Chapter 9 - Creating New Weapons and Inventory

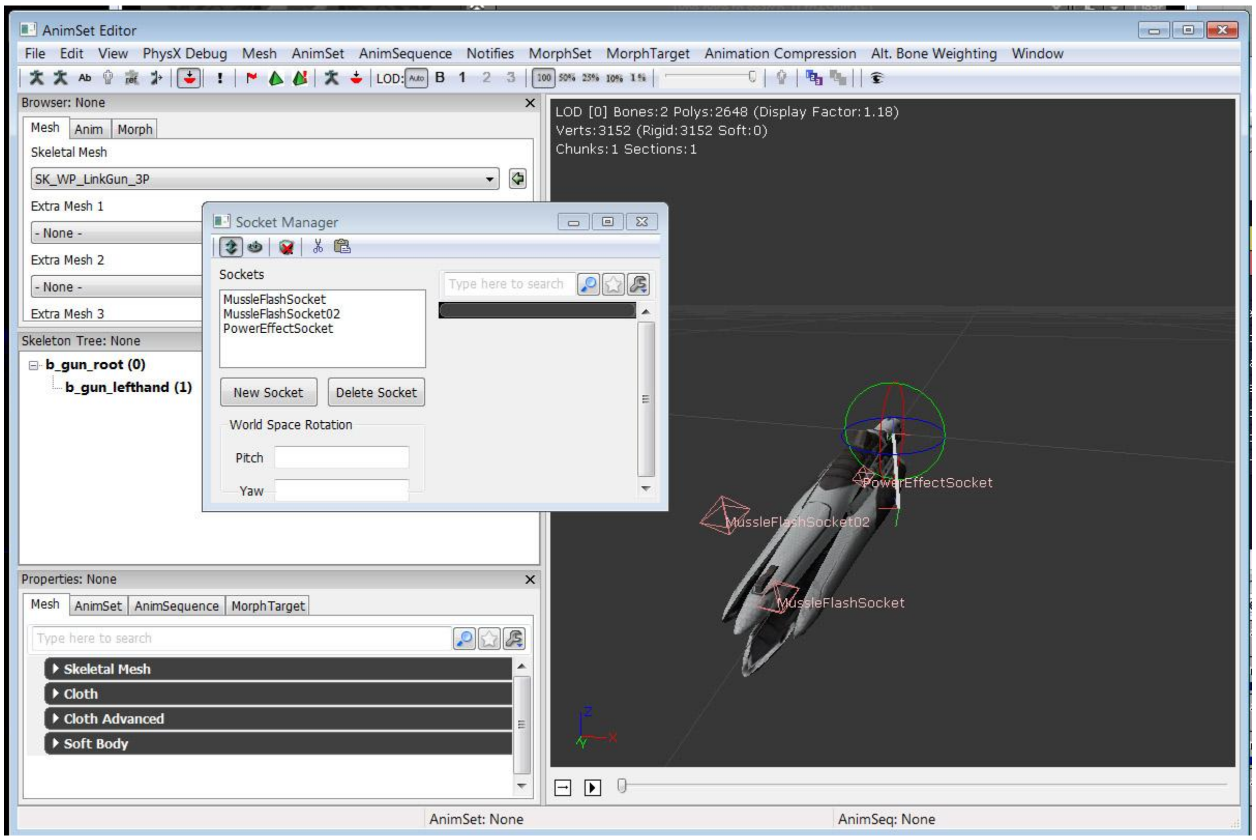
Contents

Introduction

Now that we've got side-scrolling action going on, it's time for us to arm our pawn. In this chapter we'll be developing a brand new weapon with original behavior; something a little simpler than the complex dual-mode guns Unreal uses, with an emphasis on dissociating the inventory and controls from Unreal Tournament. We won't be doing it with original assets since this isn't a modeling tutorial, but we'll be covering the necessities on that end briefly. In fact, that's where we'll start.

Assets

Gun and weapon meshes, you may have noticed, are also skeletal meshes.



