayerController;

state PlayerWalking
{
ignores SeePlayer, HearNoise, Bump;


}
```

Next, we need to copy a function called "ProcessMove," though we're going to simpify it considerably. For those not in the know, ProcessMove is the function that finally gives the command to process a pawn's movement; it takes in all the input information we have in other functions, then tells it to act based on that infrormation.

```
state PlayerWalking
{
ignores SeePlayer, HearNoise, Bump;

      function ProcessMove(float DeltaTime, vector NewAccel, eDoubleClickDir
DoubleClickMove, Rotator DeltaRot){


            if (Pawn == none)
            {
                  return;
            }

            Pawn.Acceleration = NewAccel;
      }

}
```

Again, this function is just copied; it's an old function that we're going to need to keep.

Getting Input

Now we're at a bit of an impasse, as we're not quite sure where the other step in this process--getting input and performing movement calculations--actually *is*. To find the answer, we're going to open up DefaultInput.ini in the Configs folder. This contains the reference to all the inputs that Unreal recognizes. We can add them and get rid of them as we please, if we wish--but that won't be necessary. Right now, we're just looking for the inputs that handle movement, namely the WASD keys.

DefaultInput.ini

```
.Bindings=(Name="GBA_MoveForward",Command="Axis aBaseY Speed=1.0")
.Bindings=(Name="GBA_Backward",Command="Axis aBaseY Speed=-1.0")
.Bindings=(Name="GBA_StrafeLeft",Command="Axis aStrafe Speed=-1.0")
.Bindings=(Name="GBA_StrafeRight",Command="Axis aStrafe Speed=+1.0")

...

.Bindings=(Name="W",Command="GBA_MoveForward")
.Bindings=(Name="S",Command="GBA_Backward")
.Bindings=(Name="A",Command="GBA_StrafeLeft")
.Bindings=(Name="D",Command="GBA_StrafeRight")
```

These two sets of data, under "BINDINGS USED TO ORGANIZE ALL ACTIONS" and "Keyboard and mouse inputs" are what Unreal looks for to recognize inputs--and also a valuable reference to the PlayerInput and PlayerController classes.

The first set of info defines bindable actions and what information the buttons and inputs actually send to Unreal, inside the PlayerInput class. In this case, the Strafe buttons set the the "strafe" axis--that being an axis that controls side-to-side movement--to negative or positive one; we can infer that this value multiplies the pawn's base movement speed, causing left and right movent in one axis--in our

case, the X axis. The second set of info is where the actual keys are assigned to these functions. If we look around at the different bindable actions here, we see a variety of different pieces of information being relayed to the PlayerInput, including function calls, whether a key is held down or released, and more.

PlayerController.uc (Engine class - Not to be confused with PlatformerController)

Now we know what value we're looking to change--that being "aStrafe"--but we need to know where to change it. Open up PlayerController.uc, do a CTRL+F search, and look through references to "aStrafe." The first thing you'll find is a reference to it in a function called "PlayerMove."

```
function PlayerMove( float DeltaTime )
	{
		local vector 			X,Y,Z, NewAccel;
		local eDoubleClickDir	DoubleClickMove;
		local rotator 			OldRotation;
		local bool 			bSaveJump;

		if( Pawn == None )
		{
			GotoState('Dead');
		}
		else
		{
			GetAxes(Pawn.Rotation,X,Y,Z);

			// Update acceleration.
			NewAccel = PlayerInput.aForward*X + PlayerInput.aStrafe*Y;
			NewAccel.Z  = 0;
			NewAccel = Pawn.AccelRate * Normal(NewAccel);
...
```

We can determine at a quick glance that this is the function that processes movement, and therefore the one we need to re-write in the PlayerWalking state for our own Controller.

PlatformerController.uc (PlayerWalking State)

```
function PlayerMove(float DeltaTime)
	{

	}
```

What we need to do with this is fairly straightforward: we just need to take the A and D keys' inputs, make the pawn face left and right onscreen when we press them, and make the pawn move in those directions as well. ProcessMove is going to need an Acceleration value, so we'll start out by defining one. While we're at it, we'll throw in a safeguard that's in the original PlayerMove under PlayerController; namely, if there isn't a pawn assigned, we'll just go to the "Dead" state.

```
function PlayerMove(float DeltaTime)
	{
		local Vector NewAccel;

		if (Pawn == none)
		{
			GotoState('Dead');
		}
		else
		{

		}
	}
```

Now all we have to do is fill the "else," and since we already know that PlayerInput is processing the "aStrafe" axis, we're just going to act based on that data.

