ENTREGA FINAL UNREAL. TANK-PROJECT IN C++.

VÍCTOR GARCÍA CORTÉS

ESNE

GRUPO 3.3 VIDEOJEUGOS. PROYECTO ALTERNATIVO EN UNREAL

17/06/2019

Víctor García Cortés

GitHub: https://github.com/VictorGar96/Tank-Project

Tabla de contenido

Introducción, objetivos motivación del Proyecto.

Para esta entrega final he optado por hacer un trabajo alternativo al planteado como entrega final. En un principio el objetivo era entregar el TFG pero debido a que estamos en una fase muy poco avanzada del proyecto y al poco tiempo que hemos tenido para avanzar en él he decidido hacer este trabajo alternativo.

El trabajo consiste en un proyecto en Unreal enfocado a afianzar los conocimientos obtenidos en las 3ª entrega que consistía en hacer un proyecto hecho mediante código en Unreal. Por eso este proyecto está hecho en su gran mayoría por código o la combinación de código y blueprint, pero como comento, he intentado hacer todo mediante código.

El objetivo o la finalidad que me he planteado con este proyecto ha sido como preparación para afrontar el TFG que va a ser el mayor desafío que he hecho hasta la fecha. Por eso con este proyecto lo he enfocado en aprender todo lo posible de Unreal para poder afrontar el TFG (el cual será hecho en Unreal). Esto a su vez ha sido la principal motivación para embarcarme a realizar un proyecto con un cierto grado de dificultad debido en su gran mayoría a su realización mediante código y a utilizar el menor número de blueprints posible. Esto también se debe a que no me gusta nada programar en blueprint, no me gusta el concepto y creo que hacerlo con código optimiza mucho el proyecto, aunque al mismo tiempo sea mucho más complicado debido a la poca documentación y se nota que se han enfocado en blueprint.

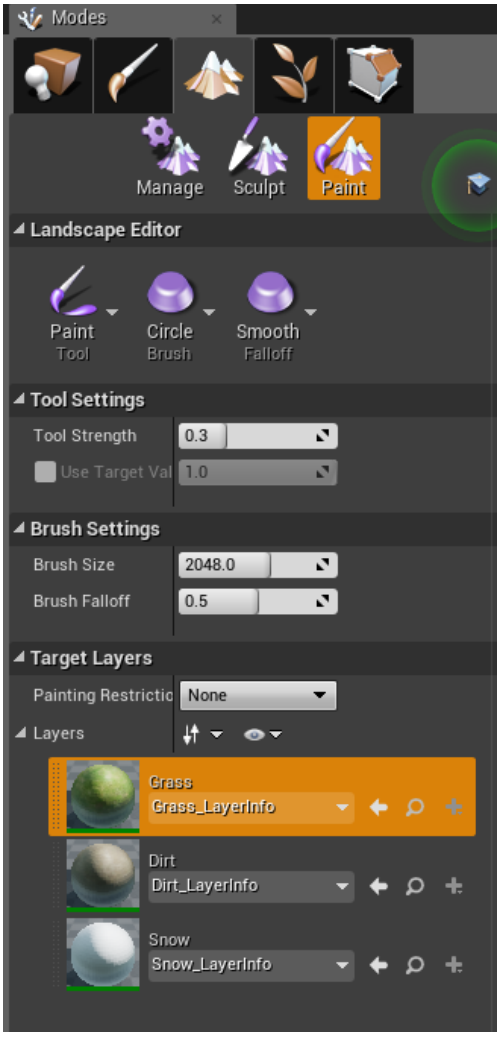
En este proyecto he combinado todo tipo de herramientas dentro de Unreal, desde crear un terreno en el cual podremos movernos libremente y controlar un tanque, otro que será controlado por una IA sencilla y al mismo tiempo utilizar el User Interface para visibilizar tanto la vida que tiene el jugador y los enemigos como el número de balas que nos quedan, así como un menú principal desde el cual podremos jugar para acceder al juego como salir. En este proyecto habrá en escena varios tanques, uno de ellos será controlado por el jugador y los demás por la IA, el jugador tendrá una vida, representado por una barra de vida en el UI, este podrá moverse por el terreno creado con las herramientas que proporciona Unreal y podrá disparar un número limitado de proyectiles. Habrá un sistema de vida provisional para mejorar la funcionalidad del juego. Por el momento podemos matar y ser matados por los tanques enemigos que tanto el propio jugador como los tanques enemigos dejarán de disparar y de moverse para indicar que han muerto. A parte en este proyecto he hecho uso del sistema que tiene implementado Unreal de AI pathfinding, sistemas de partículas, diferentes tipos de fuerzas y daños con proyectiles que son lanzados por los tanques, colisionadores, etc…

Descripción técnica de los componentes del proyecto.

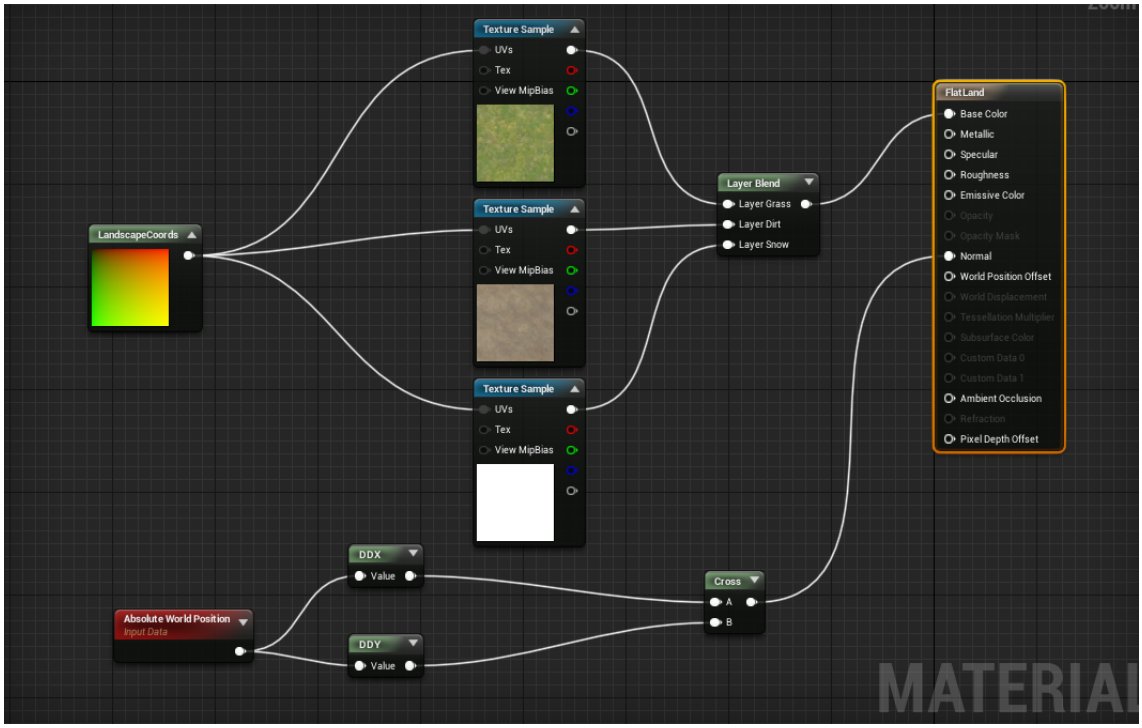
Comenzar diciendo que ha sido un proceso bastante largo, en gran parte se debe a no saber cómo iba a hacer las diferentes funcionalidades que me había planteado ni el proceso de hacerlo por lo que ha sido un trabajo de bastante investigación y de prueba/error hasta que conseguía tener algo funcional. Por tanto voy a ir por partes. Lo primero de lo que voy a hablar va a ser el terreno, como lo he creado y los materiales que utilizado. A continuación pasaré a hablar del tanque, las partes de las que se compone y su funcionamiento, después hablaré de los tanques enemigos y por último del UI.

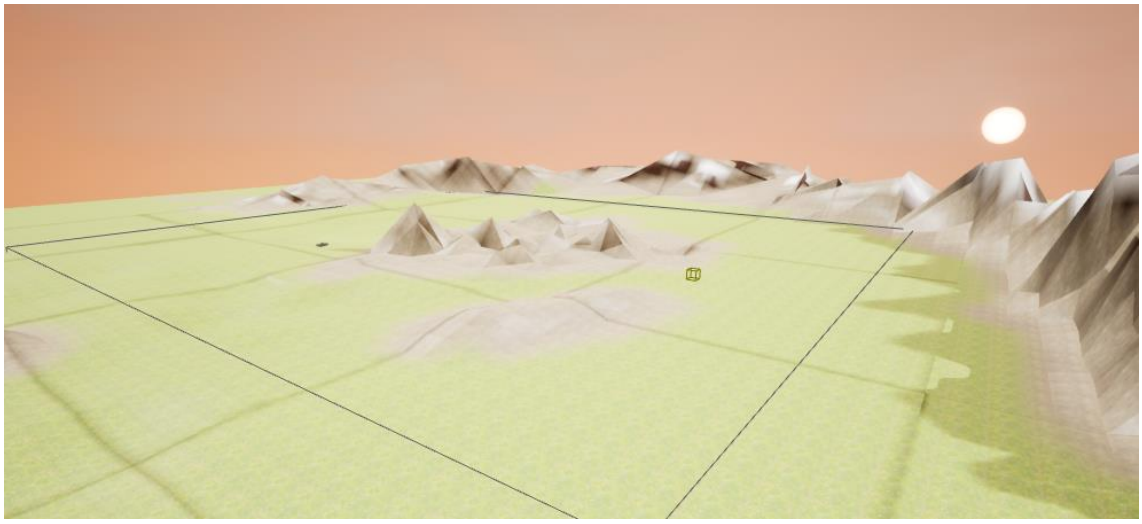
Terreno.

Para la creación del terreno hice uso de las herramientas que presenta Unreal para la creación de terrenos. Era la primera vez que los utilizaba por lo que fue un poco ver que hacía cada pincel y ver que tal iba quedando. Después para darle vida al terreno y que no fuera todo del mismo color implementé unos materiales para pintar directamente estos materiales sobre el terreno. Esto lo hice añadiéndolos como diferentes layers con las cuales iba pintado. Una fue con una textura de hierba, otra de tierra (para montañas) y la última de nieve.



