Project Sirius - Source Code

Document Compiled by Jay Shunta Igarashi

Table of Contents

Color Code

Black - Code written by Team Sirius
Gray - Code taken from Unreal Tournament 2004
Blue - Comments
Green - Special import commands
Red - Code/descriptions that cannot be expressed via comments

*Note: Code assumes that a version of Unreal Tournament 2004 with certain bonus packs is installed

Sirius.uc

```
class Sirius extends xDeathMatch;

#exec AUDIO IMPORT FILE="..\Sounds\Sirius\MissionAccomplished.wav" NAME="MissionAccomplished" GROUP="Mission"
#exec AUDIO IMPORT FILE="..\Sounds\Sirius\MissionFailed.wav" NAME="MissionFailed" GROUP="Mission"

var bool bGoalReached;
var sound EndGameSound[2];
var(LoadingHints) private localized array<string> SiriusHints;

// No longer adds enemy bots at start of the match
function bool AddBot(optional string botName)
{
	return true;
}

// The player is forced to the Remmy mesh and name is changed to Remmy
function SetPlayerDefaults(Pawn PlayerPawn)
{
	Super.SetPlayerDefaults(PlayerPawn);
	Remmy(PlayerPawn).SetSiriusMesh();
	xPawn(PlayerPawn).bFrozenBody=false;
	ChangeName(PlayerPawn.Controller, default.DefaultPlayerName, false );
}

// Play voice "Mission Accomplished"/"Mission Failed" on game end
function PlayEndOfMatchMessage()
{
	local controller C;
	for ( C = Level.ControllerList; C != None; C = C.NextController )
	{
		if ( C.IsA('RemmyController') )
		{
			if (bGoalReached)
				PlayerController(C).PlayAnnouncement(EndGameSound[0],1,true);
			else
				PlayerController(C).PlayAnnouncement(EndGameSound[1],1,true);
		}
	}
}

// Retrieve hint for loading screen
// Based off function in DeathMatch
static function string GetNextLoadHint( string MapName )
{
	local array<string> Hints;
	// Removed code that prioritizes hints of DeathMatch
	Hints = GetAllLoadHints(true);
	if ( Hints.Length > 0 )
		return Hints[Rand(Hints.Length)];
	return "";
}

// Returns the array of SiriusHints
```

Sirius.uc

```
// Based off function in DeathMatch
static function array<string> GetAllLoadHints(optional bool bThisClassOnly)
{
	local int i;
	local array<string> Hints;

	for ( i = 0; i < default.SiriusHints.Length; i++ )
		Hints[Hints.Length] = default.SiriusHints[i];
	return Hints;
}

// Based off state in DeathMatch
State MatchInProgress
{
	function Timer()
	{
		local Controller P;

		Global.Timer();
		if ( !bFinalStartup )
		{
			bFinalStartup = true;
			PlayStartupMessage();
		}
		if ( bForceRespawn )
			For ( P=Level.ControllerList; P!=None; P=P.NextController )
			{
				if ( (P.Pawn == None) && P.IsA('PlayerController')
&& !P.PlayerReplicationInfo.bOnlySpectator )
					PlayerController(P).ServerReStartPlayer();
			}
		if ( NeedPlayers() && AddBot() && (RemainingBots > 0) )
			RemainingBots--;

		// Code for  decrementing time limit and processing time overs removed

		ElapsedTime++;
		GameReplicationInfo.ElapsedTime = ElapsedTime;
	}

	function beginstate()
	{
		local PlayerReplicationInfo PRI;

		ForEach DynamicActors(class'PlayerReplicationInfo',PRI)
			PRI.StartTime = 0;
		ElapsedTime = 0;
		bWaitingToStartMatch = false;
		StartupStage = 5;
		PlayStartupMessage();
		StartupStage = 6;
	}
}
```

Sirius.uc

```
defaultproperties
{
        // References to voices
        EndGameSound(0)=Sound'zwolf.Mission.MissionAccomplished'
        EndGameSound(1)=Sound'zwolf.Mission.MissionFailed'
        // Array of hints for Sirius gametype
        SiriusHints(1)="You can make an extra jump while in mid-air."
        SiriusHints(2)="While in the air, you can make a dash by tapping the direction keys twice rapidly."
        SiriusHints(3)="You can dodge off a wall in mid-air by pushing towards a wall and pressing the jump key."
        SiriusHints(4)="Your freeze ray has a slow recharge rate but it can keep an enemy in place for a long time."
        SiriusHints(5)="You can stand on top of enemies that you have immobilized with the freeze ray."
        SiriusHints(6)="Your alternate fire is the breath blowback where enemies are pushed away from you."
        SiriusHints(7)="You can use the breath on a frozen enemy to move them around."
        SiriusHints(8)="You can play taunts or other voice messages through the voice menu by
pressing %SPEECHMENUTOGGLE%."
        SiriusHints(9)="While crouching (by holding down %DUCK%), you cannot fall off a ledge."
        // Adjustments to make the game begin immediately
        bQuickStart=True
        CountDown=0
        // Set default classes to be used
        DefaultPlayerClassName="zwolf.Remmy"
        HUDType="zwolf.HUDSirius"
        PlayerControllerClassName="zwolf.RemmyController"
        // Define properties of the gametype
        MapPrefix="Sirius"
        DefaultPlayerName="Remmy"
        // Change the displays shown in the Unreal menu
        GameName="Sirius"
        Description="A plague is spreading through the city of Arcas and Remmy needs to escape the city because he is being
suspected of infection."
}
```

