实现对象池模式

创建两个C++类

Actor：PooledActor

ActorComponent：StaticActorPool

PooledActor应当有两个变量：Bool的InUse和Float的LifeTime

其中的LifeTime是可编辑实例生成时公开。

在PooledActor中创建函数void SetInUse（bool InUse）。

函数整体逻辑

SetInUse和核心是，当Actor没有使用时，设置碰撞和打开隐藏Actor，并且关闭Tick

当我们SetInUse为True时，我们会创建一个定时器，在LifeTime时间过后，重新调用SetInUse(false)

这里就要求我们在C++中实现一个Timer逻辑

// 定时器逻辑

private:

    FTimerHandle TimerHandle;

    void OnTimerElapsed();// 定时器回调函数

public:

    // Timer完成委托（蓝图可绑定）

    UPROPERTY(BlueprintAssignable,Category="Timer")

    FOnTimerFinished OnTimerFinished;

    // 启动Timer

    UFUNCTION(BlueprintCallable,Category="Timer")

    void StartTimer(float Delay);

    // 停止Timer

    UFUNCTION(BlueprintCallable,Category="Timer")

    void StopTimer();

    // 检查Timer是否激活

    UFUNCTION(BlueprintPure,Category="Timer")

    bool IsTimerActive() const;

然后实现函数

首先是几个前置函数

void APooledActor::StopTimer()

{

    UWorld\* World = GetWorld();

    if (!World) return;

    World->GetTimerManager().ClearTimer(TimerHandle);

}

bool APooledActor::IsTimerActive() const

{

    UWorld\* World = GetWorld();

    if (!World) return false;

    return World->GetTimerManager().IsTimerActive(TimerHandle);

}

void APooledActor::OnTimerElapsed()

{

SetInUse(false);

    OnTimerFinished.Broadcast();

}

然后是StartTimer函数

void APooledActor::StartTimer(float Delay)

{

    UWorld\* World = GetWorld();

    if (!World) return;

    StopTimer();// 清除现有定时器

    // 设置新定时器

    World->GetTimerManager().SetTimer(

        TimerHandle,    // 定时器句柄

        this,   // 回调对象

        &APooledActor::OnTimerElapsed,  // 回调函数

        Delay,  // Timer时间

        false   // 是否循环

    );

}

接下来实现SetInUse函数

void APooledActor::SetInUse(bool InUse)

{

    bInUse = InUse;

    SetActorEnableCollision(InUse);

    SetActorHiddenInGame(!InUse);

    SetActorTickEnabled(InUse);

    if (InUse)

    {

        StartTimer(LifeTime);

    }

}

测试，完成

最后再给PooledAmmo加一个SphereCollision就结束了，这一部分的碰撞逻辑不在C++里实现，在蓝图子类中实现

然后实现ActorPool

创建ActorComponent类，命名为StaticActorPool

这个对象池的关键就是控制好PooledActor的数量和SetInUse

下面创建子弹类PooledActor，命名为PooledAmmo

子弹类的关键是加一个发射物移动组件和静态网格体组件和NiagaraParticle组件

PooledAmmo的静态网格体组件的碰撞预设应当是Overlap而不是Block

添加Niagara需要在Build.cs里面加一个“Niagara”

UPROPERTY(EditAnywhere,BlueprintReadWrite,Category="InitValues")

float Speed;

UPROPERTY(EditAnywhere,Category="InitValues")

UStaticMeshComponent\* StaticMesh;

UPROPERTY(EditAnywhere,Category="InitValues")

UNiagaraComponent\* NiagaraParticle;

UPROPERTY(VisibleAnywhere,BlueprintReadWrite,Category="InitValues")

UProjectileMovementComponent\* ProjectileMovement;

在构造函数中

APooledAmmo::APooledAmmo()

{

    StaticMesh = CreateDefaultSubobject<UStaticMeshComponent>("AmmoMesh");

    RootComponent=StaticMesh;

StaticMesh->SetMobility(EComponentMobility::Movable);

StaticMesh->SetCollisionResponseToAllChannels(ECR\_Overlap);// 碰撞预设

    NiagaraParticle = CreateDefaultSubobject<UNiagaraComponent>("AmmoParticle");

    NiagaraParticle->SetupAttachment(RootComponent);

NiagaraParticle->SetMobility(EComponentMobility::Movable);

    ProjectileMovement = CreateDefaultSubobject<UProjectileMovementComponent>("ProjectileMovement");

    ProjectileMovement->UpdatedComponent = StaticMesh;

    ProjectileMovement->InitialSpeed = Speed;

    ProjectileMovement->MaxSpeed = Speed;

    ProjectileMovement->bRotationFollowsVelocity = true;

    ProjectileMovement->ProjectileGravityScale = 0;

    ProjectileMovement->Velocity = GetActorForwardVector() \* Speed;

}

其中

ProjectileMovement->Velocity = GetActorForwardVector() \* Speed;

这一句有问题

因为我们的使用方法是：创建PooledAmmo蓝图子类，然后在根组件的细节面板中给InitValues的Speed属性手动赋值，然后在场景中应用

但是，当我们在C++的构造函数中进行Velocity的赋值时，Speed并不能从蓝图细节面板中传递到C++中，所以在C++的构造函数发生的时候，Speed值还是0

解决方法就是把这一句放到BeginPlay中

同理，所有在蓝图中给值的，都不能在C++的构造函数中作为右值赋值。如果可以的话，在BeginPlay中给值。

当我们给一个Actor加上StaticActorPool，并将PooledActor设置为子弹类时，我们可以发现：发射的子弹的运动方向是锁定的，都朝着一个方向，而不是朝着角色的方向。

解决方法就是：重载SetInUse函数，在父类的基础上加上SetVelocity功能，而不是让这些在BeginPlay中实现

需要修改PooledActor的UFUNCTION声明为BlueprintNativeEvent和Callable

然后在子类中声明该函数

// UFUNCTION(BlueprintNativeEvent,BlueprintCallable)

// 不需要声明UFUNCTION

virtual void SetInUse\_Implementation(bool InUse) override;

实现该函数

void APooledAmmo::SetInUse\_Implementation(bool InUse)

{

    Super::SetInUse\_Implementation(InUse);

    if (InUse)

    {

        ProjectileMovement->Velocity = GetActorForwardVector()\*Speed;

        NiagaraParticle->Activate();

    }

    else

    {

        ProjectileMovement->Velocity =FVector(0,0,0);

        NiagaraParticle->Deactivate();

    }

}

这里我们特别的需要对Particle进行Deactivate和Activate的调用。因为父类的SetInUse的相关设置不能作用到Particle