Tras un rato creando el terreno, el resultado no era el esperado ya que se creaban demasiados polígonos y esto en un terreno pequeño no hay problema pero en uno mayor afecta bastante al rendimiento por lo que busqué como darle otro estilo. Después de investigar que otros estilos podían quedar bien, encontré un tutorial que convertía los materiales utilizados en estos mismo pero con un estilo low poly. Me gustó bastante el resultado por lo que me decanté por este estilo y aparte, fue bastante sencillo de utilizar. Esto lo hice mediante blueprint y este fue el resultado:



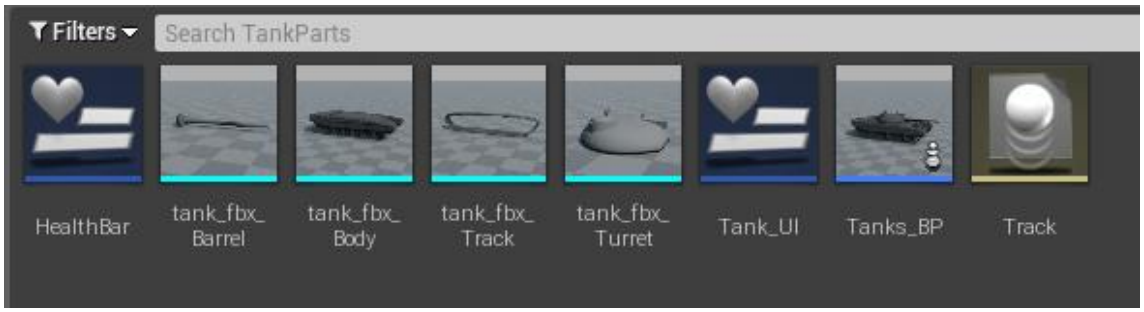


Con este estilo se redujo bastante el número de polígonos creados y al mismo tiempo el aspecto de las texturas mejoraba bastante.

Tanque.

Esta ha sido la pieza clave del proyecto, crear un tanque compuesto por diferentes partes que cada una de ellas tendrá una funcionalidad.

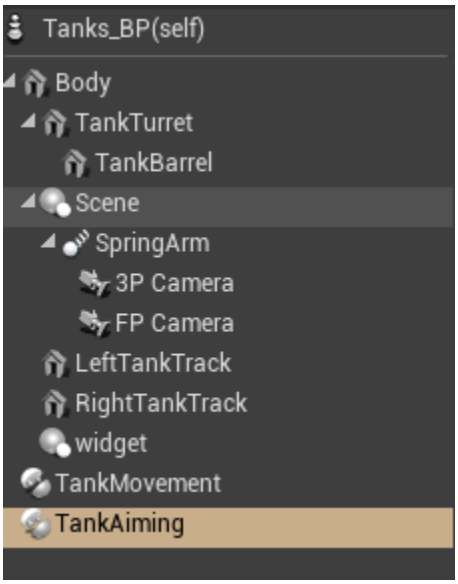
Partí de unos assets gratis de un tanque que encontré en internet, estos estaban compuestos por diferentes partes, el cuerpo del tanque, la torreta, el cañón y las correas que rodean las ruedas y que hace que se mueva el tanque.



Lo primero que hice fue montar el tanque atachando las diferentes partes del mismo por medio de los sockets para posicionarlos en su sitio correcto, era necesario que el tanque no fuera una sola pieza, ya que, si no, no permitiría tener ninguna funcionalidad.

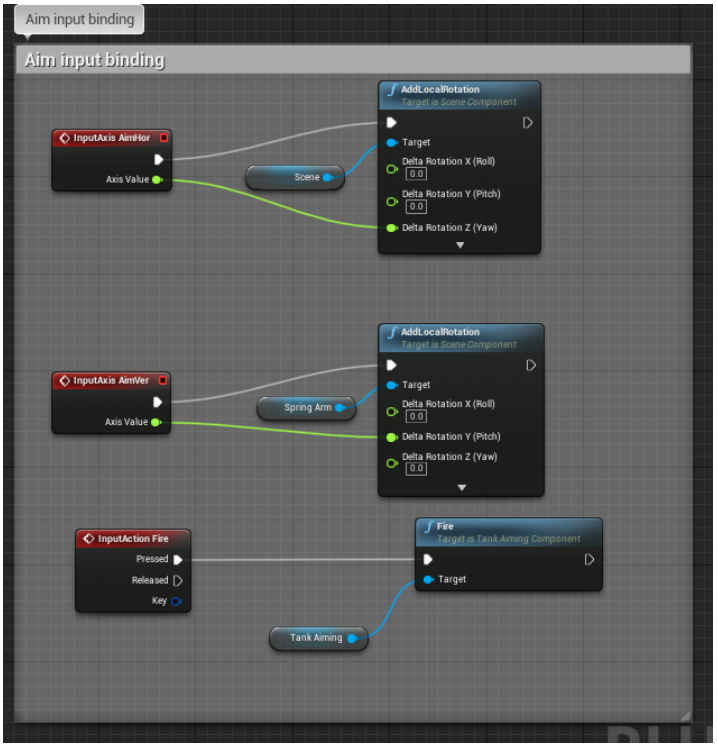
Una vez que tuve cada parte en su sitio pasé a implementar la cámara, está será en tercera persona y sería colocada detrás del tanque con un poco de ángulo.

Este sería el resultado de la jerarquía:

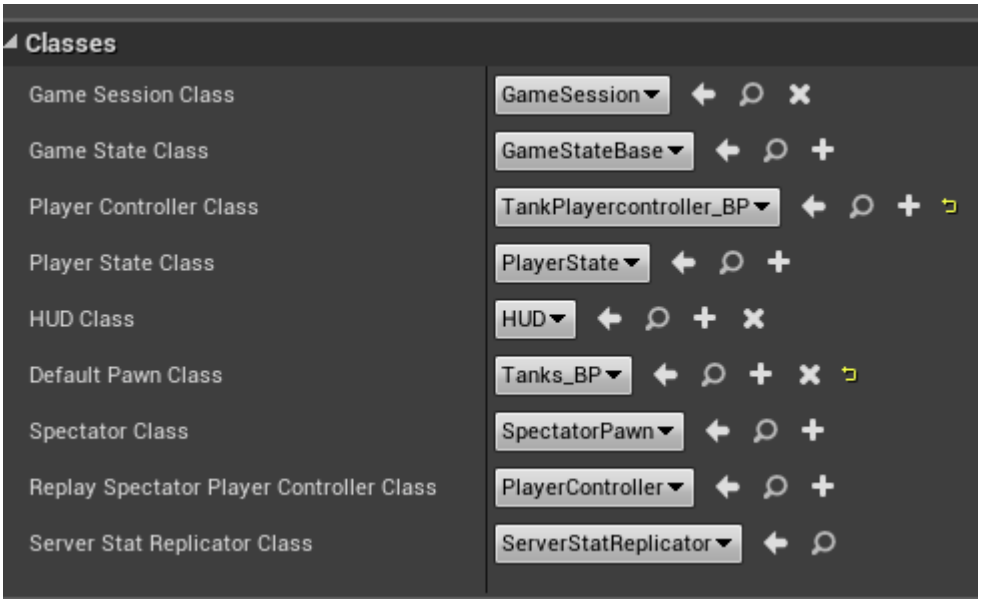


Y este el del tanque:

El siguiente paso fue dar rotación a la cámara cuando movemos el ratón en las diferentes direcciones. Esto lo hice añadiendo varios inputs y por medio de blueprint añadirles una rotación local. Para mejorar el manejo de la cámara Unreal tiene un elemento llamado “springArm” que te permite hacer hija de este a la cámara para evitar que esta rote en su eje Y. Sin esto la cámara a medida que te mueves rota tanto en el eje X como en Y por lo que hay que hacer uso de “springArm” añadiéndole a este la rotación en el eje Y y a la cámara en eje X. Los “springArms” tienen esta finalidad.



Para que el jugador no tenga ventajas con respecto a los enemigos, estos tendrán las mimas características que el player, es decir, que los enemigos serán tanques exactamente iguales que el que controla el player pero mediante el GameMode indicamos cual va a ser el que controle el jugador y cuáles serán los controlados por la IA.



A partir de aquí comenzó el proceso de implementación de las mecánicas que iba a tener el tanque. Para ello cree una clase c++ APawn. Para añadirla al blueprint creado se hace yendo dentro del blueprint a “Class Settings” y donde pone “class Options” seleccionamos la clase creada de c++.

Después de varias pruebas he descubierto que para la creación de un personaje o un tanque en este caso, sí que era mejor crear un blueprint primero y después realizar la funcionalidad del

mismo por medio de c++ pero para otros casos como componentes se puede realizar primero la clase en c++ y después añadir el blueprint a esa clase.

La primera mecánica que implementé fue la de apuntado. Para ello cree TankPlayerController_BP, este blueprint es de tipo PlayerController pero como en el tanque cree una clase en c++ de tipo playercontroller la cual va a ser padre de TankPlayerController_BP.

Esta clase en blueprint tan solo va a tener una referencia al "crosshair" creado mediante un widget UI y en la clase creada de c++ añadiremos la funcionalidad.

Una vez tengo esto lo primero que hice fue crear dos coordenadas con las cuales coincida el "crosshair" creado en el widget. A continuación cree las diferentes funciones que me permitirían apuntar en la dirección del "crosshair" con un rango de distancia editable tanto en código como en blueprint.

La finalidad de esta clase era comprobar si hay algo en la posición a la cual estamos apuntando en un rango determinado. Esto lo hice a través de FHitResult que sería el equivalente a Raycast en Unity.

```
bool ATank_PC::GetLookVectorHitLocation(FVector lookDirection, FVector& hitLocation) const
{
    FHitResult hit;

    auto StartLocation = PlayerCameraManager->GetCameraLocation();
    auto EndLocation = StartLocation + (lookDirection * lineTraceRange);

    if (GetWorld()->LineTraceSingleByChannel(
            hit,
            StartLocation,
            EndLocation,
            ECollisionChannel::ECC_Camera)
      )
    {
     hitLocation = hit.Location;
     return true;
    }
    hitLocation = FVector(0);
    return false; /// Line Trace didn't succeed
}
```

Y para acceder al vector que nos indica la posición en el mundo a la cual estamos apuntando lo hacemos mediante el método `DeprojectScreenPositionToWorld();` pasando como valores las coordenadas en la pantalla en X y en Y, un vector de la cámara y el vector director hacia el cual queremos proyectar.

```
bool ATank_PC::GetLookDirection(FVector2D screenLocation, FVector& LookDirection) const
{
    /// Screen position os crosshair to world direction
    FVector cameraWorldLocation;

    return DeprojectScreenPositionToWorld(screenLocation.X, screenLocation.Y, cameraWorldLocation, LookDirection);

}
```

La rotación del cañón la añadí más adelante en otra clase a la cual va a llamar tank llamada TankAimingComponent.

Una vez hecho esto pasé a crear TankAimingComponent, como el propio nombre indica esta clase será de tipo ActorComponent, la cual añadiremos más adelante al blueprint Tank_BP.

