FPS Multiplayer Server UE4

Gameplay Server

Prototipo multiplayer FPS con servidor dedicado en UE4

.Controles

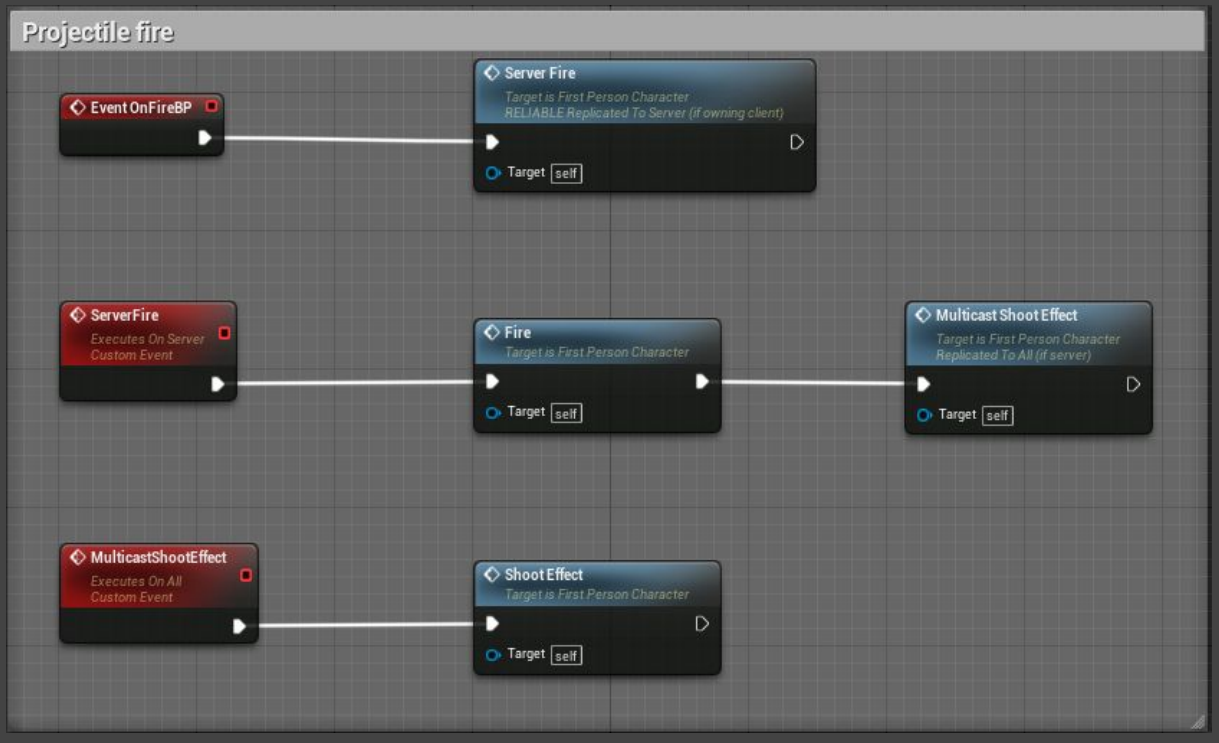
A, W, S, D	Desplazamiento horizontal
SPACE	Salto
LEFT MOUSE	Disparo
RIGHT MOUSE	Habilidad ofensiva especial.

.Sección 1

.Disparo del proyectil

El Character llama a **ServerFire()** en el **servidor** para crear un actor de tipo **proyectil** en la dirección en la que se está apuntando.

A continuación para activar los efectos de disparo en los clientes se hace una llamada **multicast** a **MulticastShootEffect()**. Las acciones realizadas en los **clientes** remotos son puramente cosméticas.



.Primera y tercera persona

Contamos con dos modelos de personaje distintos. Si somos **cliente local** se muestra el modelo de **primera persona** (FP_Mesh). Para el resto de clientes se visualizará un modelo básico de **tercera persona** (Mesh heredado de Pawn).

