《Exploring in UE4》物理模块浅析[原理分析]

Jerish

一．Mesh组件与物理

关于UE物理的基本使用，官方文档以及我之前CSDN博客里已经做了较为详细的介绍。

这里主要是从代码方面，简单分析一下UE4里面的物理是如何使用与生效的，StaticMesh以及SkeletalMesh对应的物理都是如何产生与作用的。第四部分会涉及到一些PhyX引擎的内容，简单谈谈UE与PhysX间的交互。

这篇文章只讨论刚体物理。

首先，在游戏中常见的带有物理的物体一般有5种，虽然这5种类型本质上产生物理的规则都大同小异，但为了方便我们只针对StaticMesh与SkeletalMesh来总结。

1. 胶囊体一类（USphereComponent，UBoxComponent，UCapsuleComponent）
2. 静态网格物体StaticMesh
3. 骨骼网格物体SkeletalMesh
4. Landscape地形
5. PhysicsVolume（BrushComponent）

于带动画表现的玩家模型，是通过3D建模软件导入的带骨骼信息的资产。导入到引擎资源文件夹后就变成了USkeletalMesh，从资源文件夹拖到场景中后就变成了SkeletalMeshActor。当然，单独把SkeletalMeshActor放在场景中就如一个StaticMeshActor一样，没有任何动画，也可能没有任何物理（如果没有特殊处理的话）。

由于UE4提倡组件式的开发，Actor身上的很多特性都是通过组件提供的，所以物理数据都是挂在组件上的而不是Actor上。任何Actor上面都可以挂上N多个组件，因此一个玩家身上就可以有多个UStaticMeshCompnent与USkeletalMeshCompnent（一般还有一个胶囊体作为根组件）。举个简单的例子，玩家自身的模型是一个USkeletalMeshCompnent，然后身上的衣服装备就可以用一个UStaticMeshCompnent来表示。二者最大的差别就是SkeletalMesh可以产生动画，因此他自身的每个骨骼物理也就需要跟随动画而改变，所以相比StaticMesh要复杂不少。

对于一个静态网格物体StaticMesh，他的物理一般在建模软件里面就应该创建好，导入到编辑器时UE就会根据导入的数据创建物理信息，当然UE4本身也提供了物理碰撞的创建，如图1-1所示。不过无论哪种做法，本质上都是在编辑器里给UStaticMeshComponent构建一个UBodySetup，在开始游戏的时候在创建运行时的基本物理数据UBodyInstance。

图1-1 UE编辑器添加碰撞

> UBodySetup与UBodyInstance：我个人理解UBodySetup就是一个静态的物理数据，一般在在游戏运行前就已经构建好了[当然，你在游戏运行时创建也没什么问题]。你可以理解为一个类，编译以后就存在了。而UBodyInstance是一个在游戏时真正起作用的物理数据，可以理解为通过这个类创建的对象，运行时才真正出现。通过一个UBodySetup是可以创建出多个UBodyInstance的

而对于骨骼网格物体SkeletalMesh，由于数据比较多，他的物理数据存储在PhysicsAsset里面。在游戏运行的时候，SkeletalMeshComponent会读取物理资产里面的数据UBodySetup随后再通过UBodySetup给角色创建对应的基本物理数据UBodyInstance。再进一步深入就是NVIDA的PhysX物理引擎了（当然你也可以采用BOX2D物理引擎），这篇文章后面会有简单的讲解。

> UE4里面除了SkeletalMeshComponent.cpp以外还有SkeletalMeshComponentPhysics.cpp，PhysAnim.cpp用来专门处理SkeletalMeshComponent物理相关的逻辑

图1-2 物理资产

下面的图片描述了Mesh、 Component与物理基本类的基本关系

图1-3 物理相关类图

如果对BodyInstance还是觉得比较陌生的话，不妨结合下面我们熟悉的图来理解。我们知道在给Mesh设置物理的时候需要设置准确的碰撞通道，才能让不同的物理之间有碰撞效果。仔细看一下，CollisionResponses，ObejctType这些其实都是FBodyInstance里面的成员，我们在编辑器里面设置的这些属性其实就是在给BodyInstance设置（进一步还会去给到PhysX里面的PxRigidActor，后面讲）。