Desde TankAimingComponent vamos a controlar toda la mecánica del movimiento tanto del tanque del player como los de la IA, es decir tanto TankAIController como TankPlayerController van a llamar a la clase Tank que esta delegará a TankAimingComponent. Y después usaremos blueprint para la inicialización.

En TankAimingComponent vamos a tener varias funciones, destaco Initialise(), AimAt() y MoveBarrelTowards().

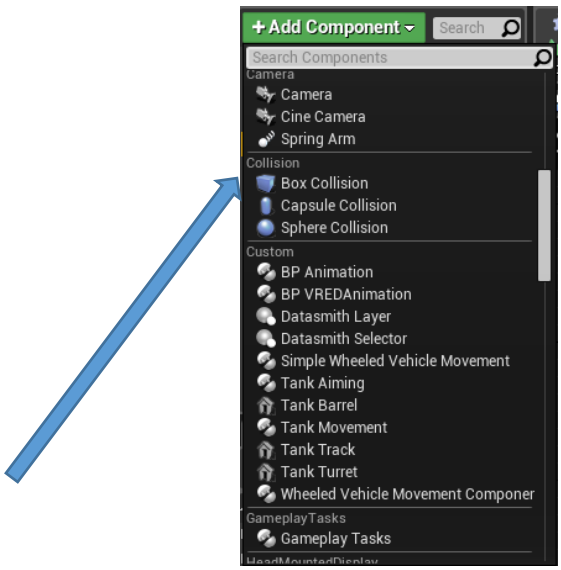
Desde la función Initialise() vamos a inicializar los elementos a los cuales les queremos dar la funcionalidad de apuntado. En este caso van a ser la torreta y el cañón. Para poder inicializarlos primero tuve que crear dos clases nuevas, en esta ocasión serán de tipo StaticMesh. Algo que he aprendido de Unreal es que puedes crear clases de todo tipo, al igual que creas elementos en blueprint o componentes para un blueprint, puedes crear clases como en este caso un StaticMesh y darle funcionalidad desde código. Una vez que tienes esto, puedes añadir este StaticMesh al blueprint como un componente más y mínimo va a tener las mismas funcionalidades que si lo hubieras añadido como componente directamente en el blueprint. Pero al añadirlo como una clase de c++ nos permite añadir variables, funcionalidades, etc a este StaticMesh.

Por tanto cree dos clases en c++ de tipo StaticMesh una para el cañón y otra para la torreta. En el cañón le añadí una función para que se pudiera elevar hasta cierto grado (Elevate()) y en la torreta le añadí una rotación en el eje horizontal de 360º (Rotate()). Al estar atachados por medio de los sockets era bastante lógico añadir estas rotaciones respectivamente.

Para poder añadir estos componentes a un blueprint hay que añadir esta línea de código a la clase en la cabecera:

```
UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class TANKPROJECT_API UTankAimingComponent : public UActorComponent
```

Aquí podemos especificar el ClassGroup, en este caso Custom, esto indica la categoría en la cual va a aparecer cuando busquemos el componente en el blueprint.



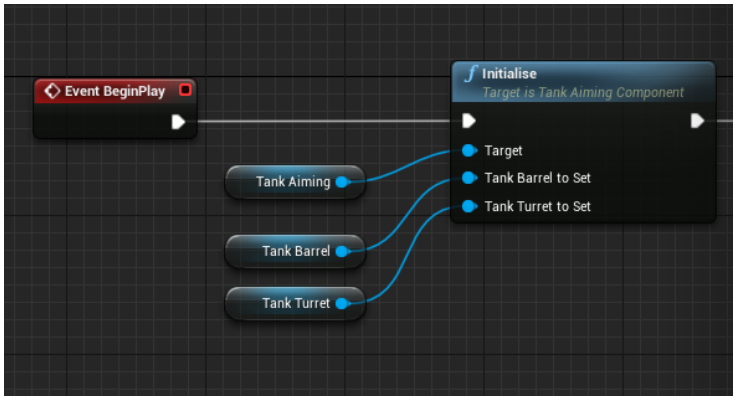
Una vez hecho esto cree la función Initialise() en TankAimingComponent, aquí creaba una referencia a los static mesh creados anteriormente y los inicializaba.

```
void UTankAimingComponent::Initialise(UTankBarrel * tankBarrelToSet, UTankTurret * tankTurretToSet)
{
    if (!tankBarrelToSet || !tankTurretToSet) { return; }

    barrel = tankBarrelToSet;
    turret = tankTurretToSet;
}
```

Una vez hecho esto al compilar podremos añadir TankAimingComponent al blueprint del tanque y podremos acceder a esta función en la cual inicializaremos los static mesh. Pero esto no funcionaría si no ponemos este tipo de propiedad a la función en la cabecera. Con la cual le indicamos que va a poder ser llamada desde blueprint.

```
UFUNCTION(BlueprintCallable, Category = Setup)
void Initialise(UTankBarrel* tankBarrelToSet, UTankTurret* tankTurretToSet);
```

Inicializamos en blueprint.



Para finalizar (por el momento) TankAimingComponent cree ambos métodos comentados anteriormente que iban a permitir mover el cañon en el eje “Pitch” y la torreta en el “Yaw”.

```
void UTankAimingComponent::MoveBarrelTowards(FVector aimDirection)
{
    /// Work-out difference between current barrel rotation and aimDirection

    if (!barrel || !turret) { return; }

    auto barrelRotation = barrel->GetForwardVector().Rotation();
    auto aimAsRotatot = aimDirection.Rotation();
    auto deltaRotator = aimAsRotatot - barrelRotation;

    /// Rotation of the barrel
    barrel->Elevate(deltaRotator.Pitch);
    if (FMath::Abs(deltaRotator.Yaw) < 180)
    {
        turret->Rotate(deltaRotator.Yaw);
    }
    else
        turret->Rotate(-deltaRotator.Yaw);

}
```

En la función AimAt() utilicé un método propio de Unreal que genera automáticamente una simulación de cómo va a ser un lanzamiento de un proyectil. Es decir crea la propia curvatura que va a tener el proyectil lanzado. `UGameplayStatics::SuggestProjectileVelocity();`

Este método devuelve true o false en caso de que en la posición a la cual estamos apuntando detecta que hay algo en la escena, en un rango determinado.

Así sería como quedó todo el código.

```
void UTankAimingComponent::AimAt(FVector hitLocation)
{
    ///auto ourTankName = GetOwner()->GetName();
    ///auto barrelLocation = barrel->GetComponentLocation().ToString();

    if (!barrel) { return; }

    FVector outLaunchVelocity;
    FVector startLocation = barrel->GetSocketLocation(FName("Projectile"));

    /// Calculate the outLaunchVelocity

    bool bHaveAimSolution = UGameplayStatics::SuggestProjectileVelocity
    (
        this,
        outLaunchVelocity,
        startLocation,
        hitLocation,
        launchSpeed,
        false,
        0,
        0,
        ESuggestProjVelocityTraceOption::DoNotTrace /// Allows to trace the projectile curve
    );

    if (bHaveAimSolution)
    {
        aimDir = outLaunchVelocity.GetSafeNormal();

        /// Move Barrel
        MoveBarrelTowards(aimDir);
    }
}
```

Rotaciones terminadas. Continué con la mecánica de disparo, en la cual, al pulsar el botón derecho el tanque lanzaría un proyectil. Esta función fue creada en TankAimingComponent.

Como con los anteriores componentes, cree una clase en c++ para el proyectil lanzado. En este caso no sería un staticMesh ya que no es lo que buscaba. Al ser un objeto que voy a querer spawnear cada vez que hago click, añadir una fuerza de lanzamiento, que detecte colisiones, añadir sistemas de partículas… Todo encajaba en que fuera una clase de tipo Actor. Como he hecho anteriormente cree un blueprint llamado Projectile_BP al cual he hecho hijo de la clase en c++. Dentro de c++ he creado las propiedades que va a tener, en este caso serán: StaticMesh, dos sistemas de partículas y un componente que genera una explosión en área. También creé una función que añadiría fuerza de lanzamiento al proyectil y otra función para detectar las colisiones en la cual desactivo y activo los sistemas de partículas, añado daño en área y destruyo el propio objeto. Para añadir el movimiento del proyectil cuando este es lanzado he utilizado una función de Unreal llamada `UProjectileMovementComponent*`.

La cual va a ser llamada en la función LaunchProjectile() que, a su vez, será llamada desde la función Fire() en TankAimingComponent.

```
UProjectileMovementComponent* projectileMovement = nullptr;

UPROPERTY(VisibleAnywhere, Category = "Components")
UStaticMeshComponent* collisionMesh = nullptr;

UPROPERTY(VisibleAnywhere, Category = "Components")
UParticleSystemComponent* launchBlast = nullptr;

UPROPERTY(VisibleAnywhere, Category = "Components")
UParticleSystemComponent* impactBlast = nullptr;

UPROPERTY(VisibleAnywhere, Category = "Components")
URadialForceComponent* explotionForce = nullptr;

UPROPERTY(EditDefaultsOnly, Category = "SetUp")
float destroyDelay = 5.f;

UPROPERTY(EditDefaultsOnly, Category = "SetUp")
float projectileDamage = 10.f;
```

Para añadir el daño en área por código se llama a la misma función que utilizaríamos en blueprint:

```
/// Aplicamos daño en área
UGameplayStatics::ApplyRadialDamage(
    this,
    projectileDamage,
    GetActorLocation(),
    explotionForce->Radius,
    UDamageType::StaticClass(),
    TArray<AActor*>() /// Damage all actors
);
```

Para la instanciación de elementos en escena por medio de un evento se hace a través del método SpawnActor<Atype>, que equivaldría al método instanciate en unity. Al tener ya creado la clase Proyectile y haberle añadido las funcionalidades pasamos a hacer el spawn de proyectiles cuando disparemos en TankAimingComponent.