Skeletal Mesh: SK_WP_LinkGun_3P
Extra Mesh 1: - None -
Extra Mesh 2: - None -

- b_gun_root (0)
  - b_gun_lefthand (1)

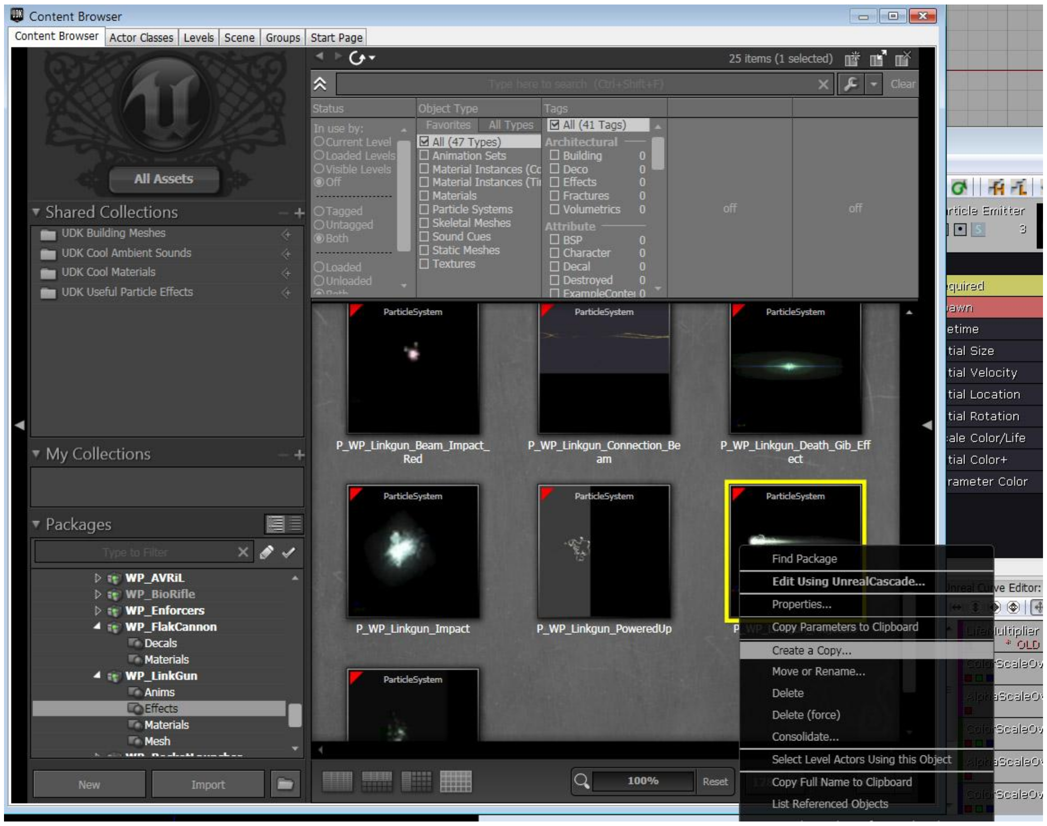
Sockets:
MussleFlashSocket
MussleFlashSocket02
PowerEffectSocket

```
LOD [0] Bones:2 Polys:2648 (Display Factor:1.18)
Verts:3152 (Rigid:3152 Soft:0)
Chunks:1 Sections:1
```