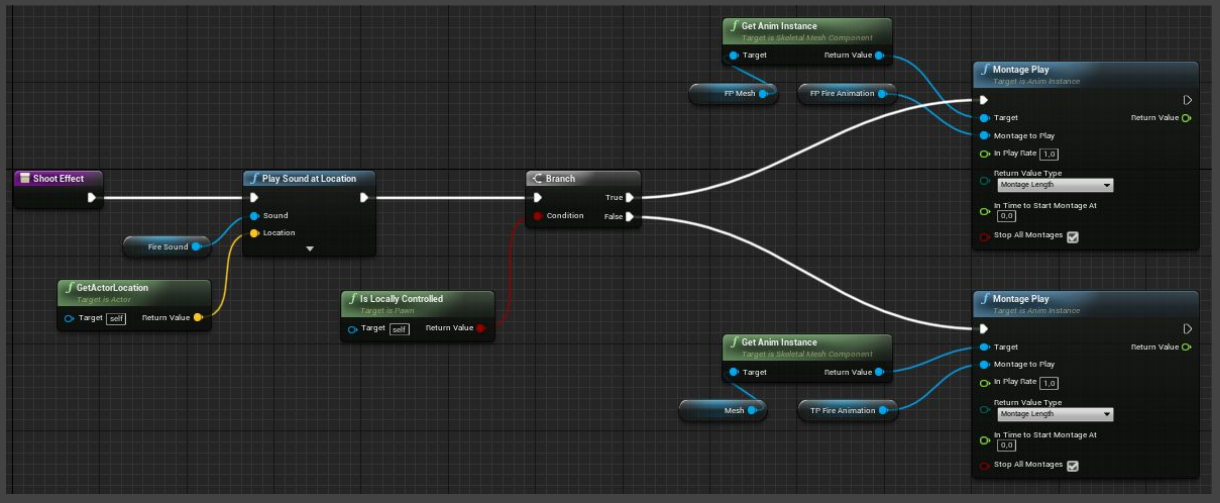
También tenemos dos **montajes de animación** distintos que ejecutaremos en el efecto de disparo dependiendo de si somos cliente local o no.

(Por simplicidad no se hace blending de las animaciones de andar y disparar)
(Tampoco existe Blend Space para el apuntado en el modelo de tercera persona)

.Efectos de disparo

Las animaciones de los personajes y los sonidos de disparo son ejecutados únicamente en los **clientes** remotos.

En la función multicast de efecto de disparo comprobamos si el personaje está **controlado localmente** para ejecutar la animación correspondiente.



.Sección 2

.Componente de vida

Se ha creado el componente de vida BMHealthComponent.
La única propiedad replicada es **CurrentHealth** y tiene activado **RepNotify** para los clientes.

```
/** The player's current health */
UPROPERTY(ReplicatedUsing = OnRep_CurrentHealth)
float CurrentHealth;
```

Para desacoplar la funcionalidad de vida y que sea reutilizable entre cualquier tipo de actor que pueda recibir daño se hace un **registro** a **eventos** de delegados en **dos direcciones** en la creación del actor.

1. El personaje crea el componente de vida y se registra al evento **OnHealthChange** (opcional)

```
HealthComp->OnHealthChangeDelegate.AddDynamic(this, &ABMGameplayServerCharacter::HealthChange);
```

2. El componente de vida se registra automáticamente al evento de recibir daño de su owner **OnTakeAnyDamage** para actualizar la vida siempre que sea necesario.

```
Owner->OnTakeAnyDamage.AddDynamic(this, &UBMHealthComponent::HandleTakeAnyDamage);
```

De esta manera siempre que el actor reciba algún tipo de daño el componente de vida será notificado automáticamente para realizar las gestiones de vida. A su vez, el propio componente hará **broadcast** notificando de cambios en la vida que el actor **owner**, en caso de que considere oportuno, puede escuchar y gestionar por su cuenta.