Lo primero es crear un input binding, con el cual activaremos la función Fire() creada en c++. Una vez hecho esto dentro ya de la función Fire() spawneamos el proyectil y llamamos a la función LaunchProjectile() para darle el movimiento. Así quedaría le código:

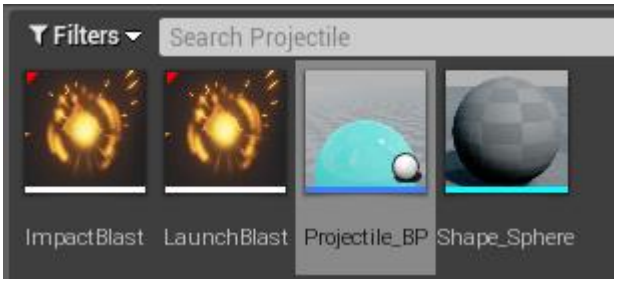
```cpp
void UTankAimingComponent::Fire()
{
    /// Returns a double

    if (state == EFiringState::Locked || state == EFiringState::Aiming)
    {
        //UE_LOG(LogTemp, Warning, TEXT("Fired projectile"));

        /// Spawn projectile
        if (!ensure(barrel)) { return; }
        if (!ensure(projectileBlueprint)) { return; }

        auto projectile = GetWorld()->SpawnActor<AProjectile>(
                projectileBlueprint,
                barrel->GetSocketLocation(FName("Projectile")),
                barrel->GetSocketRotation(FName("Projectile"))
            );

        projectile->LaunchProjectile(launchSpeed);


        lastFireTime = GetWorld()->GetRealTimeSeconds();
        roundsLeft--;
    }
}
```

Cuando spawneamos un actor en escena tenemos que especificar el tipo de actor que queremos. Para ello hay que pasar el blueprint del proyectil ya que es ahí donde tiene todas las funcionalidades. Por tanto para especificar que blueprint queremos que coja cuando vaya a spawnear un objeto, lo hacemos a través de un tipo de propiedad llamada DefaultSubObject y lo creamos de esta manera:

```cpp
UPROPERTY(EditDefaultsOnly, Category = "SetUp")
TSubclassOf<AProjectile> projectileBlueprint;
```

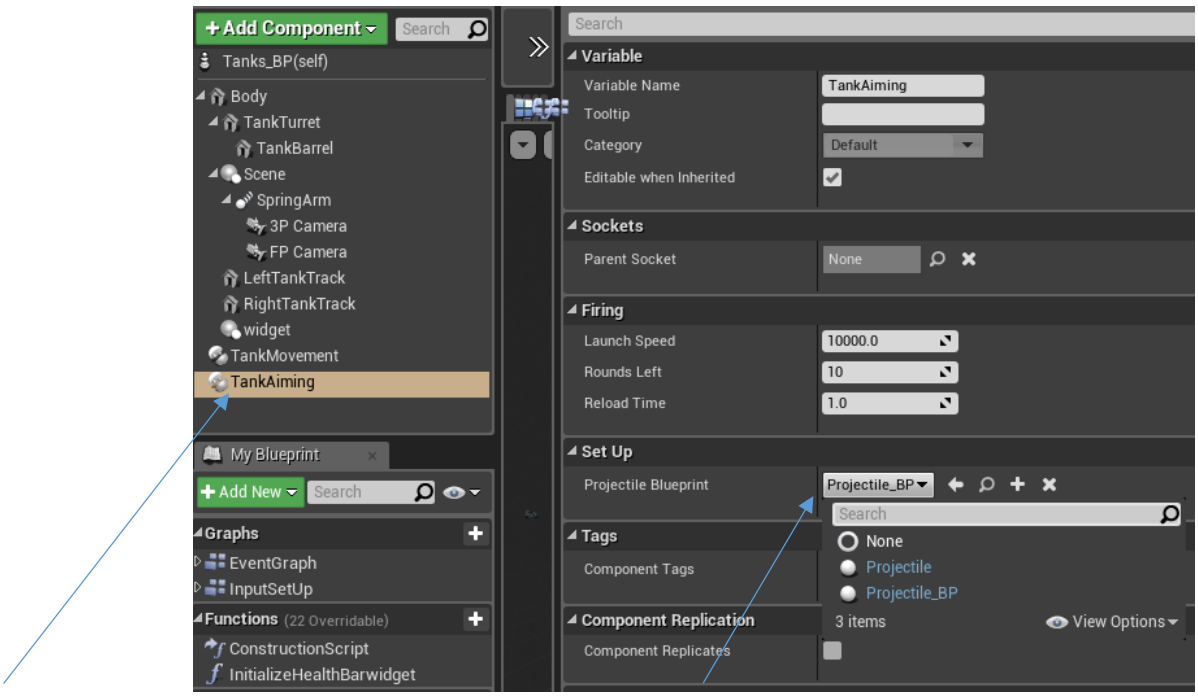
Al proyectil como he comentado añadí a su vez unos sistemas de partículas. Esto tiene una finalidad visual para hacer más épico el disparo y al mismo tiempo trabajar con sistemas de partículas con los cuales no había trabajado hasta el momento.

He añadido dos sistemas de partículas, uno que recreará la estela del disparo y otro recreará una explosión cuando colisiones con algún elemento de la escena.



Los parámetros principales que he retocado para recrear la estela fueron “Color Over Life” y “Size By Life” pero hay mil parámetros que puedes retocar y al mismo tiempo añadir nuevos etc…

Una vez que compilamos podemos seleccionar el tipo de blueprint al cual esto va a hacer referencia, al mismo tiempo podemos crear la categoría y darle el nombre que queramos:



De esta manera cuando accionamos el botón de disparar instanciaremos este tipo de proyectil. Hasta aquí la mecánica de disparo.

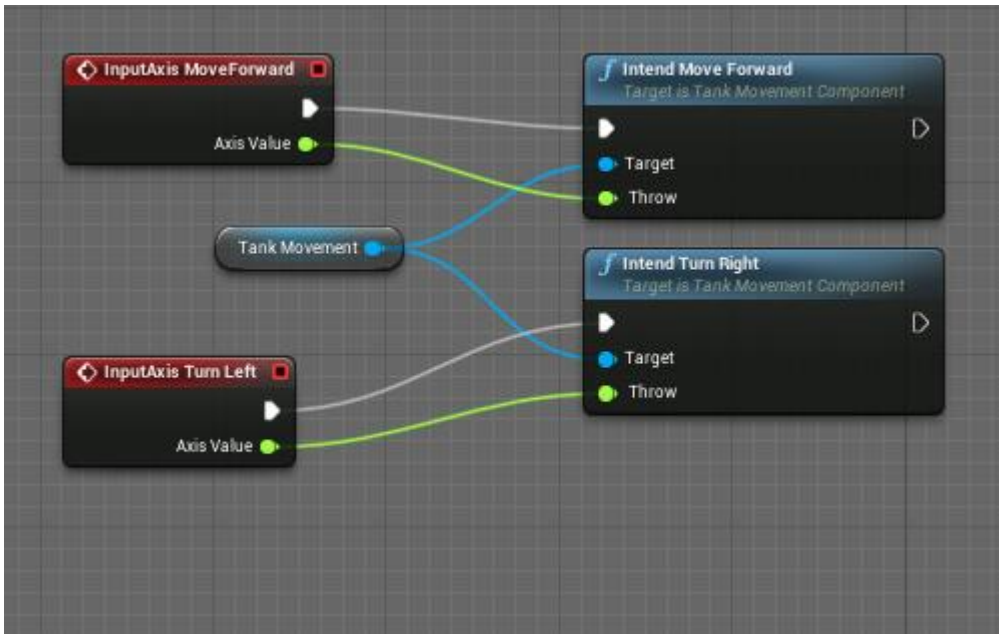
Teniendo una mecánica de disparo funcional, pasé a añadir movimiento al tanque. Como comenté al principio el tanque se compone de 4 piezas, cuerpo, torreta, cañón y 2 ruedas. Para el movimiento he hecho una cosa bastante parecida. He creado un ActorComponent de tipo clase c++ al cual añadiremos al tanque. Desde aquí igual que con el cañón y la torreta los inicializaremos y para ello hay que crear la clase StaticMesh en c++ para poder ser inicializada. Esto lo hice exactamente igual que con el cañón y la torreta.

Lo primero fue crear la función Initialise() con las referencias a las ruedas. El tanque tiene dos ruedas por lo que será el mismo asset duplicado. Ambas tendrán las mismas funcionalidades.

Lo primero fue añadir una fuerza a ambas ruedas para ello cree la función DriveTrack(), aquí calculo la fuerza en el vector forward para que vaya en la dirección deseada. Cojo una referencia de la posición actual del tanque y por último añado esta fuerza al propio tanque y no a las ruedas. Esto se debe a varios problemas encontrados si se lo añadía a las ruedas.

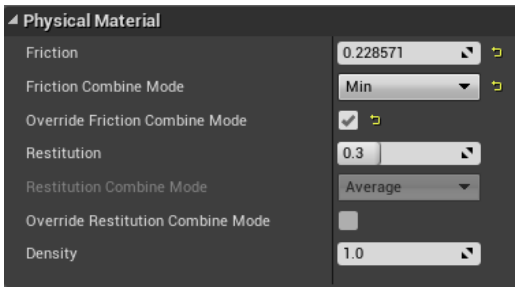
```
void UTankTrack::DriveTrack()
{
    /// Añadimos una fuerza para mover el tanque
    auto forceApplied = GetForwardVector() * currentThrottle * trackMaxDrivingforce;
    auto forceLocation = GetComponentLocation();
    auto tankRoot = Cast<UPrimitiveComponent>(GetOwner()->GetRootComponent());
    tankRoot->AddForceAtLocation(forceApplied, forceLocation);
}
```

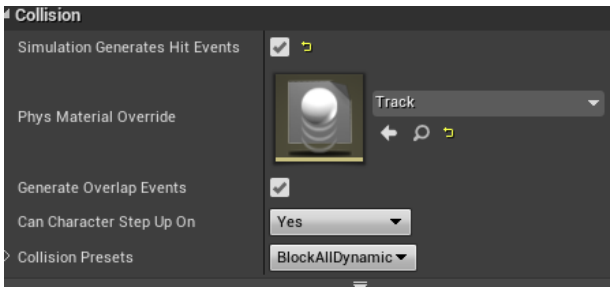
A continuación pasé a crear los métodos dentro de TankMovementComponent. Estos serán llamados en el propio blueprint del tanque en la sección de input bindings.



Las funciones son IntendMoveForward() e IntendMoveRight() desde las cuales añadimos el movimiento en las diferentes direcciones.

Con esto encontré varios problemas que explicaré en más detalle en la tercera sección de esta memoria pero uno que canta a la vista es que no compruebo cuando estoy colisionando con el suelo por lo que podía añadir fuerza aun estando en el aire. Esto fue arreglado con la función OnHit() que es exactamente igual que la creada en Projectile. Otro ajuste que hice fue crear un nuevo material de fricción en Unreal para que frenara cuando dejaba de añadir fuerza para moverse. Esto lo hice creando un “Physical Material” y añadiéndoselo a las Ruedas.