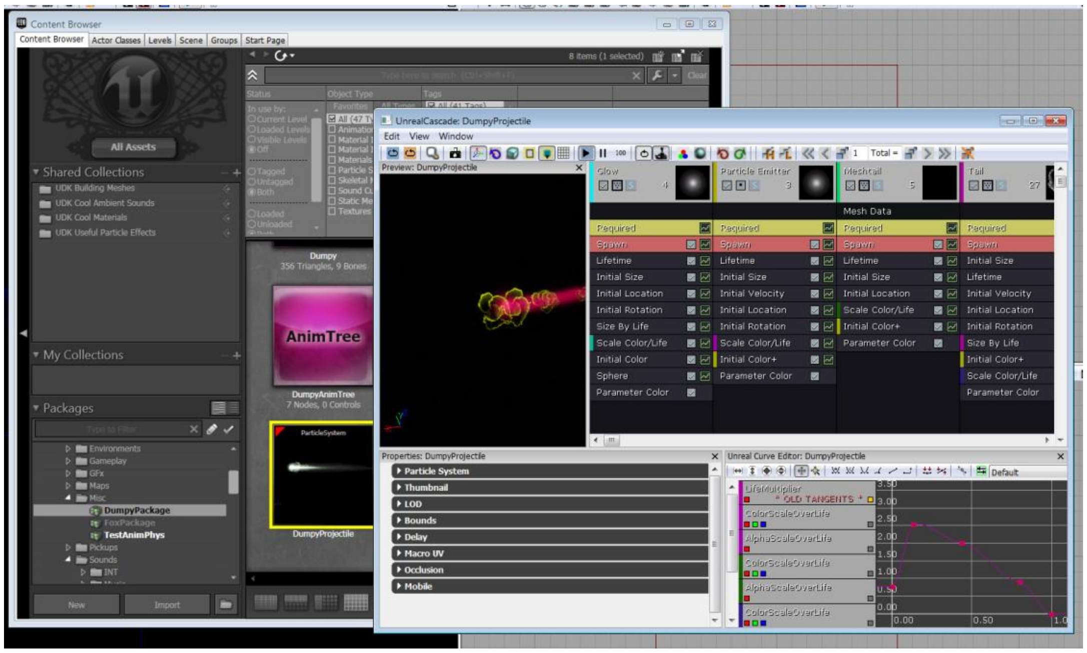
They're not *complicated*, consisting of exactly one bone and two joints, but they're skeletal meshes. This is so that we can take advantage of Unreal's socket support, adding projectiles and particles accordingly. Anytime you make a new weapon, make sure to use a skeletal mesh and add whatever sockets you need, and take note of the names. In this case we want to take note of "MussleFlashSocket," which we'll be referencing when we code our weapon.

For the weapon mesh we're just going to be using the Link Gun mesh. It's serviceable enough. For the projectile, though, we're going to take a little time to make a brand new particle. And by that I mean we're going to take the Link Gun particle, duplicate it, and make it flaming hot pink.

Find the Link Gun projectile in the "Weapons" package, under "WP_Linkgun" and "effects." Right click on the LinkPlasma and pick "Create a Copy." Send the copy to DumpyPackage.



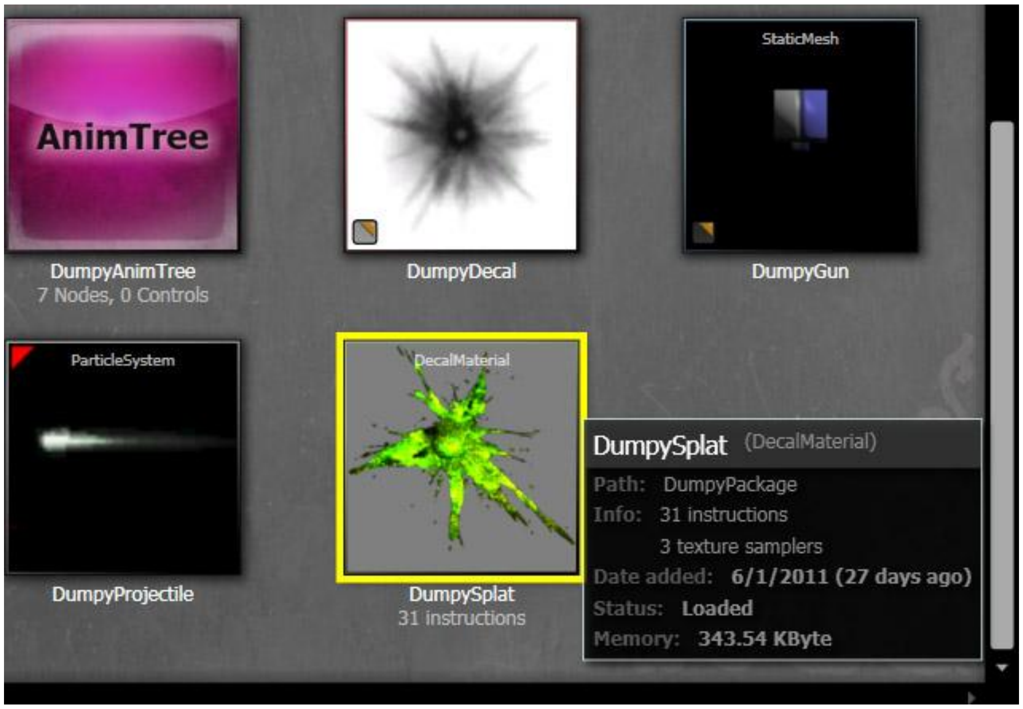
I called it "DumpyProjectile." Now, double-click it to bring up Unreal Cascade, Unreal's particle editor.


