Programming for Artists and Designers

Tutorial 10: Continuing with Modding Weapons

Aims: To continue with modifying weapons in preparation for Assessment 2.

Introduction:
This weapon tutorial builds upon what you have learnt before. It expands upon some of the classes already available, but more importantly alters the way the projectile behaves.
We are basically going to be coding a napalm thrower.

Files Needed

Again we are going to need to need to code 7 files for our weapon to work, luckily most of these build upon what has been before making only slight modifications.

If you remember we generally need the following files:
Weapon - Inventory and Pickup
Ammo - Inventory and Pickup
Projectile – Type, Fire Class and Damage Type
Attachment Class

Coding the Weapon Class

As our weapon is going be relatively unique we do not need to inherit any default properties or functionality from any other class so extend from the base weapon class.

The weapon class contains all the core functionality for the weapon operate.

It's an abstract class, so one can never instantiate a "weapon", we must deriver from it – basically we can never make an in game instance of a "weapon" we can only make instances of classes derived from weapon such as "linkgun"

Most of the code below is actually very similar to what is defined in the linkgun weapon class.

```
class BoomStick extends Weapon;

defaultproperties
{
    ItemName="BoomStick"

    FireModeClass(0)=BoomFire //what we shoot
    FireModeClass(1)=BoomFire
    //the pickup item which spawns this to our inventory
```

```
    PickupClass=class'BoomStickPickup'

                          bMatchWeapons=true
    IconMaterial=Material'InterfaceContent.Hud.SkinA'
    IconCoords=(X1=200,Y1=190,X2=321,Y2=280)

    InventoryGroup=5
    DrawScale=1.0
    Mesh=mesh'Weapons.LinkGun_1st' //which mesh to use
    BobDamping=1.575000
    EffectOffset=(X=100,Y=25,Z=-3)
    IdleAnimRate=0.03
    PutDownAnim=PutDown
    DisplayFOV=60

    PlayerViewOffset=(X=-2,Y=-2,Z=-3)
    PlayerViewPivot=(Pitch=0,Roll=0,Yaw=500)
    UV2Texture=Material'XGameShaders.WeaponEnvShader'

    AttachmentClass=class'BoomAttach' //attachement class
    SelectSound=Sound'WeaponSounds.LinkGun.SwitchToLinkGun'
    SelectForce="SwitchToLinkGun"

    AIRating=+0.68
    CurrentRating=+0.68
    DefaultPriority=5
}
```

Coding the Fire class

All we are doing in this class is defining what we shoot.

```
class BoomFire extends BioFire;

defaultproperties
{
    AmmoClass=class'BoomAmmo' // what type of ammo to use
    ProjectileClass=class'CVGPackage.BoomProj'
    //what projectile to spawn
}
```

Ammo Type

Here we just set up what ammo type our weapon uses – with the other weapon we used redeemer rockets this weapon has its own ammunition that we will define.
This is very similar to what we have covered before – we set the name of the ammo the interface material used, the max you can carry and how many you get when you first pickup the weapon.

```
class BoomAmmo extends Ammunition;

DefaultProperties
{
    ItemName="Boomstick shells" //our name
```

```
    IconMaterial=Material'InterfaceContent.Hud.SkinA'
    IconCoords=(X1=545,Y1=75,X2=644,Y2=149)

    PickupClass=class'BoomAmmoPickup'
// what our ammo pickup class is
    MaxAmmo=50
    InitialAmount=20
}
```

Ammo Pickup

When you pickup an item, it spawns a *different* item in your inventory for use by the player.

In this case:

- BoomAmmoPickup creates the BoomAmmo inventory item
- BoomAmmo knows what item creates it

```
class BoomAmmoPickup extends UTAmmoPickup;

DefaultProperties
{
    InventoryType=class'BoomAmmo' //what item to create in inventory

    PickupMessage="You picked up some BOOM-stick ammo"
    PickupSound=Sound'PickupSounds.FlakAmmoPickup'
    PickupForce="FlakAmmoPickup"  // jdf

    AmmoAmount=20

    MaxDesireability=0.320000
    CollisionHeight=8.250000

    StaticMesh=StaticMesh'WeaponStaticMesh.BioAmmoPickup' //mesh to
use
    DrawType=DT_StaticMesh
}
```

Projectile Fire Class

This is where it gets complicated.