(Más adelante veremos cómo se utilizan estos eventos para hacer daño al jugador o para actualizar el HUD de vida)

Por último destacar que únicamente el **servidor** tiene **autoridad** para modificar el valor de la vida. Siempre que hay una actualizacion de vida se hace **Broadcast()** del evento.

```
void UBMHealthComponent::SetCurrentHealth(float healthValue)
{
    if (GetOwnerRole() == ROLE_Authority)   // We only modify health on the server
    {
        CurrentHealth = FMath::Clamp(healthValue, 0.f, MaxHealth);  // Impossible to set CurrentHealth to an invalid value
        OnHealthUpdate();   // This is necessary because the server will not recieve the RepNotify
    }
}
```

.Sección 3

.Daño

El disparo daña al jugador en el evento **OnComponentHit**. El **servidor** es el único que tiene **autoridad** para hacer daño. Se ha utilizado el sistema genérico de daño de UE4 llamando a la función **TakeDamage** de la clase Actor.

```
void ABMGameplayServerProjectile::OnHit(UPrimitiveComponent* HitComp, AActor* OtherActor, UPrimitiveCom
{
    if (GetLocalRole() == ROLE_Authority)
    {
        if ((OtherActor != NULL) && (OtherActor != this) && (GetInstigator() != OtherActor))
        {
            FDamageEvent DamageEvent;
            OtherActor->TakeDamage(Damage, DamageEvent, GetInstigatorController(), GetInstigator());
            Destroy(true, true);
        }
    }
```

Esta función lanzará el evento **OnTakeAnyDamage** y gracias a que el componente de vida está registrado ejecutará el método **HandleTakeAnyDamage** que dañará al actor en cuestión. Cualquier actor que disponga del componente de vida recibirá daño de forma transparente.

(Se ha dejado el perfil de colisión “Projectile” por defecto pero lo suyo sería implementar nuestros propios canales de colisión)

.Muerte

El concepto de muerte es **responsabilidad** del propio **Character** por lo que no está incluido en el componente de vida.

Como hemos comentado anteriormente los jugadores están registrados a los eventos de cambio de vida y llaman a **HealthChange()**. Cuando el **servidor** detecta que la **vida es 0** marca al jugador como muerto con **bDeath = true**

```cpp
void ABMGameplayServerCharacter::HealthChange()
{
    if (GetLocalRole() == ROLE_Authority)
    {
        // If player has died time to respawn
        if (HealthComp->GetCurrentHealth() <= 0 && !bDeath)
        {
            bDeath = true;

            // After 10 sec respawn
            FTimerHandle respawnTimer;
            GetWorldTimerManager().SetTimer<ABMGameplayServerCharacter>
                (respawnTimer, this, &ABMGameplayServerCharacter::Respawn, 3.0f, false);
        }
    }
```

Por motivos de seguridad, ya que la muerte de los jugadores es muy delicada, he decidido **replicar** la variable **bDeath** y avisar a los clientes remotos con RepNotify.

```cpp
/** Death control. */
UPROPERTY(ReplicatedUsing = OnRep_Death, VisibleAnywhere, BlueprintReadOnly)
bool bDeath;

/** RepNotify for death: activate ragdoll / respawn */
UFUNCTION()
void OnRep_Death();
```

Al morir se activan físicas de **Ragdoll** para los modelos en tercera persona de los clientes y un **“modo muerte”** también para el cliente local que ha muerto.

(La posición de la cámara de muerte no está muy allá)

.Respawn

El **servidor** lanza un **timer** cuando el jugador muere y pasado **RespawnTime** llama a la función **Respawn()**

```
// After 10 sec respawn
FTimerHandle respawnTimer;
GetWorldTimerManager().SetTimer<ABMGameplayServerCharacter>
    (respawnTimer, this, &ABMGameplayServerCharacter::Respawn, RespawnTime, false);
```

Para **evitar exploits** la función Respawn comprueba que tenemos la **autoridad** necesaria para ejecutarla.