UnrealCascade: DumpyProjectile
Preview: DumpyProjectile
Properties: DumpyProjectile
Unreal Curve Editor: DumpyProjectile
DumpyPackage
DumpyProjectile

Copyright Michael Scott Prinke 2011

You can use "Initial Color" and "Scale Color/Life" to mess with it. I just picked this awful shade of pink and yellow to make it look reasonably different. Obviously you can also make a custom projectile particle or pick any other particle here.

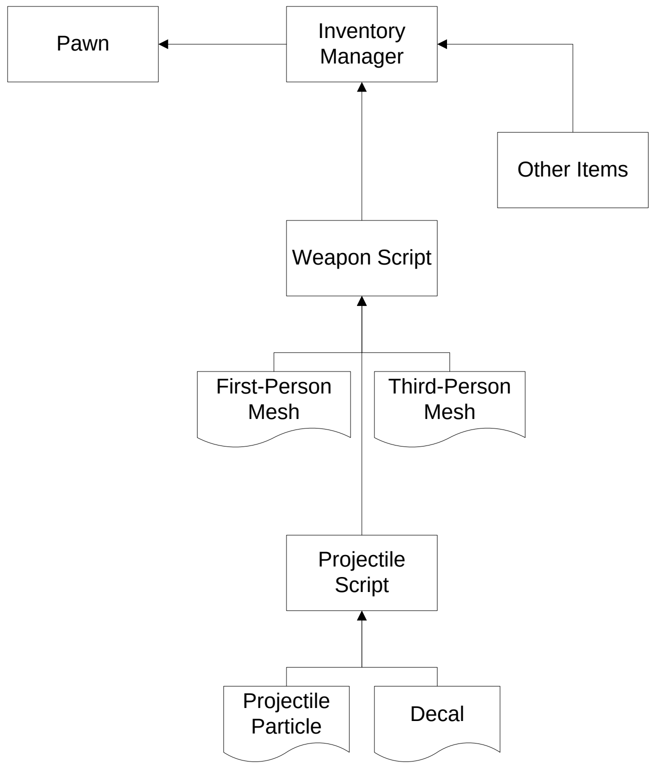
There's one last ingredient we need for our weapon, and that's a decal we can slap on walls and objects when we hit them. Since this is hideously garish so far I'm going to take the Bio Rifle's splat decal, duplicate it, and call it "DumpySplat."



DumpySplat (DecalMaterial)
Path: DumpyPackage
Info: 31 instructions
3 texture samplers
Date added: 6/1/2011 (27 days ago)
Status: Loaded
Memory: 343.54 KByte

With all that gathered up, we can move on to the code.

Inventory Logic Overview



Everything in a player's inventory is handled through an Inventory Manager script, which simply stores an array of weapons and items that the player has at their disposal, along with some info on how the player interacts with them. The default InventoryManager.uc script does a lot of what any first-person shooter needs, storing weapons and the like, while UDKInventoryManager and UTInventoryManager add a little bit of the specifics of Unreal Tournament like accommodations for powerups and health kits.