```
	...
		else
		{

			if (PlayerInput.aStrafe < 0)
			{
			}
			else
			if (PlayerInput.aStrafe > 0)
			{
			}

		}
```

Simply put, we're going to make the Pawn do one thing if we have a positive Strafe value and another thing if we have a negative Strafe value. We can get a lot more specific about this if we have, say, control stick input, but for right now we'll deal in the simplest terms possible. Remember that aStrafe **does not refer** to any value that actually is happening ingame, that's just the name of the value that the A and D keys are spitting at us and filling in inside of the PlayerInput class. For it to actually have an effect, we have to set the acceleration value.

```
			if (PlayerInput.aStrafe < 0)
			{
				NewAccel.X = PlayerInput.aStrafe;
			}
			else
			if (PlayerInput.aStrafe > 0)
			{
				NewAccel.X = PlayerInput.aStrafe;
			}
```

Here I've set NewAccel.X to "PlayerInput.aStrafe," specifically tying movement in the X plane to these buttons--and nothing else.

Handling Rotation

Now we just need to get the player facing in the right direction. By looking around the original PlayerController class you'd think we'd want to use "UpdateRotation," but in actuality we want to use a *different* function--the one that "UpdateRotation" is actually referencing, which is actually in the Pawn class, and that's called "FaceRotation." "UpdateRotation" is like "ProcessMove;" it processes information about the way you're moving the pawn and then aggregates it together to form one value, then tells the pawn to face that rotation. "Pawn.FaceRotation" just tells the pawn to rotate in a specific direction, and that's all we need to do, so we're going to call it inside our if/else statements:

```
if (PlayerInput.aStrafe < 0)
{
	NewAccel.X = PlayerInput.aStrafe;
	Pawn.FaceRotation(RRight, DeltaTime);
}
else
if (PlayerInput.aStrafe > 0)
{
	NewAccel.X = PlayerInput.aStrafe;
	Pawn.FaceRotation(RLeft, DeltaTime);
}
```

The problem we now face is that FaceRotation takes in a Rotator--a vector that stores 3D rotation values--so it's not as easy as just giving it the rotations we want it to take. Fortunately all we have to do is define a set of rotators do handle this for us at the top of our script. Here I've called them RLeft, for rotation left, and RRight, for rotation right.

```
class PlatformerController extends UTPlayerController;

var Rotator RLeft, RRight;

...
```

And then, we need to give them values in DefaultProperties.

```
DefaultProperties
{
	bBehindView=true
	bForceBehindView=true
	bMouseControlEnabled=true
	RLeft = (Pitch=0, Roll=0, Yaw=65536)
	RRight = (Pitch=0, Roll=0, Yaw= 32768)
}
```

You'll notice that the notation for defining a Rotator or a Vector is a little funny; you define each sub-value separately inside a set of parenthases. You'll also notice that these values are not relative to the pawn's *own* rotation--otherwise I'd just point its spawner into the X axis and go between 0 and 180 degrees. Rather, it's based on absolute values in world space, putting left and right at 360 and 180 degrees. I'm as of yet unsure of why these values are the way they are, but that's how this works.

There's just one more thing we have to add at the bottom of the PlayerMove function to make it work for us. This goes at the very bottom of the function, but **still inside the else statement** from above.

```
...
            NewAccel.Y = 0;
            NewAccel.Z = 0;
            NewAccel=Pawn.AccelRate * Normal(NewAccel);

      ProcessMove(DeltaTime, NewAccel, unusedDoubleClickMove, Pawn.Rotation);

      }

}
```

What I'm doing here is forcing the Y and Z acceleration values to be 0 so that the pawn can't move into the background or upwards based on my input. Then, I'm borrowing a line from the original PlayerMove function. What this does is take the whole NewAccel vector we've been piecing together throughout this function, normalize it (IE make it stable, if we *are* getting a control stick as input), and then multiply it by the Pawn's acceleration. After that, I simply call the ProcessMove function that we wrote earlier and send my value for NewAccel into it. It wants a rotation to set the controller to, so for the Rotator I just pass it what the Pawn's rotation already is. Finally, there's one more value that you won't recognize, called "unusedDoubleClickMove." Let's journey back up to the top of our function to fix this one:

```
function PlayerMove(float DeltaTime)
      {
            local Vector NewAccel;
            local eDoubleClickDir unusedDoubleClickMove;

...
```

eDoubleClickDir specifically refers to "dodging" in Unreal Tournament; IE, double-tapping a direction. ProcessMove wants this value, even though we don't do anything with it. As of right now we can't re-define it to get *rid* of eDoubleClickDir, so our workaround is to simply feed it "unusedDoubleClickMove," which we're leaving undefined. We'll get a warning for not defining it from our compiler, but the pawn will work fine since we never use this value anyway.

Now if we build and try out our game, we'll have a fully functional, side-scrolling pawn! Except for one thing...

Adding Jump Support

That's right--this new pawn can't jump. We haven't defined any functionality that would allow it to jump! Fortunately we can add that in quickly. Another examination of the original PlayerMove function reveals to us that there isn't actually any function in the controller for jumping; rather, PlayerMove does calls to a function inside the Pawn called "DoJump()." If we open that one up, we find that it's got a little bit of flow control and some references to double jumping and physics states--but for the most part, all it does is set "Velocity.Z" to a value called "JumpZ." In other words, it sets the pawn's upwards velocity instantaneously to some value, more or less approximating a jump.

So, all we really have to do is make a quick reference to "DoJump" in our own PlatformerController script.