Se utiliza el **navmesh** para obtener un punto aleatorio del mapa.

```
void ABMGameplayServerCharacter::Respawn()
{
    if (GetLocalRole() == ROLE_Authority)
    {
        // Respawn at random navmesh location
        UNavigationSystemV1* navSys = UNavigationSystemV1::GetCurrent(GetWorld());
        FNavLocation navLocation;
        navSys->GetRandomPoint(navLocation);

        SetActorLocation(navLocation);

        // Max health
        HealthComp->RestoreHealth();

        bDeath = false;
    }
}
```

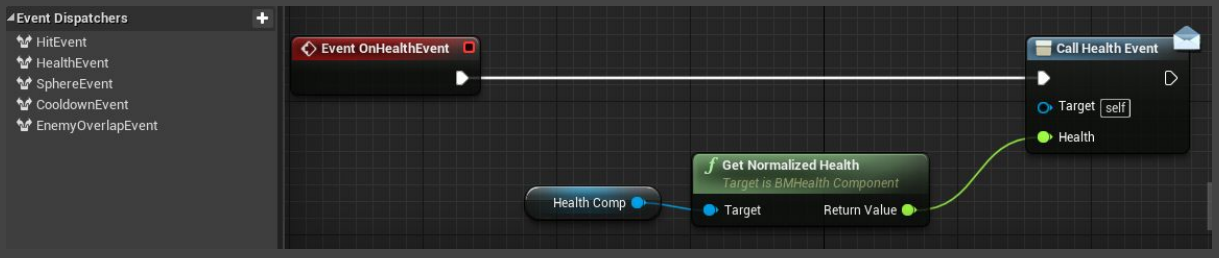
(He evitado destruir/crear el character y volver a poseerlo cada vez que muere ya que estas operaciones son las más costosas, sin embargo por motivos de seguridad en un juego real quizá sea lo más conveniente)

.Sección 4

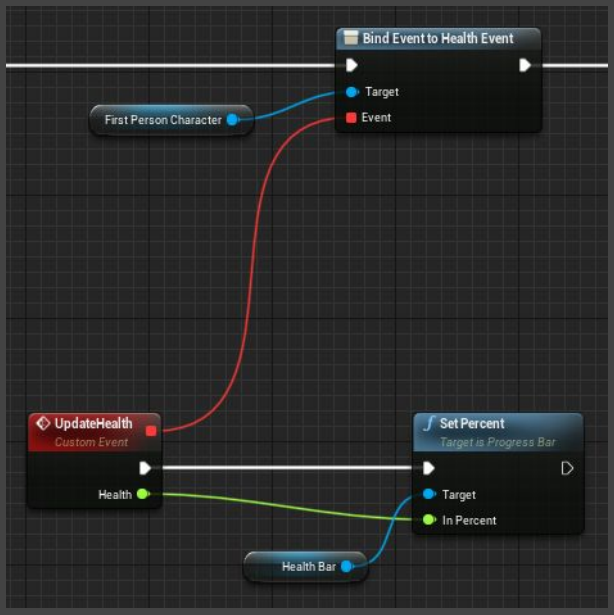
.Widget y eventos

El personaje tiene una serie de funciones del tipo **BlueprintImplementableEvent** que sirven de pasarela para llamar a unos **EventDispatchers** de **FirstPersonCharacter** que están **bindeados** en el **widget** principal para actualizar el HUD únicamente cuando recibe los eventos de actualización.

```
/** Client widget update events */

// Health change event
UFUNCTION(BlueprintImplementableEvent, meta = (DisplayName = "OnHealthEvent"))
void OnHealthEvent();

// Sphere radius event
UFUNCTION(BlueprintImplementableEvent, meta = (DisplayName = "OnSphereEvent"))
void OnSphereEvent();

// Cooldown change event
UFUNCTION(BlueprintImplementableEvent, meta = (DisplayName = "OnCooldownEvent"))
void OnCooldownEvent();

// Sphere enemy overlap event
UFUNCTION(BlueprintImplementableEvent, meta = (DisplayName = "OnEnemyOverlapEvent"))
void OnEnemyOverlapEvent();
```



Se utilizan las funciones públicas del **componente de vida** para actualizar el widget de salud.