Remmy.uc

```
class Remmy extends xPawn;

var bool moveForward, moveBack, moveLeft, moveRight;
var float WallJumpSpeedZ, WallJumpSpeedFactor;
var Mesh SiriusMesh;
var Material SiriusHeadSkin, SiriusBodySkin, SiriusFrozenHeadSkin, SiriusFrozenBodySkin;
var class<xPawnSoundGroup> SiriusVoice;
var int RegenWaitTime;
var int LastDamageTime;
var class<Emitter> RegenEmitterClass;
var Emitter RegenEmitter;

// Set the mesh, textures and voice pack for Remmy
function SetSiriusMesh()
{
	LinkMesh(SiriusMesh);
	Skins[0]=SiriusBodySkin;
	Skins[1]=SiriusHeadSkin;
	SoundGroupClass=SiriusVoice;
}

// Make initial view third person
simulated function bool PointOfView()
{
	return true;
}

// Obtain input direction
simulated function RawInput(float DeltaTime, float aBaseX, float aBaseY, float aBaseZ, float aMouseX, float aMouseY, float
aForward, float aTurn, float aStrafe, float aUp, float aLookUp)
{
	moveForward = aForward > 0;
	moveBack = aForward < 0;
	moveLeft = aStrafe < 0;
	moveRight = aStrafe > 0;
}

// Function to determine wall jump
// Based off Dodge() in xPawn
function bool DoWallJump()
{
	local eDoubleClickDir DoubleClickMove;
	local vector X,Y,Z, TraceStart, TraceEnd, Dir, Cross, HitLocation, HitNormal;
	local Actor HitActor;
	local rotator TurnRot;

	// Pretend input is the reverse of a double tap direction
	DoubleClickMove = DCLICK_None;
	if(moveForward)
		DoubleClickMove = DCLICK_Back;
	else if(moveBack)
		DoubleClickMove = DCLICK_Forward;
	else if(moveLeft)
```

```
            DoubleClickMove = DCLICK_Right;
        else if(moveRight)
            DoubleClickMove = DCLICK_Left;

        TurnRot.Yaw = Rotation.Yaw;
        GetAxes(TurnRot,X,Y,Z);

        // No need to test for Physics == PHYS_Falling
        if ( !bCanWallDodge )
            return false;
        if (DoubleClickMove == DCLICK_Forward)
            TraceEnd = -X;
        else if (DoubleClickMove == DCLICK_Back)
            TraceEnd = X;
        else if (DoubleClickMove == DCLICK_Left)
            TraceEnd = Y;
        else if (DoubleClickMove == DCLICK_Right)
            TraceEnd = -Y;
        TraceStart = Location - CollisionHeight*Vect(0,0,1) + TraceEnd*CollisionRadius;
        TraceEnd = TraceStart + TraceEnd*32.0;
        HitActor = Trace(HitLocation, HitNormal, TraceEnd, TraceStart, false, vect(1,1,1));
        if ( (HitActor == None) || (!HitActor.bWorldGeometry && (Mover(HitActor) == None)) )
            return false;
        if (DoubleClickMove == DCLICK_Forward)
        {
            Dir = X;
            Cross = Y;
        }
        else if (DoubleClickMove == DCLICK_Back)
        {
            Dir = -1 * X;
            Cross = Y;
        }
        else if (DoubleClickMove == DCLICK_Left)
        {
            Dir = -1 * Y;
            Cross = X;
        }
        else if (DoubleClickMove == DCLICK_Right)
        {
            Dir = Y;
            Cross = X;
        }
        if ( AIController(Controller) != None )
            Cross = vect(0,0,0);
        PerformWallJump(DoubleClickMove, Dir,Cross);
        return true;
}

// Animation sequence of wall jump
// Based off PerformDodge() in xPawn
function bool PerformWallJump(eDoubleClickDir DoubleClickMove, vector Dir, vector Cross)
{
```

```
        local float VelocityZ;
        local name Anim;

        if ( Physics == PHYS_Falling )
        {
                if (DoubleClickMove == DCLICK_Forward)
                        Anim = WallDodgeAnims[0];
                else if (DoubleClickMove == DCLICK_Back)
                        Anim = WallDodgeAnims[1];
                else if (DoubleClickMove == DCLICK_Left)
                        Anim = WallDodgeAnims[2];
                else if (DoubleClickMove == DCLICK_Right)
                        Anim = WallDodgeAnims[3];

                if ( PlayAnim(Anim, 1.0, 0.1) )
                        bWaitForAnim = true;
                AnimAction = Anim;

                TakeFallingDamage();
                // All instances of DodgeSpeedZ replaced WallJumpSpeedZ
                if (Velocity.Z < -WallJumpSpeedZ*0.5)
                        Velocity.Z += WallJumpSpeedZ*0.5;
        }

        VelocityZ = Velocity.Z;
        // DodgeSpeedFactor replaced with WallJumpSpeedFactor
        Velocity = WallJumpSpeedFactor*GroundSpeed*Dir + (Velocity Dot Cross)*Cross;

        if ( !bCanDodgeDoubleJump )
                MultiJumpRemaining = 0;
        if ( bCanBoostDodge || (Velocity.Z < -100) )
                Velocity.Z = VelocityZ + WallJumpSpeedZ;
        else
                Velocity.Z = WallJumpSpeedZ;

        CurrentDir = DoubleClickMove;
        SetPhysics(PHYS_Falling);
        PlayOwnedSound(GetSound(EST_Dodge), SLOT_Pain, GruntVolume,,80);
        return true;
}

// Based off function in xPawn
function DoDoubleJump( bool bUpdating )
{
        local vector X,Y,Z,tempVel;
        local rotator TurnRot;
        PlayDoubleJump();

        if ( !bIsCrouched && !bWantsToCrouch )
        {
                if ( !IsLocallyControlled() || (AIController(Controller) != None) )
                        MultiJumpRemaining -= 1;
```

```
                // Define or limit the speed on the double jump
                TurnRot.Yaw = Rotation.Yaw;
                GetAxes(TurnRot,X,Y,Z);

                if(moveForward || moveBack || moveLeft || moveRight)
                {
                        Velocity.X = 0;
                        Velocity.Y = 0;
                        Velocity.Z = 0;
                        if(moveForward)
                                Velocity += GroundSpeed*X;
                        if(moveBack)
                                Velocity += GroundSpeed*-X;
                        if(moveLeft)
                                Velocity += GroundSpeed*-Y;
                        if(moveRight)
                                Velocity += GroundSpeed*Y;
                }
                else
                {
                        Velocity.X = 0;
                        Velocity.Y = 0;
                        Velocity.Z = 0;
                }

                tempVel.X = Velocity.X;
                tempVel.Y = Velocity.Y;
                tempVel.Z = 0;

                if(VSize(tempVel) > GroundSpeed)
                        Velocity = (GroundSpeed/VSize(tempVel))*tempVel;

                Velocity.Z = JumpZ + MultiJumpBoost;

                SetPhysics(PHYS_Falling);
                if ( !bUpdating )
                        PlayOwnedSound(GetSound(EST_DoubleJump), SLOT_Pain, GruntVolume,,80);
        }
}

// Based off function in xPawn
function bool DoJump( bool bUpdating )
{
        // Attempts to wall jump if possible
        if (!bUpdating && (Physics == PHYS_Falling) && DoWallJump() )
                return true;
        // Attempts to double jump
        // No longer tests whether vertical speed is less than 100 UU/s (At top of jump)
        if ( !bUpdating && Super.CanDoubleJump()&& IsLocallyControlled() )
        {
                if ( PlayerController(Controller) != None )
                        PlayerController(Controller).bDoubleJump = true;
                DoDoubleJump(bUpdating);
```

```
                MultiJumpRemaining -= 1;
                return true;
        }
        if ( Super.DoJump(bUpdating) )
        {
                if ( !bUpdating )
                        PlayOwnedSound(GetSound(EST_Jump), SLOT_Pain, GruntVolume,,80);
                return true;
        }
        return false;
}

// Function to determine air dash
// Based off function in xPawn
function bool Dodge(eDoubleClickDir DoubleClickMove)
{
        local vector X,Y,Z, Dir, Cross;
        local rotator TurnRot;

        // No longer allows Physics != PHYS_Walking
        if ( bIsCrouched || bWantsToCrouch || (Physics != PHYS_Falling) )
                return false;

        TurnRot.Yaw = Rotation.Yaw;
        GetAxes(TurnRot,X,Y,Z);

        // Removed test for wall behind player in dodge direction

        if (DoubleClickMove == DCLICK_Forward)
        {
                Dir = X;
                Cross = Y;
        }
        else if (DoubleClickMove == DCLICK_Back)
        {
                Dir = -1 * X;
                Cross = Y;
        }
        else if (DoubleClickMove == DCLICK_Left)
        {
                Dir = -1 * Y;
                Cross = X;
        }
        else if (DoubleClickMove == DCLICK_Right)
        {
                Dir = Y;
                Cross = X;
        }
        if ( AIController(Controller) != None )
                Cross = vect(0,0,0);
        return PerformDodge(DoubleClickMove, Dir,Cross);
}
```