Weapons are comprised of several parts, including a first-person mesh, a third-person mesh, and a projectile. Their only function is to give off the cosmetic aspects of the weapon being held and to produce projectiles. A custom melee weapon like a sword can easily be altered to have all its functionality self-contained, adding collision to the blade and whatnot and damaging enemies based on that.

The projectile *itself* is actually what matters in the case of a gun. It's what does the damage and its script is where we specify its behavior; things like its firing arc, whether it's affected by gravity, whether it bounces, whether it explodes on impact, et cetera.

We're going to make an Inventory Manager for our game to start out with.

PlatformerInventoryManager.uc

```
class PlatformerInventoryManager extends InventoryManager;

DefaultProperties
{
	PendingFire(0)=0
}
```

This is the only code we'll be writing for it since all we're doing is putting in gun support. You'll notice I'm only using the InventoryManager script from the top level, because I want to use as little of UT's logic as I can and may want to expand my inventory's capabilities in other ways. The "PendingFire" in the Default Properties is a blank that I've filled in specifying that fire mode zero is the default firing mode. InventoryManager, I'll note, does not contain any references to ammo. We have to add this ourselves if we want it, but since we're imitating Contra--a game where players fire limitless bullets streams of without reloading--we won't be doing that.

Building The Gun

Next we need to create our weapon. I'm calling it "ContraGun" since our game is imitating Contra.

A short note on making weapons: virtually every function necessary to make a weapon work is contained in the "Weapon.uc" script. Most of them are undefined, however, and it's necessary to go back and add functionality to them, which is mainly what I'm doing here.

ContraGun.uc

```
class ContraGun extends UDKWeapon;

DefaultProperties
{
    Begin Object class=SkeletalMeshComponent Name=GunMesh
      SkeletalMesh=SkeletalMesh'WP_LinkGun.Mesh.SK_WP_Linkgun_3P'
      HiddenGame=FALSE
      HiddenEditor=FALSE
      Scale=2.0
    end object
    Mesh=GunMesh
    Components.Add(GunMesh)
}
```

I've taken UDKWeapon since it has the majority of the support we need to make a decent gun. This is still a fairly baseline, UT-dissociated script without all the references to first-person models, as we won't be using a first-person model at all.

I've already filled in the default properties for adding the mesh here. It's straightforward stuff; I put the Link Gun's skeletal mesh in and scaled it by a factor of 2 to make it look more appropriate in Dumpy's arm.

Speaking of which, let's attach it to the player.

```
simulated event SetPosition(PlatformerPawn Holder)
{
    local SkeletalMeshComponent compo;
    local SkeletalMeshSocket socket;
    local Vector FinalLocation;

    compo = Holder.Mesh;
    if (compo != none)
    {
      socket = compo.GetSocketByName('PlatformWeapSocket');
      if (socket != none)
      {
          FinalLocation = compo.GetBoneLocation(socket.BoneName);
      }
    }
    SetLocation(FinalLocation);

}
```

To explain: "compo" refers to the skeletal mesh of the player pawn, IE "Holder," and socket refers to the socket we created to hold weapons, IE "PlatformWeapSocket." All we're doing is setting the location of this weapon to that socket, which we created in an earlier tutorial.

Next we need to be able to equip the weapon when we switch to it.

```
simulated function TimeWeaponEquipping()
{
    AttachWeaponTo( Instigator.Mesh,'PlatformWeapSocket' );
    super.TimeWeaponEquipping();
}
```

All this function does is attach the weapon on equip. It's inherited from the default weapon script, but I re-wrote it to take into account "PlatformWeapSocket."

```
simulated function AttachWeaponTo( SkeletalMeshComponent MeshCpnt, optional
Name SocketName )
{
    MeshCpnt.AttachComponentToSocket(Mesh,SocketName);
}
```

"AttachWeaponTo" is declared in the "Weapon" script, but that doesn't contain anything; we need to define it ourselves. I got this line from UTWeapon, and is a necessary function for attaching weapons to a mesh, called whenever the InventoryManager switches to it. UTWeapon makes it a lot more complicated than this, with many references to the first and third-person meshes, but all we need is third-person, so that's all I'm referencing.