(Por simplicidad los eventos de widget están bindeados al FirstPersonCharacter pero lo suyo sería que estuvieran en el PlayerController)

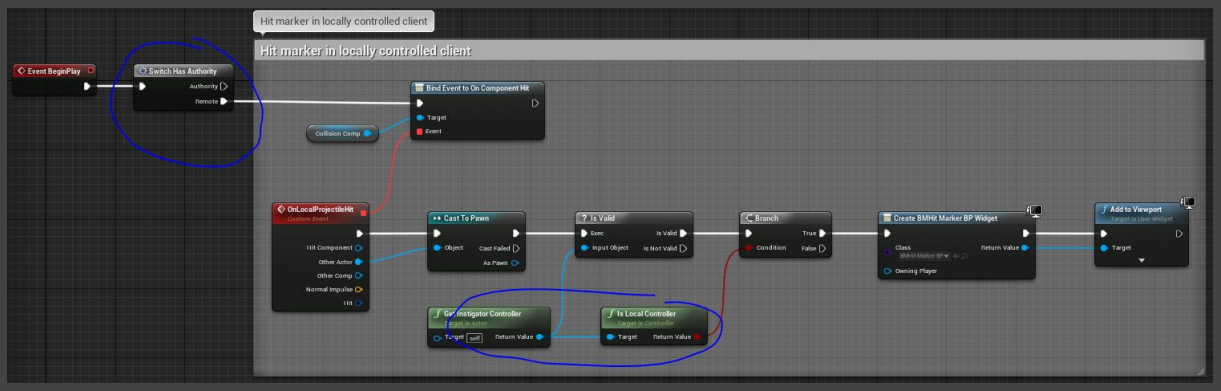
.Sección 5

.Hit marker

El **hitmarker** está implementado en blueprint en **FirstPersonProjectile**.

Al crear el proyectil comprueba si estamos en un **cliente remoto**, en ese caso bindea un evento a OnComponentHit.

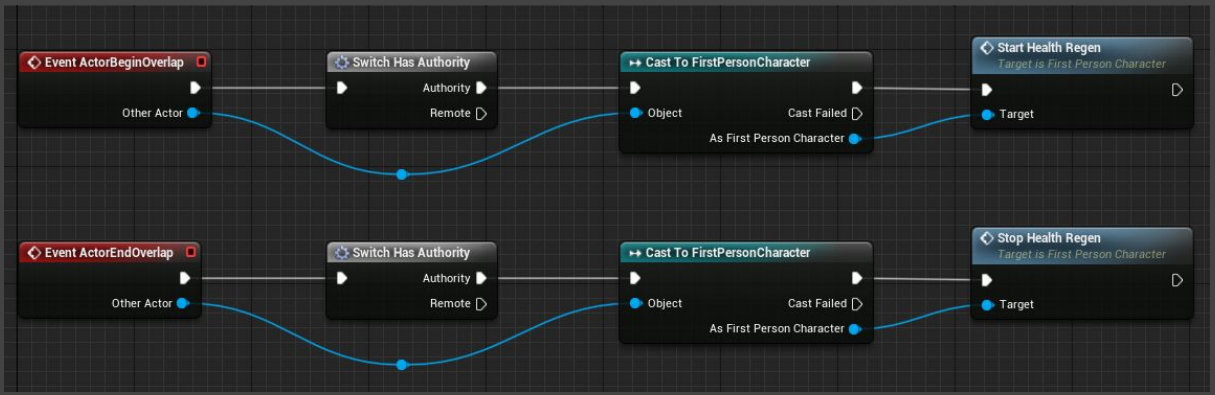
Para que únicamente se lance el hitmarker en el **cliente local** utilizo el **Instigador** del proyectil para comprobar que está localmente controlado.