图1-4

如果我们想在编辑器里直观的看到是否创建了物理就调用控制台命令Show Collision既可。下图就显示了角色的胶囊体碰撞以及对应的骨骼物理碰撞（多个胶囊体组合）。

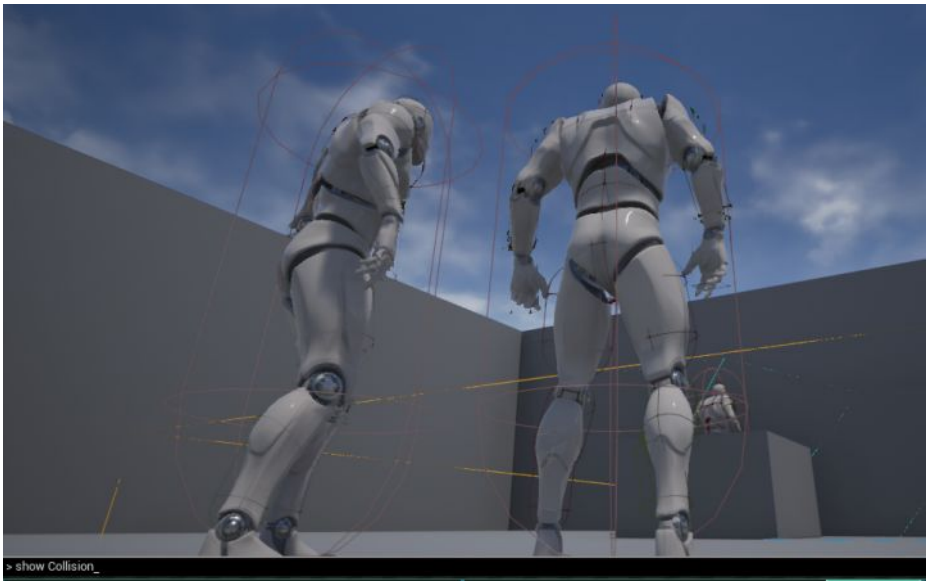


图1-5

二．物理的创建时机

生真正的物理。下面我们从游戏内具体的物理初始化流程分析一下。

2.1 UStaticMeshComponent的物理创建

首先是UStaticMeshComponent，可以看到在场景里面加载Actor并注册UActorComponent的时候会对UPrimitiveComponent组件进行物理信息创建。其实除了SkeletalMeshComponent以外，所有继承自UPrimitiveComponent的组件（第一部分提到的那5种都是）都会在注册后就创建物理数据（对于直接继承自UActorComponent的组件，如移动组件就不会执行该操作）。因此除了SkeletalMeshComponent以外（这个后面再分析），其他继承自UPrimitiveComponent的组件物理创建的时机都很明确，也就是UActorComponent被注册的时候创建物理。（当然还有一些特殊情况也需要更新物理，比如更换模型的时候）

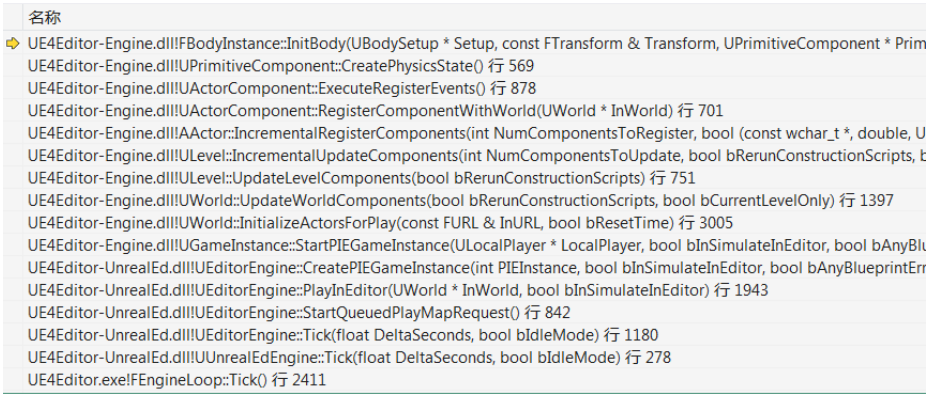


图2-1 加载场景StaticMesh物理的创建堆栈图

图2-2 玩家出生时胶囊体物理的创建堆栈图

在注册组件时是否要创建物理数据？可以参考下面代码。其实很明显的有三个条件，

1. 是否已经创建过了
2. 是否能获取到当前的物理场景（物理场景的变量为FPhysScene* PhysicsScene，理解为与游戏世界同时存在的一个物理世界。这个PhysicsScene一般是在初始化World信息，也就是在void InitWorld(const InitializationValues IVS= InitializationValues())时创建）
3. 是否应该创建 ShouldCreatePhysicsState。很明显想控制是否给组件创建对应的物理数据，写

```
    checkf(bPhysicsStateCreated, TEXT("Failed to route CreatePhysicsState (%s)"), *GetFullName());
}
```