Remmy.uc

```
// Animation sequence for air dash
// Based off function in xPawn
function bool PerformDodge(eDoubleClickDir DoubleClickMove, vector Dir, vector Cross)
{
	local float VelocityZ;
	local name Anim;

	// Switched from Physics == PHYS_Falling
	// Will always be false
	if ( Physics != PHYS_Falling )
	{
		if (DoubleClickMove == DCLICK_Forward)
			Anim = WallDodgeAnims[0];
		else if (DoubleClickMove == DCLICK_Back)
			Anim = WallDodgeAnims[1];
		else if (DoubleClickMove == DCLICK_Left)
			Anim = WallDodgeAnims[2];
		else if (DoubleClickMove == DCLICK_Right)
			Anim = WallDodgeAnims[3];

		if ( PlayAnim(Anim, 1.0, 0.1) )
			bWaitForAnim = true;
		AnimAction = Anim;

		TakeFallingDamage();
		if (Velocity.Z < -DodgeSpeedZ*0.5)
			Velocity.Z += DodgeSpeedZ*0.5;
	}

	VelocityZ = Velocity.Z;
	Velocity = DodgeSpeedFactor*GroundSpeed*Dir + (Velocity Dot Cross)*Cross;

	if ( !bCanDodgeDoubleJump )
		MultiJumpRemaining = 0;
	if ( bCanBoostDodge || (Velocity.Z < -100) )
		Velocity.Z = VelocityZ + DodgeSpeedZ;
	else
		Velocity.Z = DodgeSpeedZ;

	CurrentDir = DoubleClickMove;
	SetPhysics(PHYS_Falling);
	PlayOwnedSound(GetSound(EST_Dodge), SLOT_Pain, GruntVolume,,80);
	return true;
}

// Adjusts the rotation of the pawn
simulated function FaceRotation( rotator NewRotation, float DeltaTime )
{
	if ( Physics == PHYS_Ladder )
		SetRotation(OnLadder.Walldir);
	else
	{
		// Removed the negation of NewRotation.Pitch
```

```
                SetRotation(NewRotation);
        }
}

// No longer calculates damage upon landing from heights
function TakeFallingDamage()
{
}

// Based off function in Pawn
// Overrides function in xPawn
// To eliminate death animation using ragdoll karma (which caused a crash)
function Died(Controller Killer, class<DamageType> damageType, vector HitLocation)
{
        local Trigger T;
        local NavigationPoint N;

        if ( bDeleteMe || Level.bLevelChange || Level.Game == None )
                return; // already destroyed, or level is being cleaned up

        if ( DamageType.default.bCausedByWorld && (Killer == None || Killer == Controller) && LastHitBy != None )
                Killer = LastHitBy;

        // mutator hook to prevent deaths
        // WARNING - don't prevent bot suicides - they suicide when really needed
        if ( Level.Game.PreventDeath(self, Killer, damageType, HitLocation) )
        {
                Health = max(Health, 1); //mutator should set this higher
                return;
        }
        Health = Min(0, Health);

        // Removed code that makes the player throw weapon upon death

        if ( DrivenVehicle != None )
        {
                Velocity = DrivenVehicle.Velocity;
                DrivenVehicle.DriverDied();
        }

        if ( Controller != None )
        {
                Controller.WasKilledBy(Killer);
                Level.Game.Killed(Killer, Controller, self, damageType);
        }
        else
                Level.Game.Killed(Killer, Controller(Owner), self, damageType);

        DrivenVehicle = None;

        if ( Killer != None )
                TriggerEvent(Event, self, Killer.Pawn);
        else
```

```
                TriggerEvent(Event, self, None);

        // make sure to untrigger any triggers requiring player touch
        if ( IsPlayerPawn() || WasPlayerPawn() )
        {
                PhysicsVolume.PlayerPawnDiedInVolume(self);
                ForEach TouchingActors(class'Trigger',T)
                        T.PlayerToucherDied(self);

                // event for HoldObjectives
                //for ( N=Level.NavigationPointList; N!=None; N=N.NextNavigationPoint )
                //      if ( N.bStatic && N.bReceivePlayerToucherDiedNotify )
                ForEach TouchingActors(class'NavigationPoint', N)
                        if ( N.bReceivePlayerToucherDiedNotify )
                                N.PlayerToucherDied( Self );
        }

        // remove powerup effects, etc.
        RemovePowerups();

        Velocity.Z *= 1.3;
        if ( IsHumanControlled() )
                PlayerController(Controller).ForceDeathUpdate();
        if ( (DamageType != None) && DamageType.default.bAlwaysGibs )
                ChunkUp( Rotation, DamageType.default.GibPerterbation );
        else
        {
                NetUpdateFrequency = Default.NetUpdateFrequency;
                PlayDying(DamageType, HitLocation);
                if ( Level.Game.bGameEnded )
                        return;
                if ( !bPhysicsAnimUpdate && !IsLocallyControlled() )
                        ClientDying(DamageType, HitLocation);
        }

}

// Based off function in xPawn
simulated function PlayDyingAnimation(class<DamageType> DamageType, vector HitLoc)
{
        // Removed all code processing ragdoll death
        // non-ragdoll death fallback
        Velocity += TearOffMomentum;
        BaseEyeHeight = Default.BaseEyeHeight;
        SetTwistLook(0, 0);
        SetInvisibility(0.0);
        // Use custom animation function instead
        PlayDeathType(DamageType, HitLoc);
        SetPhysics(PHYS_Falling);
}

// Function to substitute PlayDirectionalDeath() in xPawn
// Allows DamageType to be passed as a parameter
```