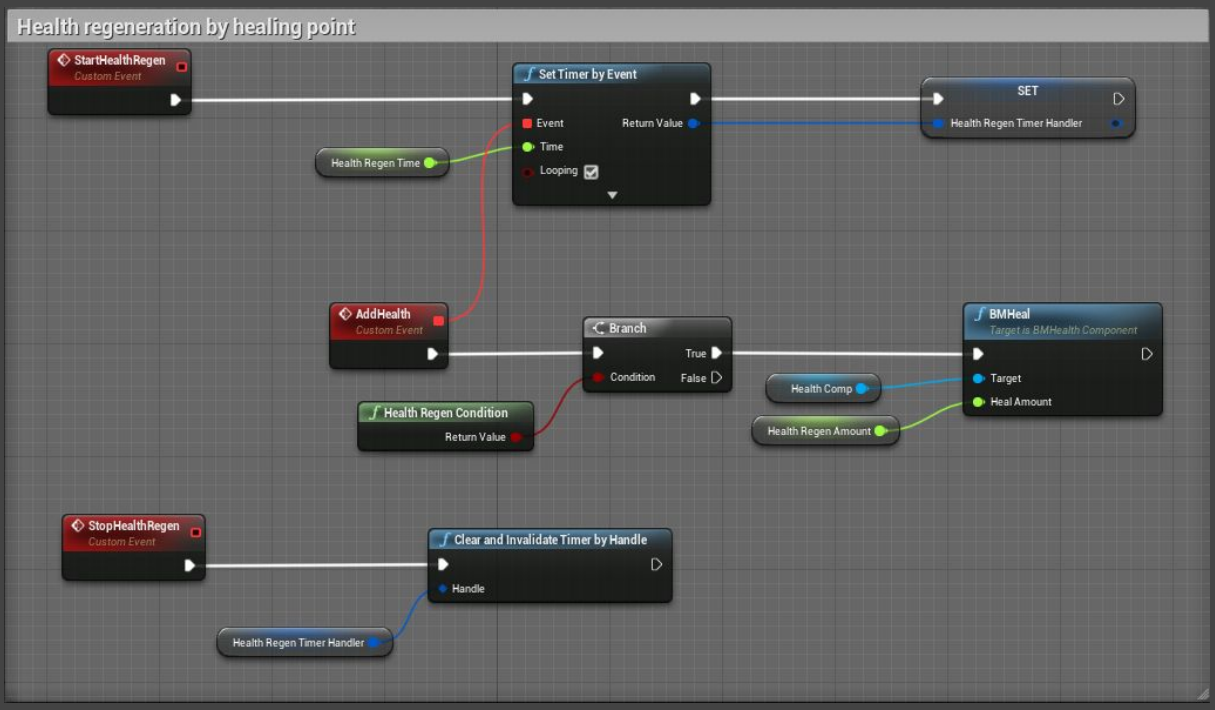
.Sección 6

.Zona de curación

La implementación de la zona de curación es bastante sencilla. En ActorBeginOverlap y ActorEndOverlap en el **servidor** se llama a las funciones **StartHealthRegen** y **StopHealthRegen** respectivamente.



Lanza un Timer en loop cada **HealthRegenTime** y llama a **Heal()** del componente de vida recuperando **HealthRegenAmount**. El componente de vida ya se encarga de poner en marcha todos los mecanismos para actualizar la vida si tenemos autoridad, replicar y notificar a los clientes remotos y actualizar el hud. Si el jugador sale de la zona de curación se invalida el Timer.