Most of the core functionality of this class still comes from BioGel, so you can cut & paste what you need.
In this case most of the variable declarations & defaults are the same, but if you cut and paste be sure to check the code here against your own – as you are likely to generate errors.

```
class BoomProj extends Projectile;
// most of the bioblob var's -eze
var xEmitter Trail;
var() int BaseDamage;
var() float RestTime;
var() float TouchDetonationDelay; // gives player a split second to
jump to gain extra momentum from blast
var() float DripTime;
```

```
var Vector SurfaceNormal;
var bool bCheckedSurface;
var int Rand3;
var bool bDrip;
var bool bOnMover;

var() sound ExplodeSound;
var AvoidMarker Fear;
```

This function does exactly what it says on tin…set pawns on fire.
StickyFire is a new class we will code – see below.
The TorchPawn function is not really that fancy.
In effect, it instantiates the napalm and sticks it to the pawn. Then we call a function in our sticky fire object to set up the pawn for the torching.

There is the possibility that we can keep piling up the fire on the pawn & this can be expensive on the CPU…So we only want there to be ONE flame attached at any point.
In the projectile class, we look for all the other stickyfires in existence, if we find one, we don't create a new one.

```
function TorchPawn(Pawn other, vector hitloc)
{
    local StickyFire SFire,otherfire;

    foreach AllActors(class'StickyFire',otherfire)
         if(otherfire.Base == other )
         {
             return; //bail if already burning
         }

            SFire=spawn(class'StickyFire',owner,,hitloc,);
                    SFire.ToTorch(other, hitloc);
                        SFire.SetBase(other);
                   SFire.default.lifespan=lifespan;
}
```

State Flying is for when the goo is airborne and has specific functionality for when it hits a wall.
We have altered the Processtouch function part of this method.

```
auto state Flying
{
    simulated function Landed( Vector HitNormal )
    {
        local Rotator NewRot;

        if ( Level.NetMode != NM_DedicatedServer )
        {
            PlaySound(ImpactSound, SLOT_Misc);
            // explosion effects
        }

        SurfaceNormal = HitNormal;

                                    spawn(class'BioDecal',,,, rotator(-HitNormal));
```

```
        bCollideWorld = false;
        SetCollisionSize(10.0,10.0);
        bProjTarget = true;

                      NewRot = Rotator(HitNormal);
                         NewRot.Roll += 32768;
        SetRotation(NewRot);
        SetPhysics(PHYS_None);
        bCheckedsurface = false;
        Fear = Spawn(class'AvoidMarker');
        GotoState('OnGround');
    }

    simulated function HitWall( Vector HitNormal, Actor Wall )
    {
        Landed(HitNormal);
        if (Mover(Wall) != None)
        {
            bOnMover = true;
            SetBase(Wall);
            if (Base == None)
                BlowUp(Location);
        }
    }
    // removed all touch functions specific to bioglob -eze
    simulated function ProcessTouch(Actor Other, Vector HitLocation)
    {
        if (Other != Instigator && Other.IsA('Pawn'))
        {
            TorchPawn(Pawn(other),HitLocation);//light the contacting
pawn on fire.-eze
        }
    }
}
```

In State onGround, we made a similar change to the process touch.
This state has all the functionality for when the goo is just sitting on a surface.
Notice that it will actually drip from the ceiling.
Also notice that the goo takes on different shapes when it airborne or on a wall…
How does it do this?...it's an animated mesh & we tell specific keyframes of that animation to be played.
By changing the skin 0 in the defaults, we can change the look & colour of our goo: Skins(0)=FinalBlend'XEffectMat.RedShell'
We can then play animations using various latent function calls.
As long as there is a matching name, an animation will play.