Remmy.uc

```
// Partially based off PlayDirectionalDeath() in xPawn
simulated function PlayDeathType(class<DamageType> DamageType, vector HitLoc)
{
	local Vector X,Y,Z, Dir;

	// If killed by FreezeBolt
	if(DamageType.name == 'FreezeDamType')
	{
		// Simulate Remmy frozen & remove gibbing
		PlayAnim('Hit_Head');
		StopAnimating();
		Skins[0]=SiriusFrozenBodySkin;
		Skins[1]=SiriusFrozenHeadSkin;
		bFrozenBody=true;
	}
	// If killed by anything else (e.g. melee attacks)
	else
	{
		// Code from PlayDirectionalDeath() in xPawn
		GetAxes(Rotation, X,Y,Z);
		HitLoc.Z = Location.Z;

		// random
		if ( VSize(Velocity) < 10.0 && VSize(Location - HitLoc) < 1.0 )
		{
			Dir = VRand();
		}
		// velocity based
		else if ( VSize(Velocity) > 0.0 )
		{
			Dir = Normal(Velocity*Vect(1,1,0));
		}
		// hit location based
		else
		{
			Dir = -Normal(Location - HitLoc);
		}

		if ( Dir Dot X > 0.7 || Dir == vect(0,0,0))
			PlayAnim('DeathB',, 0.2);
		else if ( Dir Dot X < -0.7 )
			PlayAnim('DeathF',, 0.2);
		else if ( Dir Dot Y > 0 )
			PlayAnim('DeathL',, 0.2);
		else if ( HasAnim('DeathR') )
			PlayAnim('DeathR',, 0.2);
		else
			PlayAnim('DeathF',, 0.2);
	}
}

// Stores the last time Remmy takes a hit
```

Remmy.uc

```
simulated function TakeDamage( int Damage, Pawn InstigatedBy, Vector Hitlocation, Vector Momentum, class<DamageType>
damageType)
{
	LastDamageTime = Sirius(Level.Game).ElapsedTime;
	Super.TakeDamage(Damage, InstigatedBy, Hitlocation, Momentum, damageType);
}

// Tick function called every frame
simulated function Tick(float DeltaTime)
{
	// Regenerates 1 HP per frame if haven't been damaged for a while
	// Spawns a ShieldRecover emitter
	if(Health > 0 && Health < default.HealthMax && Sirius(Level.Game).ElapsedTime - LastDamageTime >=
RegenWaitTime)
	{
		if(RegenEmitter == None)
			RegenEmitter = spawn(RegenEmitterClass,,,self.Location);
		else
			RegenEmitter.SetLocation(self.Location);
		Health++;
	}
	else if(RegenEmitter != None)
		RegenEmitter.Destroy();
	Super.Tick(DeltaTime);
}

defaultproperties
{
	// New parameters for wall jumps to distinguish from air dash
	WallJumpSpeedZ=540.000000
	WallJumpSpeedFactor=1.500000
	// Defines enforced mesh and skins for Remmy
	SiriusMesh=SkeletalMesh'ZRemmy.Remmy'
	SiriusHeadSkin=Texture'ZSiriusSkins.Skins.RemmyFaceSkin'
	SiriusBodySkin=Texture'ZSiriusSkins.Skins.RemmyBodySkin'
	SiriusFrozenHeadSkin=Texture'ZSiriusSkins.Skins.RemmyFaceSkinFreeze'
	SiriusFrozenBodySkin=Texture'ZSiriusSkins.Skins.RemmyBodySkinFreeze'
	SiriusVoice=Class'XGame.xMercMaleSoundGroup'
	// Parameters for regeneration effects
	RegenWaitTime=15
	RegenEmitterClass=Class'zwolf.ShieldRecover'
	// Removes any Unreal shield capabilities
	ShieldStrengthMax=0.000000
	// Define default equipment
	RequiredEquipment(0)="zwolf.FreezeRay"
	RequiredEquipment(1)=
	// Change jump height
	JumpZ=540.000000
	// Limit camera angles to not be extreme
	PitchUpLimit=8000
	PitchDownLimit=59153
	// Set absolute maximum health
	SuperHealthMax=100.000000
```

```
        // Change speed factor of air dash
        DodgeSpeedFactor=1.750000
}
```

RemmyController.uc

```
class RemmyController extends xPlayer;

// Raises the camera after camera position calculation
// To allow the aiming reticule to be above Remmy
function PlayerCalcView(out actor ViewActor, out vector CameraLocation, out rotator CameraRotation )
{
	local Vector HitLocation, HitNormal;
	local Actor Other;
	local Vector tempCameraLocation;
	// Calculate position of camera the way Unreal deals with it
	super.PlayerCalcView(ViewActor, CameraLocation, CameraRotation);
	// Add 32 UU in the vertical direction
	// Make sure it is always at least 10 UU below the ceiling
	tempCameraLocation = CameraLocation;
	tempCameraLocation.Z += 42.0;
	Other = Trace(HitLocation, HitNormal, tempCameraLocation, CameraLocation, false);
	if(Other != None && Other.bWorldGeometry)
	{
		CameraLocation = HitLocation;
		CameraLocation.Z -= 10.0;
	}
	else
		CameraLocation.Z += 32.0;
}

defaultproperties
{
}
```

ShieldRecover.uc

```
class ShieldRecover extends Spiral;

// Emitter for regeneration
// Currently is identical to Spiral

defaultproperties
{
}
```

FreezablePawn.uc

```
class FreezablePawn extends Monster
	config(user);

	var name BeforeFreezeState;
	var Controller FrozenController;

	var(FreezablePawn) float LongFreezeTime; //make it accessible in the editor
	var(FreezablePawn) float ShortFreezeTime;
	var float FreezeTime;
	var(FreezablePawn) int Shield;

	var(FreezablePawn) class<Emitter> FreezeEmitter;
	var(FreezablePawn) Texture FreezeTexture;
	var Emitter IceEmitter;

	var(FreezablePawn) Sound FreezeSound;
	Var(FreezablePawn) Sound UnfreezeSound;

	var(FreezablePawn) bool bStopOnFreeze;

	var(FreezablePawn) float RangedRefireRate;
	var float HoldFireTime;

	var float OldAirSpeed;

// Called just after this class spawned
event PostBeginPlay()
{
	Super.PostBeginPlay();
	// Set projectiles fired to be FreezeBolt
	MyAmmo.ProjectileClass = class'zwolf.FreezeBolt';
}

// Process actions of enemies when attacked
function TakeDamage(int Damage, Pawn instigatedBy, Vector hitlocation, Vector momentum, class<DamageType>
damageType)
{
	local vector HitNormal;

	HitNormal = Normal(HitLocation - Location);

	// If hit by a FreezeBolt
	if(damageType.name == 'FreezeDamType')
	{
		if(!(self.GetStateName() == 'Frozen'))
		{
			//determine freeze time and update shield
			if(Shield > 0)
			{
				FreezeTime = ShortFreezeTime;
				Shield--;
				//display a shield graphic maybe?
			}
```