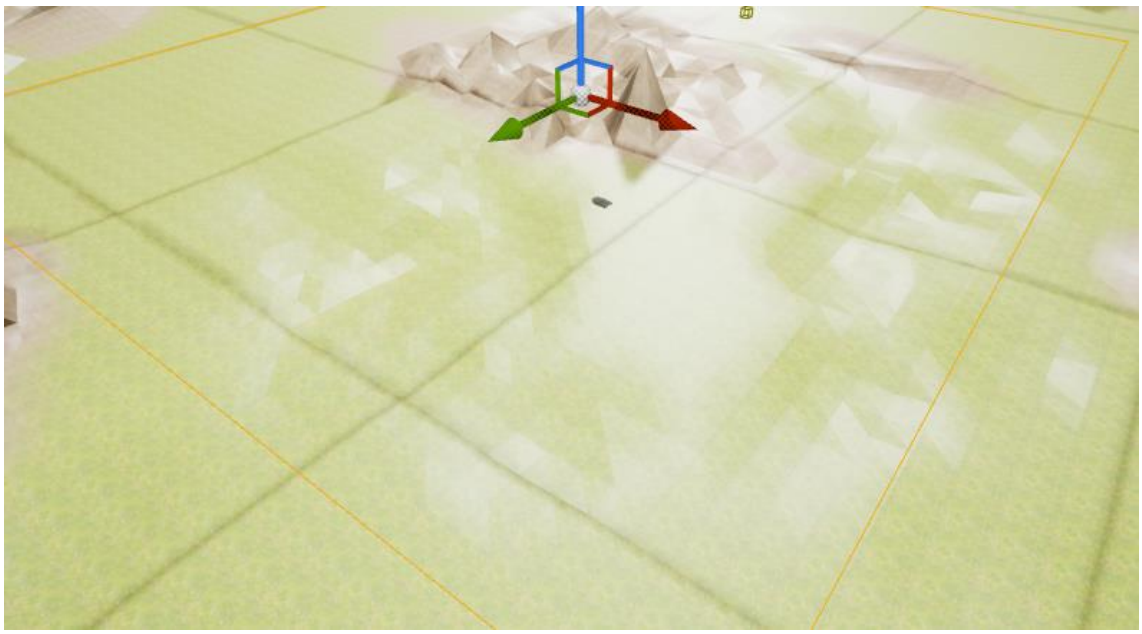




TankAIController.

Teniendo las mecánicas del tanque hechas pasé a la creación de los enemigos. Estos como he comentado anteriormente podrán realizar exactamente las mismas acciones que el player.

Lo primero que cree fue crear una “NavMeshBoundsVolume” para delimitar por donde estos se moverán.



Para hacer uso de esto nos vamos a la clase creada en c++ TankAIController. Desde aquí vamos a decir que esta clase controle a todos los tanques que haya en escena excepto el que controle el player. A continuación le damos movimientos llamando al método que nos proporciona Unreal MoToActor(), pasándole como parámetro el tanque controlado por el player y una distancia de parado, es decir, la distancia a la que queremos que se pare en vez de llegar hasta la misma posición que el player. De esta manera simulamos que nos persiguen los tanques.

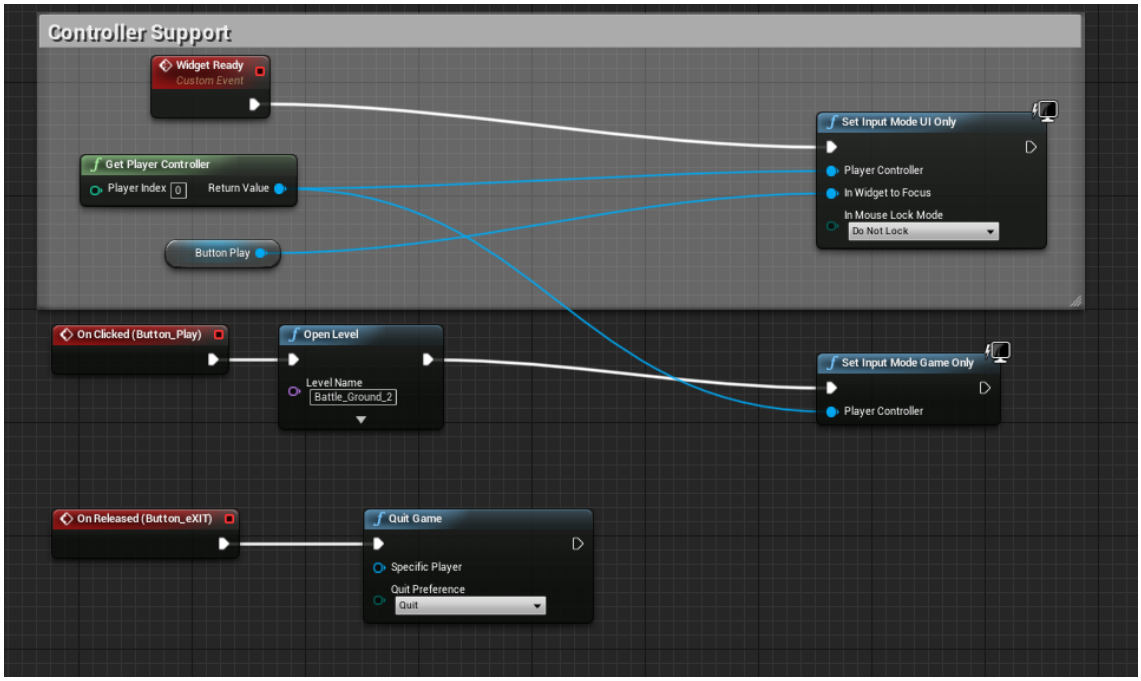
Por último igual que desde el tanque accedemos al método AimAt() con el cual apuntamos el cañon, en el ActorComponent TankAimingComponent, lo llamaremos de la misma manera en TankAIController pero en este caso le diremos que apunte al player y le dispare.

Esto se hará en el Tick de AtankAIController.

```cpp
void ATankAIController::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    auto playerTank = GetWorld()->GetFirstPlayerController()->GetPawn();

    auto controlledTank = GetPawn();

    if (!playerTank || !controlledTank) { return; }

    MoveToActor(playerTank, acceptenceRadius);

    /// Aim towards the player
    auto aimingComponent = controlledTank->FindComponentByClass<UTankAimingComponent>();
    aimingComponent->AimAt(playerTank->GetActorLocation());

    /// Fire if ready
    if (aimingComponent->GetFiringState() == EFiringState::Locked)
    {
        aimingComponent->Fire();
    }

}
```

UI.

Hay 4 tipos de UI, la primera será el menú principal desde el cual tendrá dos botones Jugar y Salir y será una escena aparte. La cual al pulsar jugar nos llevará a la escena de juego.

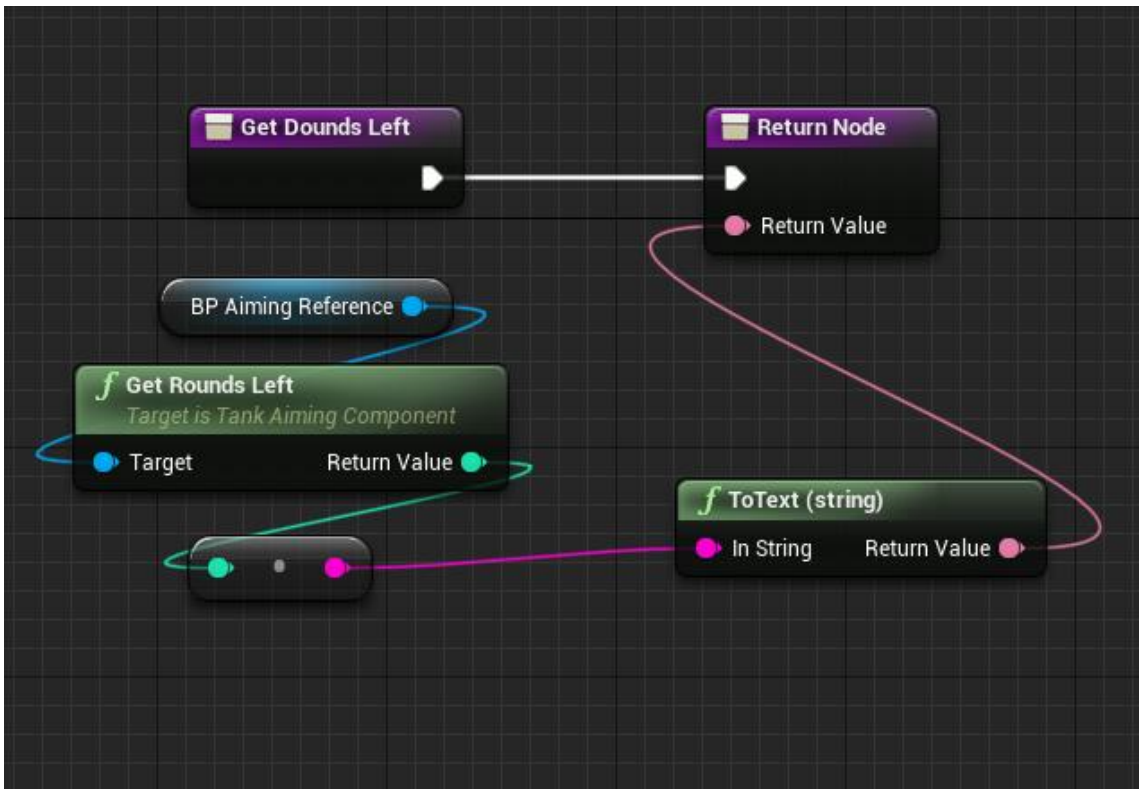


El tanque tendrá dos widgets, uno que contendrá el “crossHair” y la cantidad de munición que tenemos y otro que tendrá una barra de vida posicionado encima de los tanques. El crosshair está posicionado en la mitad de la pantalla y la munición en parte de arriba a la derecha de la pantalla.

Para acceder a la munición he creado en TankAimingComponent una variable munición a la cual le vamos restando 1 cada vez que disparamos. Para acceder a esta variable por medio de blueprint y añadirla a un widget, lo primero que hacemos es crear un texto y posicionarlo en en el panel “Designer” del UI. Una vez hecho esto creamos un binding para editar el texto. En el panel “graph” creamos una variable de tipo TankAimingComponent ya que es aquí donde hemos creado la función que devuelve en número de balas que nos quedan.

```
int32 UTankAimingComponent::GetRoundsLeft() const
{
    return roundsLeft;
}
```

Una vez tenemos esto accedemos por medio de la variable creada en el panel “graph” a esta función y la imprimimos por pantalla.