```
if (bPressedJump == true)
	{
		Pawn.DoJump( bUpdating );
		bPressedJump = false;
	}
```

All this is doing is telling the Pawn to do its "DoJump" function when bPressedJump becomes true. As soon as it's done ordering DoJump, bPressedJump becomes false, and therefore the player will be order to command another jump. The default inputs are already sending this signal, so we barely had to write a thing.

More Useable Movement

Now that we've got the pawn's behaviors working--not only that, but on Unreal's own terms--all that's left for us to do is to make this into something a little bit more savory for platforming, as the jump height on the pawn is pathetic.

UTPawn.uc (UTGame Class)

If we want to know what values to edit to make the jump work more to our tastes, all we need to do is open up UTPawn.uc and take a look at *its* defaultproperties.

```
GroundSpeed = 440.0
AirSpeed=440.0

...

AirControl=+0.35
DefaultAirControl=+0.35

...
```

JumpZ isn't defined here, so we can assume that it's defined elsewhere, probably in either UDKPawn, GamePawn, or the original Pawn class in the Engine folder. At any rate it's set to a velocity of about 330, which is pretty pathetic; less than walking speed. No wonder this thing can't platform!

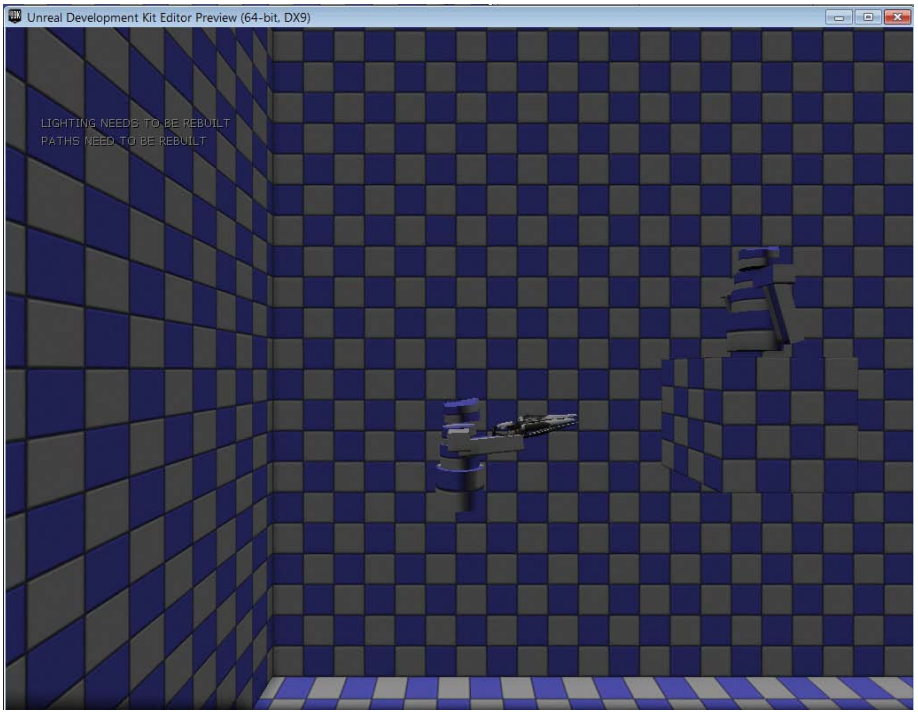
PlatformerPawn.uc

We're going to be a little more generous. We'll go into the DefaultProperties for our *own* pawn, then put in the following values:

```
AirControl=+1
DefaultAirControl=+1
JumpZ = 1000;
```

As you might've guessed, AirControl and DefaultAirControl are fractions, just like the input we get from an axis. "1" means full control, while a decimal like 0.25 is damped down to one quarter of full control. Feel free to experiment with these values to get desired results.

Now when we save it and start our game up...



... our pawn jumps *incredibly* high and turns on a dime in mid-air!

Conclusion

And voila! We now have a pawn fit for side-scrolling action. All that's left is to fix its aiming problems--which will come in our next chapter, alongside working with Animtrees.