FreezablePawn.uc

```
                        else
                        {
                                FreezeTime = LongFreezeTime;
                        }

                        //Set the freeze overlay
                        if (DamageType.default.DamageOverlayMaterial != None)
                        {
                                SetOverlayMaterial( DamageType.default.DamageOverlayMaterial, FreezeTime, false );
                                //true is the override...
                        }

                        //Spawn a FreezeFlakes emitter
                        if (FreezeEmitter != None)
                        {
                                IceEmitter = spawn(FreezeEmitter,,,(HitLocation+HitNormal), Rotator(HitNormal));
                                SetTimer(0.1,true);
                        }
                                BeforeFreezeState = self.GetStateName();
                                GotoState('Frozen');
                        }
        }
        else{
                SetPhysics(PHYS_Falling); //only seems to work with falling.
                if ( (Physics == PHYS_None) && (Momentum.Z < 0) )
                        Momentum.Z *= -1; //don't know what this does.
                Velocity += 3 * momentum/(Mass + 200); //without, makes like he's going to fall but doesn't move
        }
}

// Timer function
function Timer()
{
        // Respawn the FreezeFlakes emitter
        // In case the enemy is relocated while frozen
        if(IceEmitter != None && (IceEmitter.Location != self.Location))
        {
                IceEmitter.Kill();
                IceEmitter = spawn(FreezeEmitter,,,self.Location);
        }
}

// State when an enemy is frozen
state Frozen
{
        event BeginState()
        {
                // Set properties when this enemy freezes
                PlaySound(FreezeSound);
                Skins[0] = FreezeTexture;
                Skins[1] = FreezeTexture;
                bCanBeBaseForPawns=true;
                if(bStopOnFreeze)
```

FreezablePawn.uc

```
		{
			Velocity.X=0;
			Velocity.Y=0;
			Velocity.Z=0;
		}
	}

	event EndState()
	{
		// Set properties when this enemy unfreezes
		// Spawns FreezeShatter emitter
		SetTimer(0.0,false); //stops the emitter timer
		IceEmitter.Kill();
		Spawn(class'zwolf.FreezeShatter',,,self.Location);
		PlaySound(UnfreezeSound);
		Skins[0] = default.Skins[0];
		Skins[1] = default.Skins[1];
		bCanBeBaseForPawns=false;
	}

Begin:
	// Stops animation and movement of enemy for a given period of time
	bPhysicsAnimUpdate = false;
	FreezableController(Controller).SleepNow(FreezeTime);
	self.GroundSpeed = 0;
	StopAnimating();
	Sleep( FreezeTime );
	// Undo above processes
	self.GroundSpeed = default.GroundSpeed;
	bPhysicsAnimUpdate = true;
	GotoState( BeforeFreezeState );
}

// Sets the time limit between shots
function bool CheckFire()
{
	if(HoldFireTime>0)
	{
		return false;
	}
	HoldFireTime = RangedRefireRate;
	return true;
}

// Tick function called every frame
simulated function Tick(float DeltaTime)
{
	// Decrements limit of time between shots
	if(HoldFireTime > 0)
		HoldFireTime = (HoldFireTime - DeltaTime);
	super.Tick(DeltaTime);
}
```

FreezablePawn.uc

```
defaultproperties
{
	// Set effects of freezing
	FreezeEmitter=Class'zwolf.FreezeFlakes'
	FreezeTexture=Texture'ZSiriusSkins.Skins.Freeze'
	FreezeSound=Sound'ZSiriusSounds.Frostbolt.FreezeSound'
	UnfreezeSound=Sound'ZSiriusSounds.Frostbolt.UnfreezeSound'
	// Set refire rate
	RangedRefireRate=2.000000
	// Don't leave a gun at spawn point
	bNoDefaultInventory=True
	// Set controller to custom
	ControllerClass=Class'zwolf.FreezableController'
}
```

FreezeFlakes.uc

```
class FreezeFlakes extends WaterRing
	placeable;

// Emitter while enemy frozen
// Currently near identical to WaterRing

defaultproperties
{
	// Changing emitter properties
	AutoDestroy=False
	AutoReset=True
}
```