The 3 main functions we need to know about for playing animations. All of which are defined in the actor class as:

- PlayAnim
- LoopAnim
- FinishAnim

PlayAnim

intrinsic(259) final function PlayAnim(name Sequence,
optional float Rate, optional float TweenTime);

Plays a named animation sequence in the actor's Mesh once. The optional Rate scales the animation's default rate. If a nonzero TweenTime is specified, the animation is first tweened from whatever is currently displayed to the start of the named animation sequence, before the animation sequence plays. When the animation playing completes, it stops and calls your optional AnimEnd() event and causes any latent FinishAnim() calls to return.

LoopAnim
Intrinsic(260) final function LoopAnim(name Sequence,
optional float Rate, optional float TweenTime,
optional float MinRate);

Loops the animation sequence forever. The AnimEnd() is called at the end of the sequence, and FinishAnim() calls return at the end of each iteration of the loop.

FinishAnim
intrinsic(261) final latent function FinishAnim();

Waits for the currently playing or looping animation to reach the end of the sequence

```
state OnGround
{
    simulated function BeginState()
    {
        PlayAnim('hit');
        SetTimer(RestTime, false);
    }

    simulated function Timer()
    {
        if (bDrip)
        {
            bDrip = false;
            SetCollisionSize(default.CollisionHeight,
default.CollisionRadius);
            Velocity = PhysicsVolume.Gravity * 0.2;
            SetPhysics(PHYS_Falling);
            bCollideWorld = true;
            bCheckedsurface = false;
            bProjTarget = false;
            LoopAnim('flying', 1.0);
            GotoState('Flying');
        }
        else
        {
            BlowUp(Location);
        }
    }

    simulated function ProcessTouch(Actor Other, Vector HitLocation)
    {
        if (Other.IsA('Pawn'))
```

```
        {
            bDrip = false;
            //SetTimer(TouchDetonationDelay, false); // not req'd -
eze
            TorchPawn(Pawn(other),HitLocation);//light the contacting
pawn on fire.
        }
    }

    function TakeDamage( int Damage, Pawn InstigatedBy, Vector
HitLocation, Vector Momentum, class<DamageType> DamageType )
    {
        if (DamageType.default.bDetonatesGoop)
        {
            bDrip = false;
            SetTimer(0.1, false);
        }
    }

    simulated function AnimEnd(int Channel)
    {
        local float DotProduct;

        if (!bCheckedSurface)
        {
            DotProduct = SurfaceNormal dot Vect(0,0,-1);
            if (DotProduct > 0.7)
            {
                PlayAnim('Drip', 0.66);
                bDrip = true;
                SetTimer(DripTime, false);
                if (bOnMover)
                    BlowUp(Location);
            }
            else if (DotProduct > -0.5)
            {
                PlayAnim('Slide', 1.0);
                if (bOnMover)
                    BlowUp(Location);
            }
            bCheckedSurface = true;
        }
    }
    //removed merge goop function...not required -eze
}


function BlowUp(Vector HitLocation)
{
    if (Role == ROLE_Authority)
    {
        Damage = BaseDamage + Damage;
        DamageRadius = DamageRadius;
        MomentumTransfer = MomentumTransfer ;
        if (Physics == PHYS_Flying) MomentumTransfer *= 0.5;
        HurtRadius(Damage, DamageRadius, MyDamageType,
MomentumTransfer, HitLocation);
    }

    PlaySound(ExplodeSound, SLOT_Misc);
```

```
    Destroy();
}
```

PostBeginPlay function is needed to set our initial velocity of the napalm. It also sets the napalm effect: `local HitFlameBig BFlame;`

HitFlameBig is an xEmitter (a.k.a. a particle emitter) and we've already seen the spawn function.

But what's Setbase? And why self?