Next, we have to set where the gun actually shoots from--otherwise it fires from the player's chest by default. All we have to do is set it to shoot from MussleFlashSocket01, which we saw in the skeletal mesh viewer earlier.

```
simulated function vector GetPhysicalFireStartLoc(optional vector AimDir)
{
      Local SkeletalMeshComponent AttachedMesh;
      local vector SocketLocation;

      AttachedMesh = SkeletalMeshComponent(Mesh);
      AttachedMesh.GetSocketWorldLocationAndRotation(MussleFlashSocket,Socket
Location);
      return SocketLocation;
}
```

This is one of the few times we need to return a value, specifically a vector from which a projectile can originate. "GetSocketWorldLocationAndRotation" does exactly what it says for the mesh we specify, grabbing "MussleFlashSocket" and spitting its location out into SocketLocation, which then gets returned.

With that out of the way we don't have much more to add--just a couple of tweaks inside the DefaultProperties.

```
DefaultProperties
{
    Begin Object class=SkeletalMeshComponent Name=GunMesh
      SkeletalMesh=SkeletalMesh'WP_LinkGun.Mesh.SK_WP_Linkgun_3P'
      HiddenGame=FALSE
      HiddenEditor=FALSE
      Scale=2.0
    end object
    Mesh=GunMesh
    Components.Add(GunMesh)

      FiringStatesArray(0)=WeaponFiring
    WeaponFireTypes(0)=EWFT_Projectile
    WeaponProjectiles(0)=class'PlatformGame.ContraShot'

      FireInterval[0] = 0.1
}
```

FiringStatesArray is an array that all weapons possess, containing primary, secondary, and, if you want, tertiary, quartiary, quintupliary, sextiary, septiary, octiary, noniary, et cetera firing modes, with (0) being the primary firing mode per our specification.

WeaponFireTypes refers to the fire type of the weapon, and there are two of interest: Projectile and Hitscan. Projectile weapons shoot out physical projectiles that can be seen traveling through the air, and do well for laser and plasma bolts or missiles. Hitscan weapons are instant hit, best-suited to bullets and beams. Here we see it set to the same firetype as the primary firing state.

WeaponProjectiles refers to the projectile we'll be using--a custom projectile called "ContraShot," which we haven't made yet. Again, its number coincides with the FiringStates.

And finally, I altered the FireInterval of the primary firing state, changing it to a shot every 0.1 second to get a good stream of rapid fire and really show the differences between this and the Link Gun.

That's it. That's all our weapon does. It exists in our hand, and it shoots a projectile. We can code in more for special effects like muzzle flashes, obviously, or popping cartridges out on re-loading, but we're going to keep this simple for right now. It's time for us to move on to the projectile, which is also simple.

Building the Projectile

ContraShot.uc

```
class ContraShot extends UTProjectile;

DefaultProperties
{
	Begin Object Name=CollisionCylinder
	CollisionRadius=8
	CollisionHeight=16
	End Object

	ProjFlightTemplate=ParticleSystem'DumpyPackage.DumpyProjectile'
	DrawScale=2.8

	ExplosionSound=SoundCue'A_Vehicle_Manta.SoundCues.A_Vehicle_Manta_Shot'
	SpawnSound=SoundCue'A_Vehicle_Cicada.SoundCues.A_Vehicle_Cicada_Missile
Eject'

	Damage=25
	MomentumTransfer=10
}
```

ContraShot is extended from UTProjectile for one very, very simple reason: UTProjectile supports decals. That's literally the only thing it does that UDKProjectile doesn't, and it's a pretty important feature. We could live without it in our game, but let's just assume that for your purpose you actually do want to use it.