FreezeShatter.uc

```
class FreezeShatter extends NewExplosionA;

// Emitter when enemy unfrozen
// Animations defined in defaultproperties

defaultproperties
{
	ExplosionTextures(0)=Texture'Chapter10.Smoke'
	ExplosionTextures(1)=Texture'Chapter10.Smoke'
	Begin Object Class=SpriteEmitter Name=SpriteEmitter1
		RespawnDeadParticles=False
		SpinParticles=True
		UniformSize=True
		AutomaticInitialSpawning=False
		BlendBetweenSubdivisions=True
		MaxParticles=3
		StartLocationShape=PTLS_Sphere
		SphereRadiusRange=(Min=24.000000,Max=24.000000)
		StartSpinRange=(X=(Max=1.000000))

	StartSizeRange=(X=(Min=50.000000,Max=70.000000),Y=(Min=50.000000,Max=70.000000),Z=(Min=50.000000,Max=7
0.000000))
		InitialParticlesPerSecond=10.000000
		Texture=Texture'Chapter10.Smoke'
		TextureUSubdivisions=4
		TextureVSubdivisions=4
		SecondsBeforeInactive=0.000000
		LifetimeRange=(Min=0.400000,Max=0.600000)
	End Object
	Emitters(0)=SpriteEmitter'XEffects.NewExplosionA.SpriteEmitter1'

	Begin Object Class=SpriteEmitter Name=SpriteEmitter0
		UseColorScale=True
		RespawnDeadParticles=False
		SpinParticles=True
		UseSizeScale=True
		UseRegularSizeScale=False
		UniformSize=True
		AutomaticInitialSpawning=False
		BlendBetweenSubdivisions=True
		UseRandomSubdivision=True
		UseVelocityScale=True
		Acceleration=(Z=20.000000)
		ColorScale(0)=(Color=(B=255,G=255,R=255))
		ColorScale(1)=(RelativeTime=0.125000,Color=(B=255,G=255,R=255))
		ColorScale(2)=(RelativeTime=0.330000,Color=(B=255,G=255,R=255,A=255))
		ColorScale(3)=(RelativeTime=0.750000,Color=(B=128,G=128,R=128,A=255))
		ColorScale(4)=(RelativeTime=1.000000,Color=(B=64,G=64,R=64))
		MaxParticles=15
		StartLocationShape=PTLS_Polar
		StartLocationPolarRange=(Y=(Min=-32768.000000,Max=32768.000000),Z=(Min=10.000000,Max=10.000000))
		UseRotationFrom=PTRS_Actor
		RotationOffset=(Yaw=-16384)
```

```
		SpinsPerSecondRange=(X=(Max=0.100000))
		StartSpinRange=(X=(Max=1.000000))
		SizeScale(0)=(RelativeSize=0.200000)
		SizeScale(1)=(RelativeTime=1.000000,RelativeSize=0.500000)
		InitialParticlesPerSecond=500.000000
		DrawStyle=PTDS_AlphaBlend
		Texture=Texture'Chapter10.Smoke'
		TextureUSubdivisions=4
		TextureVSubdivisions=4
		LifetimeRange=(Min=2.000000,Max=2.000000)
		StartVelocityRadialRange=(Min=200.000000,Max=200.000000)
		GetVelocityDirectionFrom=PTVD_AddRadial
		VelocityScale(0)=(RelativeVelocity=(X=1.000000,Y=1.000000,Z=1.000000))
		VelocityScale(1)=(RelativeTime=0.300000,RelativeVelocity=(X=0.100000,Y=0.100000,Z=0.100000))
		VelocityScale(2)=(RelativeTime=1.000000)
	End Object
	Emitters(1)=SpriteEmitter'XEffects.NewExplosionA.SpriteEmitter0'

	Begin Object Class=SpriteEmitter Name=SpriteEmitter2
		FadeOut=True
		RespawnDeadParticles=False
		SpinParticles=True
		UseSizeScale=True
		UseRegularSizeScale=False
		UniformSize=True
		AutomaticInitialSpawning=False
		FadeOutStartTime=0.050000
		MaxParticles=1
		StartLocationShape=PTLS_Sphere
		SphereRadiusRange=(Min=2.000000,Max=2.000000)
		StartSpinRange=(X=(Max=1.000000))
		SizeScale(0)=(RelativeSize=0.700000)
		SizeScale(1)=(RelativeTime=1.000000,RelativeSize=1.000000)

	StartSizeRange=(X=(Min=110.000000,Max=120.000000),Y=(Min=110.000000,Max=120.000000),Z=(Min=80.000000,M
ax=90.000000))
		InitialParticlesPerSecond=10.000000
		Texture=Texture'Chapter10.Smoke'
		SecondsBeforeInactive=0.000000
		LifetimeRange=(Min=0.300000,Max=0.300000)
	End Object
	Emitters(2)=SpriteEmitter'XEffects.NewExplosionA.SpriteEmitter2'

	AutoDestroy=True
	LightType=LT_FadeOut
	LightEffect=LE_QuadraticNonIncidence
	LightHue=28
	LightSaturation=90
	LightBrightness=255.000000
	LightRadius=9.000000
	LightPeriod=32
	LightCone=128
	bNoDelete=False
```

FreezeShatter.uc

```
        bDynamicLight=True
}
```

FreezableController.uc

```
class FreezableController extends MonsterController;

var float SleepTime;
var name PreSleepState;
var Pawn BeforeSleepPawn;

// Function before entering the Asleep state
function SleepNow(float HowLong)
{
	PreSleepState = self.GetStateName();
	SleepTime = HowLong;
	GotoState('Asleep');
}

// Based off function in Monster
// Changed to not spawn on top of player
function FightEnemy(bool bCanCharge)
{
	local vector X,Y,Z;
	local float enemyDist;
	local float AdjustedCombatStyle, Aggression;
	local bool bFarAway, bOldForcedCharge;

	if ( (Enemy == None) || (Pawn == None) )
		log("HERE 3 Enemy "$Enemy$" pawn "$Pawn);

	if ( (Enemy == FailedHuntEnemy) && (Level.TimeSeconds == FailedHuntTime) )
	{
		if ( !Enemy.Controller.bIsPlayer )
			FindNewEnemy();

		if ( Enemy == FailedHuntEnemy )
		{
			GoalString = "FAILED HUNT - HANG OUT";
			if ( EnemyVisible() )
				bCanCharge = false;
			// Removed code to spawn to a PlayerStart
			if ( !EnemyVisible() )
			{
				WanderOrCamp(true);
				return;
			}
		}
	}

	bOldForcedCharge = bMustCharge;
	bMustCharge = false;
	enemyDist = VSize(Pawn.Location - Enemy.Location);
	AdjustedCombatStyle = CombatStyle;
	Aggression = 1.5 * FRand() - 0.8 + 2 * AdjustedCombatStyle
				+ FRand() * (Normal(Enemy.Velocity - Pawn.Velocity) Dot Normal(Enemy.Location -
Pawn.Location));
	if ( Enemy.Weapon != None )
```

```
		Aggression += 2 * Enemy.Weapon.SuggestDefenseStyle();
	if ( enemyDist > MAXSTAKEOUTDIST )
		Aggression += 0.5;
	if ( (Pawn.Physics == PHYS_Walking) || (Pawn.Physics == PHYS_Falling) )
	{
		if (Pawn.Location.Z > Enemy.Location.Z + TACTICALHEIGHTADVANTAGE)
			Aggression = FMax(0.0, Aggression - 1.0 + AdjustedCombatStyle);
		else if ( (Skill < 4) && (enemyDist > 0.65 * MAXSTAKEOUTDIST) )
		{
			bFarAway = true;
			Aggression += 0.5;
		}
		else if (Pawn.Location.Z < Enemy.Location.Z - Pawn.CollisionHeight) // below enemy
			Aggression += CombatStyle;
	}

	if ( !EnemyVisible() )
	{
		GoalString = "Enemy not visible";
		if ( !bCanCharge )
		{
			GoalString = "Stake Out";
			DoStakeOut();
		}
		else
		{
			GoalString = "Hunt";
			GotoState('Hunting');
		}
		return;
	}

	// see enemy - decide whether to charge it or strafe around/stand and fire
	Target = Enemy;
	if( Monster(Pawn).PreferMelee() || (bCanCharge && bOldForcedCharge) )
	{
		GoalString = "Charge";
		DoCharge();
		return;
	}

	if ( bCanCharge && (Skill < 5) && bFarAway && (Aggression > 1) && (FRand() < 0.5) )
	{
		GoalString = "Charge closer";
		DoCharge();
		return;
	}

	if ( !Monster(Pawn).PreferMelee() && (FRand() > 0.17 * (skill - 1)) && !DefendMelee(enemyDist) )
	{
		GoalString = "Ranged Attack";
		DoRangedAttackOn(Enemy);
		return;
```