SetBase
intrinsic(298) final function SetBase(actor NewBase);
Sets the actor's Base. A base of None means that the actor moves alone; setting the base to another actor in the world causes this actor to move and rotate along with its base. An example of using a base is standing on a moving platform.

So we bind it to ourselves…which in this case is the projectile. In effect you're gluing the particle emitter to the projectile.

This does not make it do damage over time like napalm so now we So we want to borrow 2 states form bioGel and make some modifications
Flying and on ground.

Most of both of these states I leave unchanged, we're really interested in the process touch function.

```
simulated function PostBeginPlay()
{
    local HitFlameBig BFlame; //added -eze
    Super.PostBeginPlay();

    SetOwner(None);

    LoopAnim('flying', 1.0);

    if (Role == ROLE_Authority)
    {
        Velocity = Vector(Rotation) * Speed;
        Velocity.Z += TossZ;
    }

    if (Role == ROLE_Authority)
         Rand3 = Rand(3);
                    if ( Level.bDropDetail )
                                {
                                                bDynamicLight = false;
                                                  LightType = LT_None;
                                }

    //Spawn the flames & bind them to the goop -eze
```

```
                BFlame=spawn(class'HitFlameBig',,,,);
                        BFlame.Setbase(Self);
}


defaultproperties
{
    //our damage class
    MyDamageType=class'DamTypeBoomStick'

    Speed=2000.0
    TossZ=0.0
    BaseDamage=10.0//reduced  -eze
    Damage=10.0    //reduced -eze
    DamageRadius=120.0
    MomentumTransfer=40000
    RestTime=2.25
    DripTime=1.8
    TouchDetonationDelay=0.15
    Physics=PHYS_Falling
    Mesh=Mesh'XWeapons_rc.GoopMesh'
    Skins(0)=FinalBlend'XEffectMat.RedShell'//make the goop red -eze
    DrawScale=1.2
    AmbientGlow=80
    bProjTarget=false
    CollisionRadius=2
    CollisionHeight=2
    SoundRadius=100
    SoundVolume=255
    LifeSpan=6.0
    bUnlit=true
    RemoteRole=ROLE_SimulatedProxy
    bNetTemporary=false
    ExplodeSound=Sound'WeaponSounds.BioRifle.BioRifleGoo1'
    ImpactSound=Sound'WeaponSounds.BioRifle.BioRifleGoo2'
    bDynamicLight=true
    LightType=LT_Steady
    LightEffect=LE_QuadraticNonIncidence
    LightBrightness=190
    LightHue=4//82 -eze
    LightSaturation=10
    LightRadius=0.6
    bSwitchToZeroCollision=true
}
```

StickyFire Class

The Stickyfire class is in fact, our napalm. It's the fire that we will *stick* to another pawn in the game.
It's been placed in it's own class for 2 reasons.

- Is for good object orientation of the code…
- It makes it real easy to have it do damage to the pawn as it now has it's own "thread" of execution and specific functionality.

This class was originally derived form pickup because it made the most sense. If you touch the flaming goo, you "*pickup"* the flames.
However, we don't want to spawn anything into the players inventory, after all. It's not an item per-say.

This is really easy, don't set the InventoryType and it never adds to the pawns inventory.

This was changed due to errors - there is possibility of an error because there's only a limited number of pickup objects allowed.
The solution… change the parent class to actor.
This also solves any inventory problems that could be incurred by defining it as a pickup.

Spawn
intrinsic(278) final function actor Spawn(class<actor>
SpawnClass, optional actor SpawnOwner, optional
name SpawnTag, optional vector SpawnLocation,
optional rotator SpawnRotation);

Spawn an actor. Returns an actor of the specified class, not of class Actor (this is hard coded in the compiler). Returns None if the actor could not be spawned (either the actor wouldn't fit in the specified location, or the actor list is full). Defaults to spawning at the spawner's location.

```
class StickyFire extends Actor;

var pawn pother;   //who we're stuck to
var HitFlameBig BFlame; // the fire animation
var int burnDamage;  // damage per tick
```

This does nothing more than bind the fire to the pawn & changes states.

```
function ToTorch(actor toTorch, optional vector hitloc)
{
    pother=pawn(toTorch);

    if(hitloc != vect(0,0,0))
        SetLocation(hitloc);
    gotostate('FlameOn');
}
```

Anyway, this 'FlameOn' state is made up of only 2 functions…Beginstate & Timer.