I've already taken the liberty of applying a Collision Cylender to the shot, whose axis is set so that the height is down the length of the projectile. I also defined "ProjFlightTemplate," which indicates the particle system for this projectile. DrawScale, again, is beefed up to make the projectile more visible.

I put in some sound effects for the "explosion," IE, when it collides with something, and threw in a spawn sound for when it initially spawns. I picked the Manta and Cicada shots for no particular reason, they just jumped out at me while I was exploring the library.

Finally, I gave the shot a damage rating (25 per shot) and a light momentum transfer to apply some force to objects that're hit by the bolts.

This is actually the bulk of the work for this projectile. Other than this, we only have to write two functions: one for colliding with Actors and one for colliding with walls.

```
simulated function ProcessTouch(Actor Other, Vector HitLocation, Vector
HitNormal)
{

...
```

As you can see ProcessTouch takes in a lot of things, including the actor itself that we've hit, the location where its collision was touched, and the angle at which it was hit (IE: HitNormal).

```
    if ( Other != Instigator )
    {
      WorldInfo.MyDecalManager.SpawnDecal
      (
          DecalMaterial'DumpyPackage.DumpySplat',
          HitLocation,
          rotator(-HitNormal),
          128, 128,
          256,
          false,
          FRand() * 360,
          none
      );
```

Next, we add the decal to whatever we just hit, spawning it into the world via a DecalManager. It simply applies the material on top of a hitlocation, rotates it facing away from the angle we hit our target at, applies a width and height (128x128), a thickness (no idea why this is important), turns "noclip" to false, rotates the decal at a random angle in 360 degrees, and then sets everything else to "none" since no other variables are important to us for spawning this. A great deal more can be done with a spawned decal than even this, including turning on and off its projection on terrain and skeletal meshes and returning precisely which *bone* it hits in a skeletal mesh, but we want simple collision and that's all.

Note the semicolon after SpawnDecal. Another way of writing this function would be:

```
WorldInfo.MyDecalManager.SpawnDecal(DecalMaterial'DumpyPackage.DumpySplat',
HitLocation, rotator(-HitNormal), 128, 128, 256, false, FRand() * 360,);
```

It is in fact a function like all the others we've been using so far, except that it takes in *many* more variables than most; hence our re-formatting for readability's sake.

```
Other.TakeDamage( Damage, InstigatorController, Location, MomentumTransfer *
Normal(Velocity), MyDamageType,, self);

Destroy();

}
```

Now we apply "TakeDamage" to the actor we hit, and call "Destroy()" to get rid of our projectile. TakeDamage, you'll notice, has a lot of parameters as well. First there's our Damage, which we set earlier, then there's the InstigatorController--IE, the person who "owns" this projectile, which is important for keeping track of scoring in some of these; the Location where hit and Momentum, which is calculated by the raw momentum that we gave the projectile and the angle of hit multiplied by the speed of the shot; our damage type, which isn't important, and the projectile class itself identifying the type of projectile the target was hit by--a specification that Unreal Tournament often falls back on for telling who got killed by what in announcements.

The "HitWall" function is practically identical, except that it applies no damage as walls can take none.

```
simulated event HitWall(vector HitNormal, actor Wall, PrimitiveComponent
WallComp)
{
	WorldInfo.MyDecalManager.SpawnDecal
	(
	    DecalMaterial'DumpyPackage.DumpySplat',
	    Location,
	    rotator(-HitNormal),
	    128, 128,
	    256,
	    false,
	    FRand() * 360,
	    none
	);
	Destroy();
```

We could change a lot about what happens to the projectile when it hits a wall, like making it bounce, but we'll concede it for now and be pleased to have a complete projectile. For now, we have a few last things we need to add in order to put our weapon in the player's hands.