图2-3 注册时是否创建物理的条件代码截图

2.2 USkeletalMeshComponent的物理创建

USkeletalMeshComponent与其他带物理组件不同，一般来说我们并不会在玩家一出生就创建出所有的骨骼物理，也不会让玩家的骨骼物理一直存在着。原因很简单，就是为了提升性能。对于一般的带物理的组件，我们只需要给他配置一个简单的碰撞体既可（包括Sphere，Box，Capsule等）。这样一个简单的物理组件在游戏运行时的开销是很小的，然而对于一个USkeletalMeshComponent，我们为了精确几乎需要给所有的骨骼都创建一个基本的物理单位，一旦玩家或者NPC过多，这个消耗是非常可观的。然而，我们也不能放弃使用USkeletalMeshComponent的物理，因为一旦我们的游戏想实现精准的打击，攻击不同位置的效果不同的时候，就必须要用到骨骼的物理。因此，常见的解决方案就是在需要的时候创建物理，在不需要的时候就拿掉。

> 默认引擎的SkeletalMesh物理会一直存在 参考图1-5。实际上，也不是一定要动态创建于删除skeletalMesh组件的物理，要结合游戏考虑是否要优化这一部分。

我们还是从组件的注册说起，USkeletalMeshComponent的物理的初始化与前面的组件不同，他首先重载了void USkeletalMeshComponent::CreatePhysicsState()函数。并通过调用InitArticulated函数来对所有的骨骼来进行物理的初始化，这是组件初始化时的逻辑代码。我们简单分析一下，

```
void USkeletalMeshComponent::CreatePhysicsState()
{
    //  UE_LOG(LogSkeletalMesh, Warning, TEXT("Creating Physics State (%s : %s)"), *GetNameSafe(GetOuter()),  *GetName());
    // Init physics
    if (bEnablePerPolyCollision == false)
    {
        if (!GWorldIsSaveGame)
            InitArticulated(World->GetPhysicsScene());

        USceneComponent::CreatePhysicsState(); // Need to route CreatePhysicsState, skip PrimitiveComponent
    }
    else
    {
        CreateBodySetup();
        Super::CreatePhysicsState();    //If we're doing per poly we'll use the body instance of the primitive component
    }

    |
}
```

图2-4 重载CreatePhysicsState代码截图

可以看到USkeletalMeshComponent创建物理有两个执行路径，一种是和其他组件一样使用基类UPrimitiveComponent的方法创建物理数据，另一种是用USkinnedMeshComponent里面的PhyiscsAsset数据。（bEnablePerPolyCollision这个变量默认是0，而且引擎没有修改过）

所以，可以看出，正常的USkeletalMeshComponent初始化物理是通过函数

```
void InitArticulated(FPhysScene* PhysScene,bool bForceOnDedicatedServer=false);
```

来对每一个骨骼来初始化物理的（Articulate表示关节连接的）。这里面会默认调用

```
BodyInst->CopyBodyInstancePropertiesFrom(&PhysicsAssetBodySetup->DefaultInstance);
```