FreezableController.uc

```
	}

	if ( bCanCharge )
	{
		if ( Aggression > 1 )
		{
			GoalString = "Charge 2";
			DoCharge();
			return;
		}
	}

	if ( !Pawn.bCanStrafe )
	{
		GoalString = "Ranged Attack";
		DoRangedAttackOn(Enemy);
		return;
	}

	GoalString = "Do tactical move";
	if ( !Monster(Pawn).RecommendSplashDamage() && Monster(Pawn).bCanDodge && (FRand() < 0.7) &&
(FRand()*Skill > 3) )
	{
		GetAxes(Pawn.Rotation,X,Y,Z);
		GoalString = "Try to Duck ";
		if ( FRand() < 0.5 )
		{
			Y *= -1;
			TryToDuck(Y, true);
		}
		else
			TryToDuck(Y, false);
	}
	DoTacticalMove();
}

// Controller state for when frozen
state Asleep
{
	ignores NotifyLanded, ReceiveWarning;

	event EndState()
	{
		Pawn = BeforeSleepPawn;
	}

Begin:
	// Disconnect Pawn until unfrozen
	BeforeSleepPawn = Pawn;
	Pawn = none;
	Sleep(SleepTime);
	GotoState( PreSleepState );
}
```

FreezableController.uc

```
defaultproperties
{
}
```

FreezableSkaarj.uc

```
class FreezableSkaarj extends FreezablePawn;

Identical to Skaarj Class except as follows:

// Skaarj function to fire projectiles
function SpawnTwoShots()
{
	local vector X,Y,Z, FireStart;
	local rotator FireRotation;

	GetAxes(Rotation,X,Y,Z);
	FireStart = GetFireStart(X,Y,Z);
	if ( !SavedFireProperties.bInitialized )
	{
		SavedFireProperties.AmmoClass = MyAmmo.Class;
		SavedFireProperties.ProjectileClass = MyAmmo.ProjectileClass;
		SavedFireProperties.WarnTargetPct = MyAmmo.WarnTargetPct;
		SavedFireProperties.MaxRange = MyAmmo.MaxRange;
		SavedFireProperties.bTossed = MyAmmo.bTossed;
		SavedFireProperties.bTrySplash = MyAmmo.bTrySplash;
		SavedFireProperties.bLeadTarget = MyAmmo.bLeadTarget;
		SavedFireProperties.bInstantHit = MyAmmo.bInstantHit;
		SavedFireProperties.bInitialized = true;
	}
	FireRotation = Controller.AdjustAim(SavedFireProperties,FireStart,600);
	Spawn(MyAmmo.ProjectileClass,,,FireStart,FireRotation);

	// Removed code to spawn a second projectile
}

function GetFireStart(vector X, vector Y, vector Z)
	removed
	// Function used to deviate to the side to allow room for the second projectile

defaultproperties
{
	// Textures for pig guard
	// Currently using placeholders taken from Skaarj textures
	FreezeTexture=Texture'ZSiriusSkins.Skins.PigSkinFreeze'
	Skins(0)=Texture'ZSiriusSkins.Skins.PigSkin'
}
```

FreezableSkaarjPupae.uc

```
class FreezableSkaarjPupae extends FreezablePawn;

Identical to SkaarjPupae class except as follows:

defaultproperties
{
	// Textures for crawler bot
	// Currently using placeholders taken from Skaarj Pupae textures
	FreezeTexture=Texture'ZSiriusSkins.Skins.CrawlerSkinFreeze'
	Skins(0)=Texture'ZSiriusSkins.Skins.CrawlerSkin'
}
```

FreezableGasbag.uc

```
class FreezableGasbag extends FreezablePawn;

Identical to Gasbag except as follows:

defaultproperties
{
	// To totally prevent movement when frozen
	bStopOnFreeze=True
}
```

TriggerSiriusEnd.uc

```
class TriggerSiriusEnd extends Trigger
	placeable;

// Function called when touched by any other actor
function Touch(actor Other)
{
	super.Touch(Other);
	// On a Sirius map upon contact with Remmy (not dead)
	if(IsRelevant( Other ) && Level.Game.IsA('Sirius') && Other.IsA('Remmy') && Remmy(Other).health > 0)
	{
		// End the game via a trigger
		Sirius(Level.Game).bGoalReached = true;
		Level.Game.EndGame(None, "triggered");
	}
}

defaultproperties
{
	// Set Trigger properties
	TriggerType=TT_LivePlayerProximity
	bTriggerOnceOnly=True
}
```

FreezeRay.uc

```
class FreezeRay extends LinkGun;

// Charge bar for main fire - FreezeFire
simulated function float ChargeBar()
{
	local float temp;
	temp = 1.0-FMax(0,(FireMode[0].NextFireTime - Level.TimeSeconds)/FireMode[0].FireRate);
	return temp;
}

// Charge bar for alternate fire - BreathFire
simulated function float ChargeBar2()
{
	local float temp;
	temp = 1.0-FMax(0,(FireMode[1].NextFireTime - Level.TimeSeconds)/FireMode[1].FireRate);
	return temp;
}

defaultproperties
{
	// Set Weapon properties
	FireModeClass(0)=Class'zwolf.FreezeFire'
	FireModeClass(1)=Class'zwolf.BreathFire'
	bShowChargingBar=True
	ItemName="Freeze Ray"
}
```

FreezeFire.uc

```
class FreezeFire extends LinkAltFire;

// Spawn a FreezeBolt projectile
// Ignores projectile class for some stupid reason, it has to be declared directly in this function.
function Projectile SpawnProjectile(Vector Start, Rotator Dir)
{
	local ShockProjectile Proj; //used to be LinkProjectile

	Start += Vector(Dir) * 10.0 * LinkGun(Weapon).Links;
	Proj = Weapon.Spawn(Class'zwolf.FreezeBolt',,, Start, Dir);
	return Proj;
}

defaultproperties
{
	// Set WeaponFire properties
	bModeExclusive=False
	FireRate=5.000000
	AmmoPerFire=0
	ProjectileClass=Class'zwolf.FreezeBolt'
	BotRefireRate=5.000000
}
```