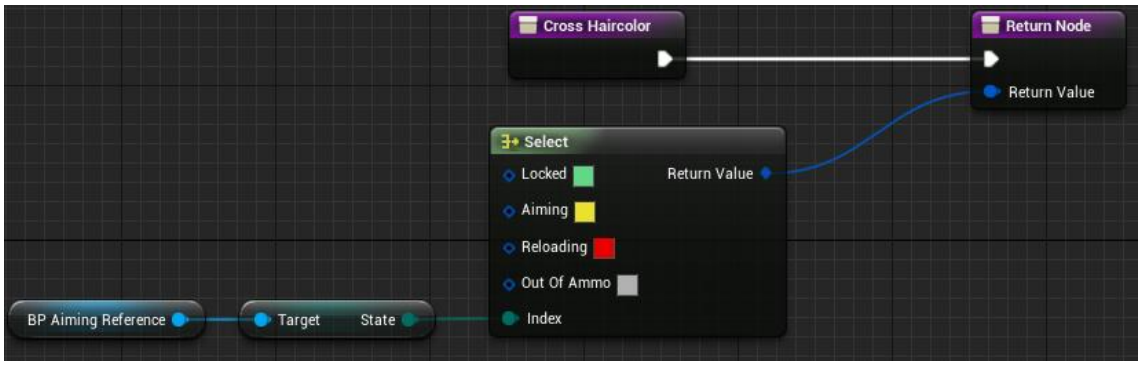
Para mejorar la funcionalidad del “crosshair”, añadí varios estados en los cuales puede estar el jugador. Esto lo hice para controlar mejor que el jugador no pueda disparar toda la munición en 1 segundo y al mismo tiempo que los tanques enemigos no disparen constantemente. A su vez al implementar esto hay que añadir una ayuda visual para que el jugador entienda que está pasando. Por tanto, cree varios estados por medio de un ENUM, estos serán:

- Locked (color verde), indica que podemos disparar.
- Aiming (color amarillo), indica que estamos apuntando.
- Reloading (color rojo), indica que estamos recargando por lo que no podemos disparar.
- OutofAmmo (color gris), sin munición.

Este ENUM fue creado en TankAimingComponent.

```
UENUM()
enum class EFiringState : uint8
{
    Locked,
    Aiming,
    Reloading,
    OutOfAmmo
};
```

Al mismo tiempo para acceder a esto por medio de blueprint y darle los diferentes colores cree una función que devuelve el estado actual. Los diferentes estados irán cambiando por medio de código.

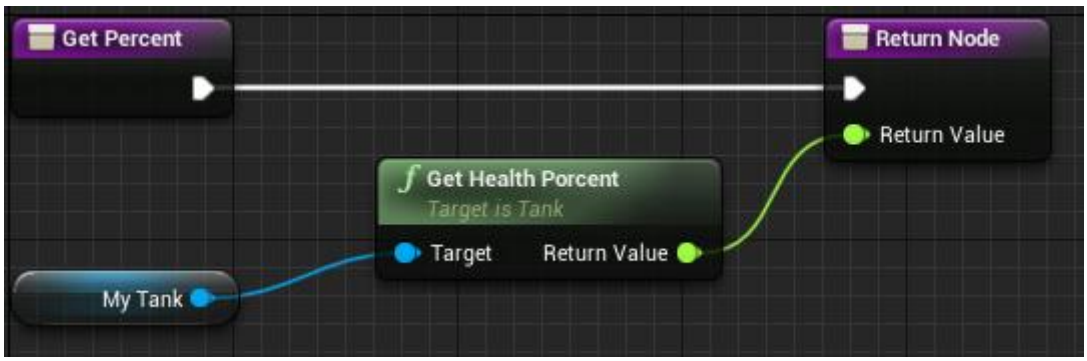


Como con la munición creamos un binding en el Widget.

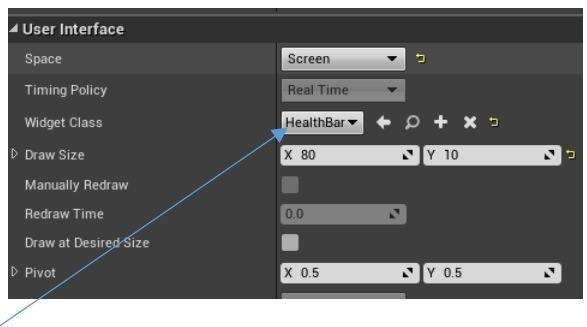
El último widget que implementé fue el de la barra de vida. Este fue un poco diferente. Lo primero que hice fue crear un nuevo widget y añadir en este una “ProgressBar”. Como con los anteriores cree un binding y cree un método que devolviera el porcentaje de vida actual de los tanques. Esto lo hice en código con esta función:

```
/// Obtenemos el porcentaje de vida
float ATank::GetHealthPorcent()
{
    return (float)currentHealth / (float)maxHealth;
}
```

Y se lo añadí a la “pregressBar” en panel “graph” del widget.



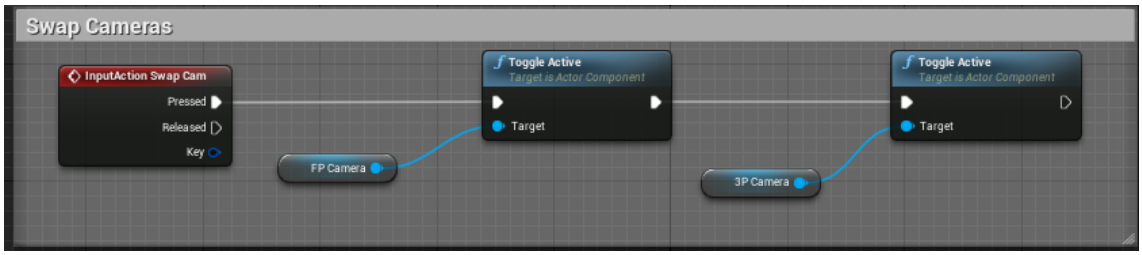
Por último añadí al blueprint del tanque un nuevo componente de tipo widget y le asigne el Widget HelthBar que acabo de crear.



De esta manera lo puedo posicionar donde quiera en el viewport del tanque.

Estos serían los widgets creados.

Por último añadí una segunda cámara al jugador debido a la sencillez de lo mismo. Tan solo creando un nuevo input con el cual cambiaremos de cámara. Después creamos una nueva cámara en el blueprint del tanque y la posicionamos donde queramos, en mi caso la he posicionado simulando primera persona y por último por medio de blueprint cambiamos de cámara cuando pulsamos el Input.



Los inputs que he utilizado para el control del tanque son los siguientes:

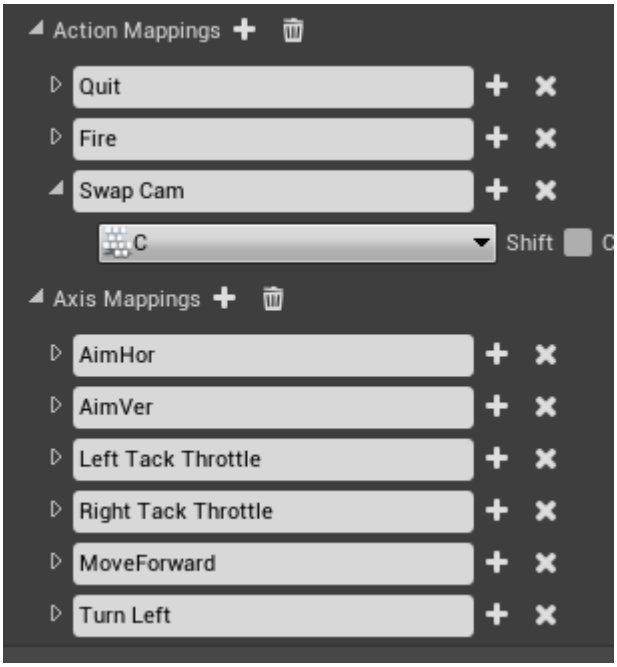
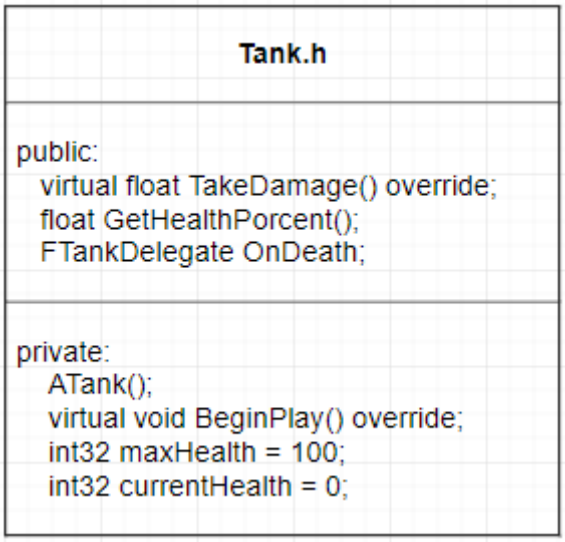
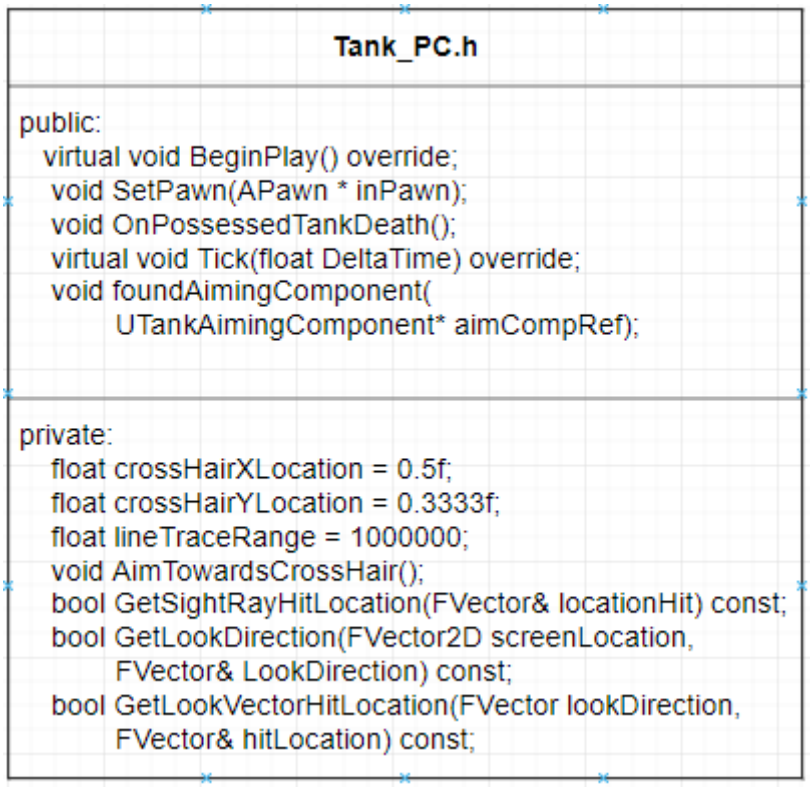
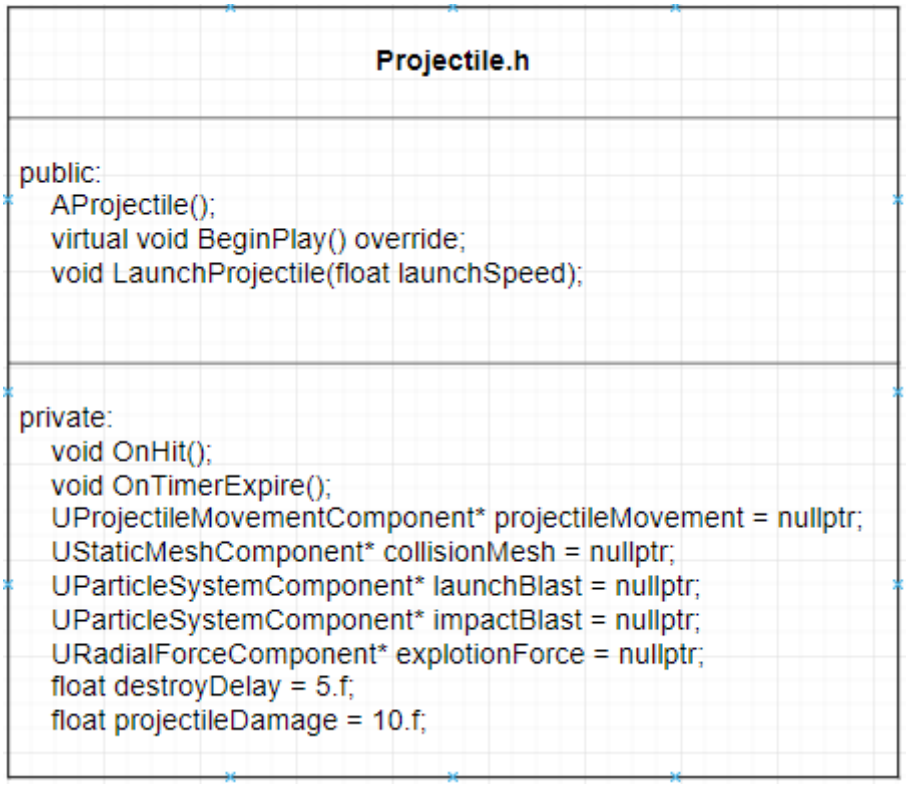
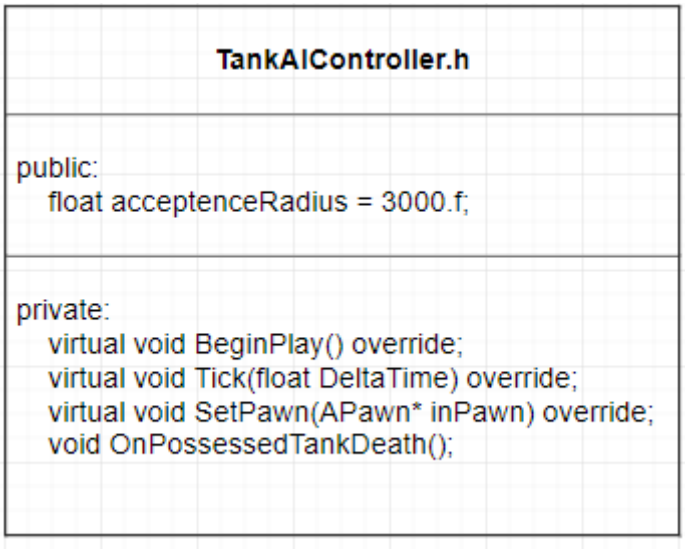