.Sección 7

.Propiedades

Las propiedades principales **replicadas** son el **radio actual** de la esfera, el tiempo de **cooldown actual** y si la habilidad está **activada**. Toda la lógica girará entorno a estas tres propiedades.

```cpp
// Replicated properties
void UBMSphereAttackComponent::GetLifetimeReplicatedProps(TArray <FLifetimeProperty>& OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);

    DOREPLIFETIME(UBMSphereAttackComponent, CurrentRadius);
    DOREPLIFETIME(UBMSphereAttackComponent, CurrentCooldown);
    DOREPLIFETIME(UBMSphereAttackComponent, Activated);
}
```

El resto de propiedades son configurables a nivel de gameplay: radio inicial, radio máximo, velocidad de crecimiento, tiempo de cooldown, cantidad de daño, etc.

.Lógica de pintado y cooldown

El componente **BMSphereAttackComponent** tiene los RCP **ServerActivateSphere**() y **ServerDeactivateSphere**(). El jugador llama al servidor al pulsar y al soltar el click derecho.
Al activar la esfera únicamente ponemos **Activated = true** para que sea el Tick del componente el que se encargue de la lógica (ya seamos servidor o clientes remotos). Comprobamos que no estemos en cooldown.

Al soltar el click derecho llamamos a **FireSpell()** y reseteamos el estado de la habilidad a su **estado de reposo** activando a su vez el **cooldown**.

```cpp
void UBMSphereAttackComponent::ActivateSphere()
{
    if (!IsInCooldown())
    {
        Activated = true;
    }
}

void UBMSphereAttackComponent::DeactivateSphere()
{
    if (Activated)
    {
        FireSpell();

        // Restore current radius
        CurrentRadius = InitialRadius;
        // Restore cooldown to max
        CurrentCooldown = Cooldown;
        // Deactivate sphere
        Activated = false;
    }
}
```

En el tick del componente el **servidor** se encarga de incrementar el radio de la esfera en caso de que esté activada y también se encarga de modificar el cooldown actual.

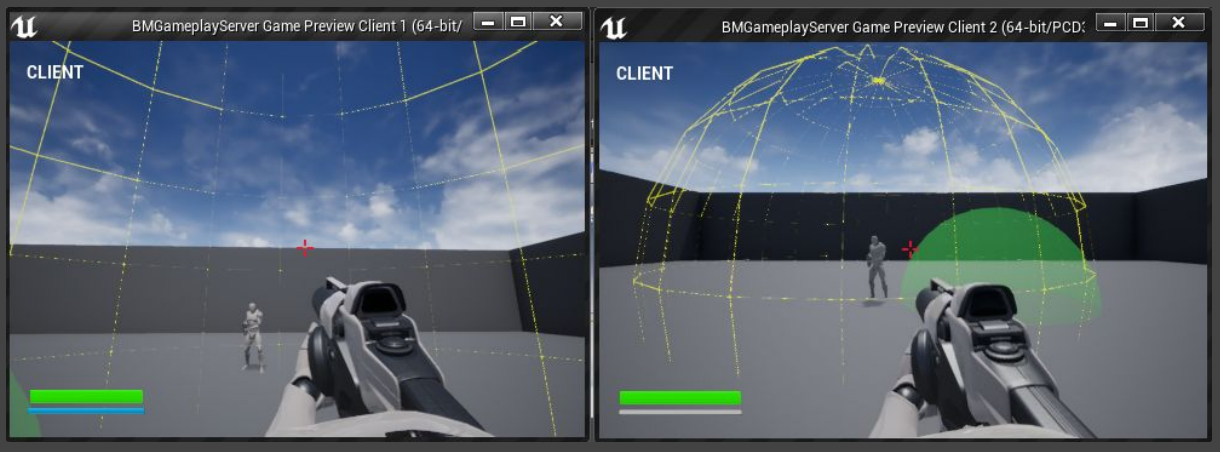
```cpp
// Called every frame
void UBMSphereAttackComponent::TickComponent(float DeltaTime, ELevelTick TickType, FActorComponentTickFunction* ThisTickFunction)
{
    Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

    // Modify radius and cooldown only on server side
    if (CharacterOwner->GetLocalRole() == ROLE_Authority)
    {
        if (Activated)
        {
            CurrentRadius += SpeedRadius * DeltaTime;
            CurrentRadius = FMath::Clamp(CurrentRadius, InitialRadius, MaxRadius);
        }

        if (IsInCooldown())
        {
            CurrentCooldown -= DeltaTime;
            CurrentCooldown = FMath::Clamp(CurrentCooldown, 0.0f, Cooldown);
        }
    }
```

La parte de **cliente** del tick es más interesante. Si la habilidad está activa pintamos una esfera de debug con el radio actual.