FreezeBolt.uc

```
class FreezeBolt extends ShockProjectile;

defaultproperties
{
	// Set Projectile properties
	Speed=1500.000000
	MaxSpeed=1500.000000
	Damage=30.000000
	DamageRadius=20.000000
	MyDamageType=Class'zwolf.FreezeDamType'
}
```

FreezeDamType.uc

```
class FreezeDamType extends DamTypeLinkPlasma;

defaultproperties
{
	// Set DamageType properties
	bCausesBlood=False
	DamageOverlayMaterial=Shader'XGameShaders.PlayerShaders.WeaponUDamageShader'
}
```

BreathFire.uc

```
class BreathFire extends ShockBeamFire;

var float BeamRadius; // The radius of each breathfire beam (remember there are 9 beams)
var bool DisplayEdgeBeams;
var int FireAmount; // How many times to fire in half a second.
var int ShotCount;

// Repeatedly call DoFireEffect of superclass
function DoFireEffect()
{
	ShotCount = FireAmount;
	SetTimer((0.5/FireAmount), true);
}

// Timer function
function Timer()
{
	// Calls DoFireEffect() of superclass for given number of shots
	if((FireAmount>0) && (ShotCount>0))
	{
		ShotCount--;
		super.DoFireEffect();
	}
}

// Fires 9 traces which trace out a cone-like shape.
function DoTrace(Vector Start, Rotator Dir)
{
	local Rotator LeftShot;
	local Rotator RightShot;
	local Rotator UpShot;
	local Rotator DownShot;
	local Rotator UpLeftShot;
	local Rotator UpRightShot;
	local Rotator DownLeftShot;
	local Rotator DownRightShot;

	// Messy code, but whatever

	RightShot = Dir;
	LeftShot = Dir;
	UpShot = Dir;
	DownShot = Dir;
	UpLeftShot = Dir;
	UpRightShot = Dir;
	DownLeftShot = Dir;
	DownRightShot = Dir;

	LeftShot.Yaw+=4096; // 45 degrees
	RightShot.Yaw-=4096; // 45 degrees in the other direction
	UpShot.Pitch+=4096;
	DownShot.Pitch-=4096;
```

```
        // In between shots
        UpLeftShot.Yaw+=2048; // Half of 45 degrees
        UpLeftShot.Pitch+=2048;

        UpRightShot.Yaw-=2048;
        UpRightShot.Pitch+=2048;

        DownLeftShot.Yaw+=2048;
        DownLeftShot.Pitch-=2048;

        DownRightShot.Yaw-=2048;
        DownRightShot.Pitch-=2048;

        DoWideTrace(Start, Dir, false);
        DoWideTrace(Start, LeftShot, DisplayEdgeBeams);
        DoWideTrace(Start, RightShot, DisplayEdgeBeams);
        DoWideTrace(Start, UpShot, DisplayEdgeBeams);
        DoWideTrace(Start, DownShot, DisplayEdgeBeams);
        DoWideTrace(Start, UpLeftShot, false);
        DoWideTrace(Start, UpRightShot, false);
        DoWideTrace(Start, DownLeftShot, false);
        DoWideTrace(Start, DownRightShot, false);
}

// This is exactly the same as the instant fire's trace, except the extent parameter is bigger.
// So the beam width is about twice the weapon holder's collision radius. Also, doesn't display
// the beam and doesn't block projectiles.
function DoWideTrace(Vector Start, Rotator Dir, bool DisplayBeam)
{
        local Vector X, End, HitLocation, HitNormal, RefNormal;
        local Actor Other;
        local int Damage;
        local bool bDoReflect;
        local int ReflectNum;
        local Vector BeamExtent;
        local Vector actualMomentum;

        MaxRange();

                ReflectNum = 0;
        while (true)
        {
                bDoReflect = false;
                X = Vector(Dir);
                End = Start + TraceRange * X;

                // Calculating area of effect of beam
                BeamExtent = BeamRadius * vect(1,1,1);
                Other = Weapon.Trace(HitLocation, HitNormal, End, Start, true, BeamExtent);

                if ( Other != None && (Other != Instigator || ReflectNum > 0) )
                {
```

BreathFire.uc

```
                        if (bReflective && Other.IsA('xPawn') && xPawn(Other).CheckReflect(HitLocation, RefNormal,
DamageMin*0.25))
                        {
                                bDoReflect = true;
                                HitNormal = Vect(0,0,0);
                        }
                        else if ( !Other.bWorldGeometry && !Other.IsA('Projectile') )
                        {
                                Damage = DamageMin;
                                if ( (DamageMin != DamageMax) && (FRand() > 0.5) )
                                        Damage += Rand(1 + DamageMax - DamageMin);
                                Damage = Damage * DamageAtten;

                                // Update hit effect except for pawns (blood) other than vehicles.
                                if ( Other.IsA('Vehicle') || (!Other.IsA('Pawn') && !Other.IsA('HitScanBlockingVolume')) )
                                        WeaponAttachment(Weapon.ThirdPersonActor).UpdateHit(Other, HitLocation,
HitNormal);
                                // Hack to adjust momentum values
                                actualMomentum = Momentum*X;
                                actualMomentum.Z = 0;
                                Other.TakeDamage(Damage, Instigator, HitLocation, actualMomentum, DamageType);
                                HitNormal = Vect(0,0,0);
                        }
                        else if ( WeaponAttachment(Weapon.ThirdPersonActor) != None )
                                WeaponAttachment(Weapon.ThirdPersonActor).UpdateHit(Other,HitLocation,HitNormal);
                }
                else
                {
                        HitLocation = End;
                        HitNormal = Vect(0,0,0);
                        WeaponAttachment(Weapon.ThirdPersonActor).UpdateHit(Other,HitLocation,HitNormal);
                }

                // Display or hide visible beams depending on parameter
                if(DisplayBeam == true)
                        SpawnBeamEffect(Start, Dir, HitLocation, HitNormal, ReflectNum);

                if (bDoReflect && ++ReflectNum < 4)
                {
                        //Log("reflecting off"@Other@Start@HitLocation);
                        Start = HitLocation;
                        Dir = Rotator(RefNormal); //Rotator( X - 2.0*RefNormal*(X dot RefNormal) );
                }
                else
                {
                        break;
                }
        }
}

defaultproperties
{
        // Hide beams by default (Turn on for debugging)
```

BreathFire.uc

```
        DisplayEdgeBeams=False
        // Properties of BreathFire
        BeamRadius=70.000000
        FireAmount=20
        // Set Weapon properties
        DamageMin=0
        DamageMax=0
        TraceRange=500.000000
        Momentum=200000.000000
        bModeExclusive=False
        FireRate=2.000000
        AmmoPerFire=0
}
```