BeginState simply creates the fire effect, binds it to the player, sets how long it burns and the timer resolution.

SetTimer
intrinsic(280) final function SetTimer(float NewTimerRate, bool bLoop);
Causes Timer() events every NewTimerRate seconds.

```
state FlameOn
{
    function Beginstate()
    {
```

```
                BFlame=spawn(class'HitFlameBig',,,,);
                 BFlame.setlocation(pother.Location);
                       BFlame.Setbase(pother);
                 Bflame.default.LifeSpan=lifespan+2;
                   SetTimer(1,True);//one sec intervals
    }
    function Timer()
    {
        if( BFlame==none)
        {
            destroy();//flame has gone out...
        }
        //deal damage
        pother.TakeDamage(burnDamage, Instigator, pother.location,
vect(0,0,0),
                          class'DamTypeBoomStick');
    }
}



defaultproperties
{
    burnDamage=3
    lifespan=4
    bHidden=true
}
```

The Damage Type

We now have our damage class which is specific to this projectile.

```
class DamTypeBoomStick extends WeaponDamageType;

defaultproperties
{   //death messages to the player
    DeathString="%o bought %k's S-Mart special."
                MaleSuicide="%o ate his own boomstick"
               FemaleSuicide="%o ate her own boomstick"
                //what weapon class causes this message
    WeaponClass=class'BoomStick'
}
```

The Pickup

How the item appears in the world before pickup & what item it spawns into the player's inventory

```
class BoomStickPickup extends UTWeaponPickup;

DefaultProperties
{   //item created in inventory
    InventoryType=class'BoomStick'
    //pickup message displayed to the player
    PickupMessage="BoomStick!!!"
    PickupSound=Sound'PickupSounds.FlakCannonPickup'
    PickupForce="BoomStickPickup"  // jdf

                         MaxDesireability=+0.7
    //mesh & draw type to use
```

```
    StaticMesh=StaticMesh'WeaponStaticMesh.LinkGunPickup'
    DrawType=DT_StaticMesh
    DrawScale=0.75
}
```

The Attachment Class

This handles the visual aspects of the weapon in third person.

```
class BoomAttach extends BioAttachment;


simulated event ThirdPersonEffects()
{
    local Rotator R;

    if ( Level.NetMode != NM_DedicatedServer && FlashCount > 0 )
    {
        if (MuzFlash3rd == None)
        {
            MuzFlash3rd = Spawn(class'XEffects.RocketMuzFlash3rd');
            MuzFlash3rd.bHidden = false;
            AttachToBone(MuzFlash3rd, 'tip');
        }
        if (MuzFlash3rd != None)
        {
            R.Roll = Rand(65536);
            SetBoneRotation('Bone_Flash', R, 0, 1.0);
            MuzFlash3rd.mStartParticles++;
        }
    }

    Super.ThirdPersonEffects();
}


defaultproperties
{
   //how the weapon appears in 3rd person
   Mesh=mesh'Weapons.LinkGun_3rd'
}
```

The int File

Add this line to the int file, in the public section:

```
Object=(Class=Class,MetaClass=Engine.Weapon,Name=CVGPackage.BoomStick
, Description=" boomstick... Shop smart..  Shop S-mart.")
```

This will allow the weapon along with a description to be displayed in the menu.

For the ammo pickups, we can set messages in our .int file:

[BoomAmmo]
ItemName="Boomstick shells"

[BoomAmmoPickup]
PickupMessage="You picked up some BOOM-stick ammo"

Making sure it Works

Load up a game and then bring down the console using the ‘ key and then type the following:

```
Summon CVGpackage.BoomStickPickup
```