Diagrama de clases.






Projectile.h
public:
AProjectile();
virtual void BeginPlay() override;
void LaunchProjectile(float launchSpeed);
private:
void OnHit();
void OnTimerExpire();
UProjectileMovementComponent* projectileMovement = nullptr;
UStaticMeshComponent* collisionMesh = nullptr;
UParticleSystemComponent* launchBlast = nullptr;
UParticleSystemComponent* impactBlast = nullptr;
URadialForceComponent* explotionForce = nullptr;
float destroyDelay = 5.f;
float projectileDamage = 10.f;


TankAIController.h
public:
float acceptenceRadius = 3000.f;
private:
virtual void BeginPlay() override;
virtual void Tick(float DeltaTime) override;
virtual void SetPawn(APawn* inPawn) override;
void OnPossessedTankDeath();

TankAimingComponent.h

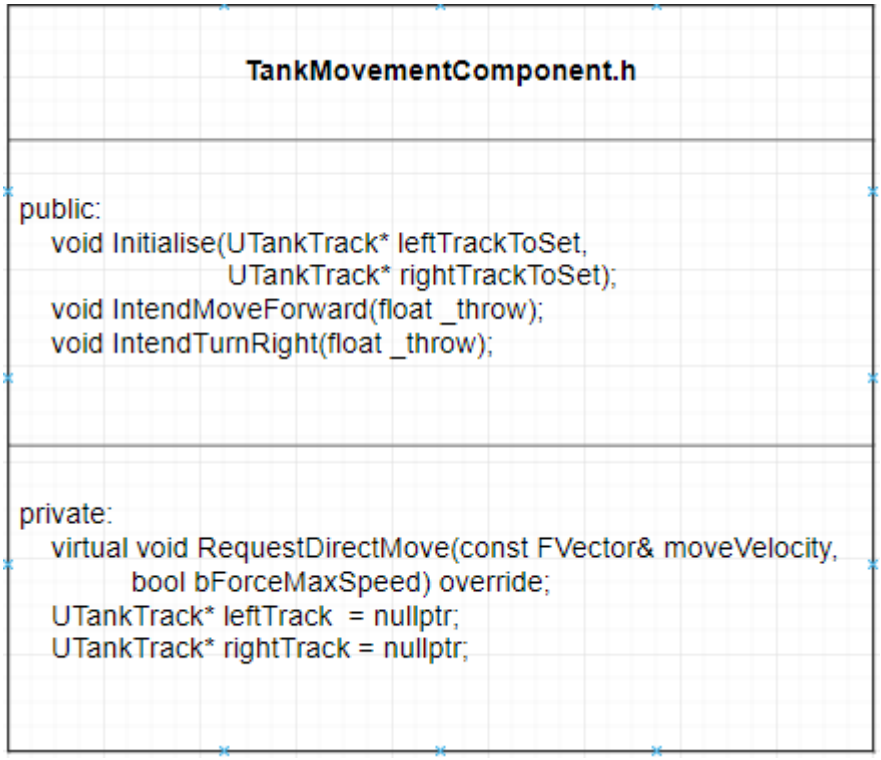
```
public:
    UTankAimingComponent();
    void MoveBarrelTowards(FVector aimDirection);
    EFiringState GetFiringState() const;
    void Initialise(UTankBarrel* tankBarrelToSet,
          UTankTurret* tankTurretToSet);
    void AimAt(FVector hitLocation);
    float launchSpeed = 10000.f;
    int32 roundsLeft = 10;
    void Fire();
    int32 GetRoundsLeft() const;
```

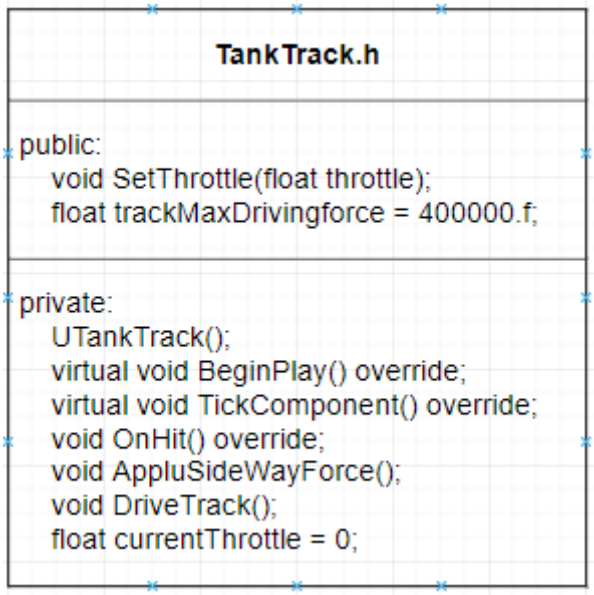
```
private:
    UTankBarrel* barrel = nullptr;
    UTankTurret* turret = nullptr;
    float reloadTime = 1.f;
    TSubclassOf<AProjectile> projectileBlueprint;
    float reloadTankInSeconds = 2.f;
    double lastFireTime = 0;
    FVector aimDir;
```

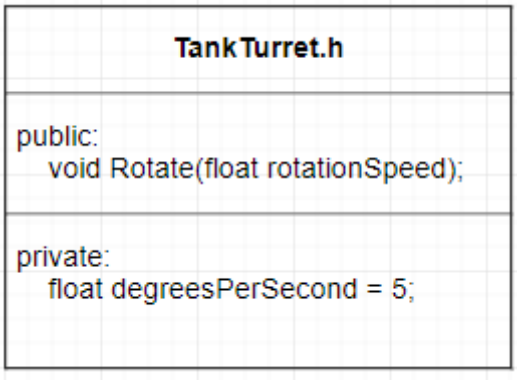
TankBarrel.h

```
public:
    void Elevate(float relativeSpeed);
```