En caso de que seamos **cliente local** vamos a calcular qué **enemigos** están dentro de la esfera llamando a **CheckOverlapEnemies()**. Esta lógica la realizamos en el cliente ya que es meramente cosmética. El **daño** en si se realiza en el **servidor**.

```cpp
// Logic for remote clients
if (CharacterOwner->GetLocalRole() < ROLE_Authority)
{
    if (Activated)
    {
        // Draw sphere on both clients
        DrawDebugSphere(GetWorld(), CharacterOwner->GetActorLocation(), CurrentRadius, 24, FColor::Yellow, false, 0.01f, 0, 1.0f);

        // check for enemy overlap info in local client
        if (CharacterOwner->IsLocallyControlled())
        {
            int currentOverlapEnemies = CheckOverlapEnemies();
            if (NumEnemies != currentOverlapEnemies)
            {
                NumEnemies = currentOverlapEnemies;
                CharacterOwner->OnEnemyOverlapEvent();
            }
        }
    }
```

El resultado final es el siguiente:



(Por simplicidad he dejado la esfera de debug como efecto visual de la habilidad pero habría que cambiarlo ya que en shipping no se ve)

.Daño

El daño lo realiza el **servidor** con la función **FireSpell()** al soltar el click derecho del ratón.

Lanzamos un **SphereOverlapActors** y sobre la lista de **actores** resultantes le aplicamos daño con **TakeDamage**. A partir de aquí el componente de vida hace el resto.

```
void UBMSphereAttackComponent::FireSpell()
{
    if (CharacterOwner->GetLocalRole() == ROLE_Authority)
    {
        TArray<AActor*> outActors;
        TArray<AActor*> actorsToIgnore;
        actorsToIgnore.Add(CharacterOwner);
        TArray<TEnumAsByte<EObjectTypeQuery>> traceObjectTypes;
        traceObjectTypes.Add(UEngineTypes::ConvertToObjectType(ECollisionChannel::ECC_Pawn));

        if (UKismetSystemLibrary::SphereOverlapActors(GetWorld(),
            CharacterOwner->GetActorLocation(), CurrentRadius, traceObjectTypes, NULL, actorsToIgnore, outActors))
        {
            for (AActor* actor : outActors)
            {
                FDamageEvent DamageEvent;
                actor->TakeDamage(DamageAmount, DamageEvent, CharacterOwner->GetController(), CharacterOwner);
            }
        }
    }
}
```

(Me he quedado con ganas de ponerle algún efecto de sonido al lanzar la habilidad y aplicarle un impulso al jugador enemigo hacia atrás)

.Sección 8

.Widget habilidad esfera

Se ha reutilizado el widget de la vida para añadir los elementos visuales de la habilidad ofensiva de esfera. Se ha seguido el mismo sistema que el de vida utilizando **event dispatchers**.

Carga de la habilidad

Se rellena la barra azul de debajo de la vida mientras se mantiene pulsado el click derecho hasta el máximo de radio posible.

Cooldown

Al soltar el botón derecho del ratón se lanza la habilidad y comienza el cooldown. Se representa visualmente haciendo más opaca la propia barra y mostrando el contador del tiempo de cooldown en azul.

Enemigos

Se muestra un número en naranja cuando hay enemigos dentro de la esfera. Se oculta si no hay ningún enemigo dentro o la esfera está desactivada.