PlatformerPawn.uc

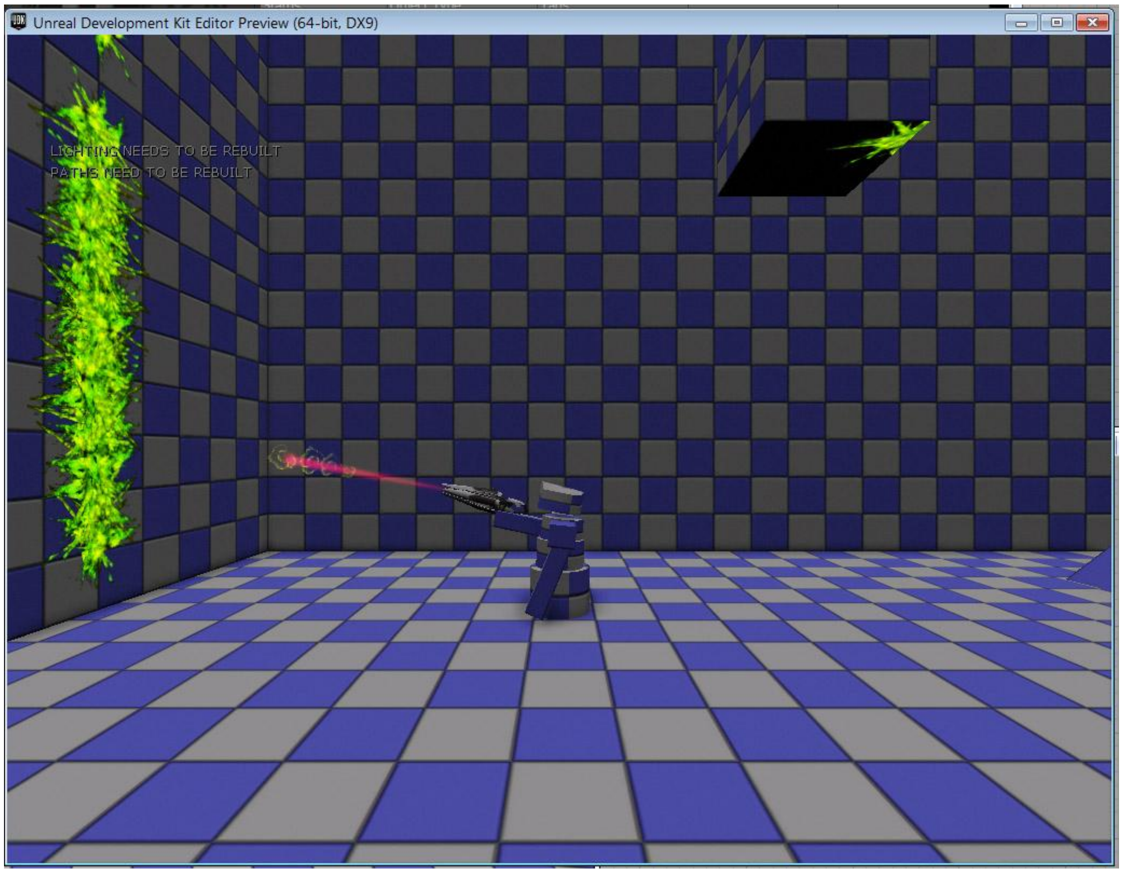
```
InventoryManagerClass = class'PlatformerInventoryManager'
```

Naturally we need to specify that Platformer Pawns use the PlatformerInventoryManager. Just add this to its default properties.

Kismet

Finally, throw in a little kismet at the start of your level to add the ContraGun to your player's inventory at the start of the game. We can do this in code too, there is a DefaultInventory we can edit, but it makes little difference.

Now, finally, you should find...


Unreal Development Kit Editor Preview (64-bit, DX9)
LIGHTING NEEDS TO BE REBUILT
PATHS NEED TO BE REBUILT

Conclusion

Voila! A functional, original (sort of) weapon. There's a lot more that has to be done in order to build a weapon for a first-person game or to fully flesh out the presentation, but since the topic of these tutorials is to get a *different* kind of game up and running--and quickly--we'll be leaving that to others for now. At any rate, we've covered the basics. Much can be done to expand its capabilities, but this should get you off to a good start creating your own custom inventory for your side-scroller.