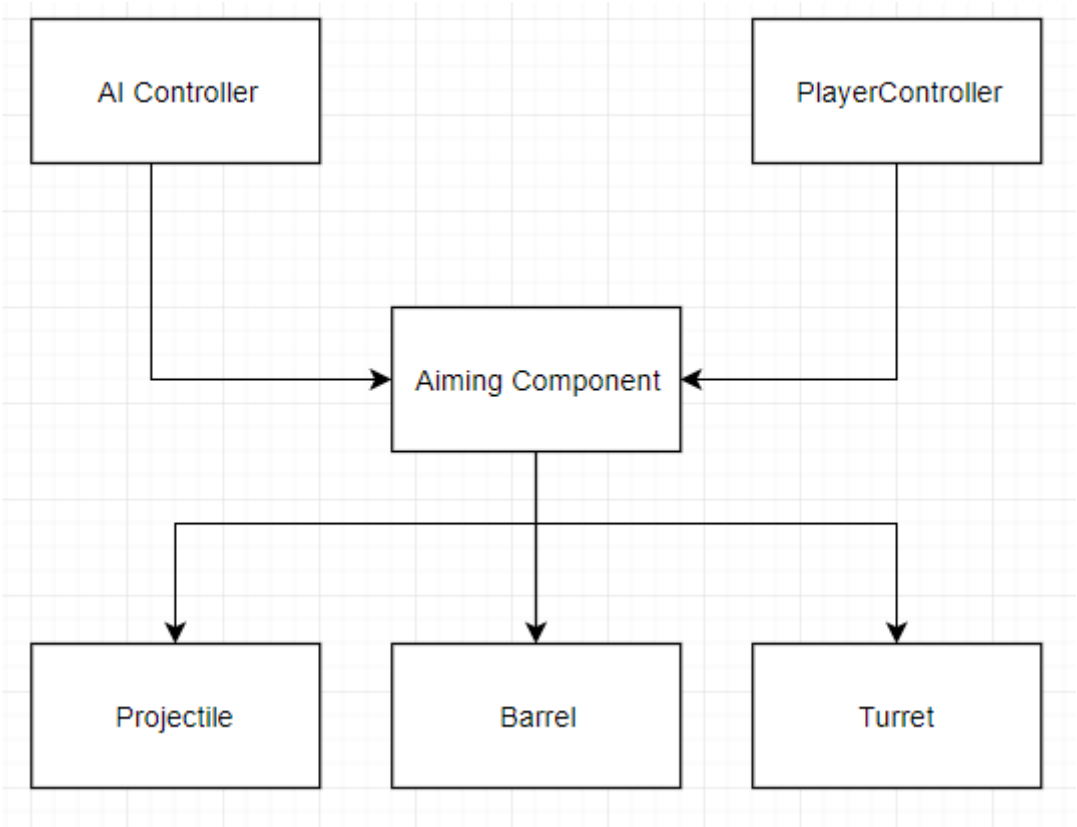
```
private:
    float MaxDregreesPerSecond = 5;
    float MaxDregreesElevation = 40;
    float MinDregreesElevation = 0;
```


TankMovementComponent.h
public:
void Initialise(UTankTrack* leftTrackToSet,
UTankTrack* rightTrackToSet);
void IntendMoveForward(float _throw);
void IntendTurnRight(float _throw);
private:
virtual void RequestDirectMove(const FVector& moveVelocity,
bool bForceMaxSpeed) override;
UTankTrack* leftTrack = nullptr;
UTankTrack* rightTrack = nullptr;


TankTrack.h
public:
void SetThrottle(float throttle);
float trackMaxDrivingforce = 400000.f;
private:
UTankTrack();
virtual void BeginPlay() override;
virtual void TickComponent() override;
void OnHit() override;
void AppluSideWayForce();
void DriveTrack();
float currentThrottle = 0;


TankTurret.h
public:
void Rotate(float rotationSpeed);
private:
float degreesPerSecond = 5;

Flujo del juego:



Diario de desarrollo y conclusiones: Obstáculos encontrados y enseñanzas aprendidas.

Quiero comenzar diciendo que ha sido un desarrollo bastante largo, con el cual he aprendido mucho sobre Unreal y es algo que me ha venido muy bien. Como he comentado en la introducción este proyecto ha estado enfocado en aprender todo lo posible de Unreal y sobre todo enfocado a aprender a programar con clases en vez de con blueprint. A su vez como el proyecto de final de carrera que estamos realizando va a ser en Unreal, era un gran aliciente para aprender lo máximo posible de Unreal y poder afrontar el TFG. Y esto me ha venido muy bien para aprender el funcionamiento de Unreal, la cantidad de herramientas que tiene y desconocía. Destaco sobre todo que, el motor está orientado a programar por blueprint, no hay duda en eso, pero sí que es cierto que baja el rendimiento del motor, al mismo tiempo, programar todo por código tampoco es recomendable debido que hay cosas muy sencillas que se pueden hacer en mucho menor tiempo en blueprint y te evitas los tiempo de compilación. Pero creo que la combinación de ambos saca el máximo partido al motor. Crear el funcionamiento en las clases en c++ y después hacer uso de las mismo por blueprint, es complicado al principio porque no es claro el funcionamiento, debido en gran parte, a que es una forma única de programar que tiene Unreal, pero después de un gran número de horas, te vas haciendo con ello y te das cuenta del gran potencial que tiene combinar ambas. Esto a su vez es de gran utilidad para los demás componentes dentro de un equipo de trabajo (diseño y arte), donde puedes hacer visible solo aquello que quieres que ellos toquen y ocultar por medio del código todo el funcionamiento y esto puede evitar una gran cantidad de posibles errores.

Por otro lado esto tiene desventajas, como he comentado el motor está enfocado a que programes por blueprint, con lo cual, en muchas ocasiones para hacer algo sencillo, vas a dedicar mucho más tiempo en buscar cómo se hace y hacer que funcione que lo que realmente te llevaría hacerlo. Esta es la principal desventaja que tiene el motor, en mi opinión.

Este ha sido el principal aliciente por el cual ha sido un proceso tan largo, el no saber el funcionamiento de muchas herramientas que proporciona Unreal y el no saber qué clase de métodos utilizar para realizar cada parte del proyecto. Por tanto ha sido un trabajo de investigación en su gran parte, debido a que en muchas ocasiones sí que sabía cómo afrontar un problema pero al mismo tiempo no sabía cómo sería la mejor manera de hacerlo en Unreal. Y mucho menos por medio de código.

Me planteé hacer este proyecto intentando utilizar el menor número de blueprints posible, ya que tras hacer la práctica de los semáforos vi el potencial que podía tener, aunque al mismo tiempo tampoco sabía qué iba a embarcarme.

El primer problema que encontré fue el que ya he comentado, muchas horas fuero invertidas en hacer funcionar scripts muy sencillos. El acostumbrarme a compilar y al estilo que utiliza Unreal. Por ejemplo, los tiempos de compilación de Unreal han mejorado bastante con respecto a hace unas versiones, esto se debía a que incluían todas las cabeceras de todos los métodos que contenía Unreal. Tras unas versiones para mejorar estos tiempos de compilación cambiaron este comportamiento y dependiendo de los métodos que quieras implementar tendrás que añadir unas cabeceras u otras. ¿Cuál es el problema? Pues que a veces incluyes un método y visual studio no lo subraya con rojo para indicarte que no lo puedes utilizar porque no has incluido la cabecera pero compilas y da error. En el error que aparece te indica que te falta un

punto y coma o algún otro error que no es lo que realmente ocurre. Por lo que te puedes tirar horas viendo el código y viendo que has escrito todo bien pero no va a compilar.

Otro problema con las cabeceras es que en los include que añadimos al principio siempre tiene que estar en “header file” el último el de la propia clase y en el archivo cpp el primero. Si no es así no va a compilar el código.

Muchas horas malgastadas en esto. Aun que una vez que te haces a esto y tienes cuidado de ponerlo todo en su sitio no hay problema.

Muchos problemas que tuve fueron con los a métodos utilizar o qué tipo de clase utilizar. Estos pude solventarlos con la gran suerte de que hay algunos tutoriales enfocados a programar en c++ en Unreal. Por lo que me salvaron mucho la vida.

Otro errores que encontré fueron bugs del propio unreal, cuando atachas una clase creada en c++ a un blueprint hay que tener mucho cuidado porque no guarda los cambios Unreal. En mi caso me ha pasado al incluir las ruedas del tanque, se quitaban solas cuando compilaba y tenía que atacharlas cada vez que compilaba, no entiendo por qué. Después de un tiempo esto se arregló pero otro, por ejemplo, en el componente que añado al tanque TankAimingComponent donde le digo que acceda al proyectil cuando accionamos la mecánica de disparo, siempre se quita, cada vez que compilo se desatacha y no sé por qué puede ser…

Por esto mismo hay que tener mucho cuidado con la programación ya que siempre estamos accediendo por medio de punteros a otras clases y estas suelen ser inicializadas en blueprint o llamadas en blueprint. Por lo que en caso de que se desatachen y no protejamos esos punteros harán que crashee Unreal. Por eso te vas a encontrar a lo largo de todo el código la misma línea:

If(!variable) {return;}

Esto me ha salvado de mil crushes y, al mismo tiempo, me ha servido para acostumbrarme a utilizar esto y hacerlo como algo normal ya que creo que es una muy buena práctica.

Un bug que tengo en el juego es que a veces cuando navegamos en el terreno, de repente el tanque colisiona con el suelo y salta por los aires. Creo que es porque el staticmesh asociado a las ruedas a veces se mete dentro del terreno y provoca este comportamiento en algunas zonas, un poco raro la verdad.

Otro problema que quiero destacar fue un fallo en la estructura del código. A mitad del proyecto me di cuenta de que estaba mezclando mucho el flujo y hacía inestable el proyecto porque llamaba a diferentes clases desde muchos sitios a la vez, luego otras clases se llamaban mutuamente. Me di cuenta de esto revisando las cabeceras en los include, prácticamente todas las clases se llamaban entre ellas de una manera u otra. Esto me hizo dejar el proyecto a un lado y trabajar en la estructura.

Este es un gran fallo que tengo, en el cual no he trabajado nunca y siempre me pasa factura. Siempre me pongo a programar y sobre la marcha voy viendo cómo tiene que ir cada cosa sin pensar previamente la estructura que quiero que tenga. Después de tener el mismo fallo en este proyecto y ver los problemas que me ha ocasionado, me ha hecho replantearme la estructura, limpiar el proyecto y rehacerlo todo prácticamente.

Este problema creo que ha sido el que mejor me ha venido para mejorar como programador, antes de ponerme a hacer nada, coger papel y boli y dedicar un tiempo a cómo voy a plantearlo enfocándome en la estructura y el flujo que quiero que tenga.

Como conclusión he de decir que he aprendido mucho con esta asignatura y que me veo capacitado para afrontar un proyecto como el que se me presenta al año que viene. He mejorado mucho este año y he aprendido muchísimo de Unreal. A la vez creo que el motor de Unreal presenta una infinidad de componentes que lo hacen que sea muy potente y por ello mismo me he dado cuenta de la infinidad de cosas que se pueden hacer en él. Yo solo he rozado la punta del iceberg.

He estado llevando un control de versiones del proyecto a través del GitHub, donde se puede observar que he ido implementando y el avance del proyecto, he dejado el link en la portada de esta práctica, lo vuelvo a dejar por aquí.

https://github.com/VictorGar96/Tank-Project

Con este Proyecto también me planteé realizar un control del avance del proyecto por medio de GitHub ya que es algo a lo cual no he dedicado el tiempo que debería, ya que no entendía del todo cómo funcionaba con Unreal pero resultó ser más sencillo de lo esperado.

Muchas gracias por todo Luis, como he dicho, he aprendido mucho este año, al principio no me gustaba nada Unreal pero tras la realización de todas las prácticas destacando esta última he ido familiarizándome con el motor y ver la cantidad de cosas que se pueden hacer con él y me ha hecho que poco a poco me guste cada vez más.