根据PhysicsAsset的Default信息来初始化每一块碰撞的BodyInst，这个物理信息与蓝图中的CollisionPresets的设置没有任何关系，默认就是physicsbody。不过实际在运行的时候，碰撞检测以及射线检测使用的都是Mesh上面的CollisionPreset，与这个默认的设置无关。

```
// Create all the OutBodies.
check(OutBodies.Num() == 0);
OutBodies.AddZeroed(NumOutBodies);

for(int32 BodyIdx = 0; BodyIdx < NumOutBodies; BodyIdx++)
{
    UBodySetup* PhysicsAssetBodySetup = PhysAsset.SkeletalBodySetups[BodyIdx];
    if (!ensure(PhysicsAssetBodySetup))
    {
        continue;
    }
    OutBodies[BodyIdx] = new FBodyInstance;
```

```
check(BodyInst);

// Get transform of bone by name.
int32 BoneIndex = GetBoneIndex(PhysicsAssetBodySetup->BoneName);
if(BoneIndex != INDEX_NONE)
{
    // Copy body setup default instance properties
    BodyInst->CopyBodyInstancePropertiesFrom(&PhysicsAssetBodySetup->DefaultInstance);
    // we don't allow them to use this in editor. For physics asset, this set up is overridden by Physics Type.
    // but before we hide in the detail customization, we saved with this being true, causing the simulate always happens
    // so adding initialization here to disable this.
    // to check, please check BodySetupDetails.cpp, if (ChildProperty->GetProperty()->GetName() == TEXT("bSimulatePhysics")
    // we hide this property, so it should always be false initially.
    // this is not true for all other BodyInstance, but for physics assets it is true.
```

一般来说，USkeletalMeshComponent在每帧TickComponent的时候都会调用到USkeletalMeshComponent::RefreshBoneTransforms函数，顾名思义就是更新骨骼的坐标旋转等。

```
名称
UE4Editor-Engine.dll!USkeletalMeshComponent::RefreshBoneTransforms(FActorComponentTickFunction * TickFunction) 行 1269
UE4Editor-Engine.dll!USkinnedMeshComponent::TickComponent(float DeltaTime, ELevelTick TickType, FActorComponentTickFunctior
UE4Editor-Engine.dll!USkeletalMeshComponent::TickComponentInternal(float DeltaTime, ELevelTick TickType, FActorComponentTickI
UE4Editor-Engine.dll!USkeletalMeshComponent::TickComponent(float DeltaTime, ELevelTick TickType, FActorComponentTickFunctior
UE4Editor-Engine.dll!UActorComponent::ConditionalTickComponent(float DeltaTime, ELevelTick TickType, FActorComponentTickFunc
UE4Editor-Engine.dll!FActorComponentTickFunction::ExecuteTick(float DeltaTime, ELevelTick TickType, ENamedThreads::Type Current
UE4Editor-Engine.dll!FTickTaskSequencer::FTickFunctionTask::DoTask(ENamedThreads::Type CurrentThread, const TRefCountPtr<FGrap
UE4Editor-Engine.dll!TGraphTask<FTickTaskSequencer::FTickFunctionTask>::ExecuteTask(TArray<FBaseGraphTask *,FDefaultAllocator>
UE4Editor-Core.dll!FTaskThread::ProcessTasks(int QueueIndex, bool bAllowStall) 行 338
```

图2-5 Tick更新骨骼Transform堆栈图

RefreshBoneTransforms函数里面，可以根据CPU核数等相关参数来决定是否开一个线程来单独更新动画以及相关物理数据（最后还是调用InitArticulated函数创建物理）。

```
if (bDoParallelEvaluation)
{
    if (SkeletalMesh->RefSkeleton.GetNum() != AnimEvaluationContext.LocalAtoms.Num())
    {
        // Initialize Parallel Task arrays
        AnimEvaluationContext.LocalAtoms = LocalAtoms;
        AnimEvaluationContext.SpaceBases = SpaceBases;
        AnimEvaluationContext.VertexAnims = ActiveVertexAnims;
    }

    // start parallel work
    check(!IsValidRef(ParallelAnimationEvaluationTask));
    ParallelAnimationEvaluationTask = TGraphTask<FParallelAnimationEvaluationTask>::CreateTask().ConstructAndDispatchWhenReady(this);

    // set up a task to run on the game thread to accept the results
    FGraphEventArray Prerequistes;
    Prerequistes.Add(ParallelAnimationEvaluationTask);
    FGraphEventRef TickCompletionEvent = TGraphTask<FParallelAnimationCompletionTask>::CreateTask(&Prerequistes).ConstructAndDispatchWhenReady(this);

    TickFunction->GetCompletionHandle()->DontCompleteUntil(TickCompletionEvent);
}
```

图2-6 开启单独线程来处理动画物理数据

下面的堆栈图就是引擎通过单独开一个线程来处理物理等数据。

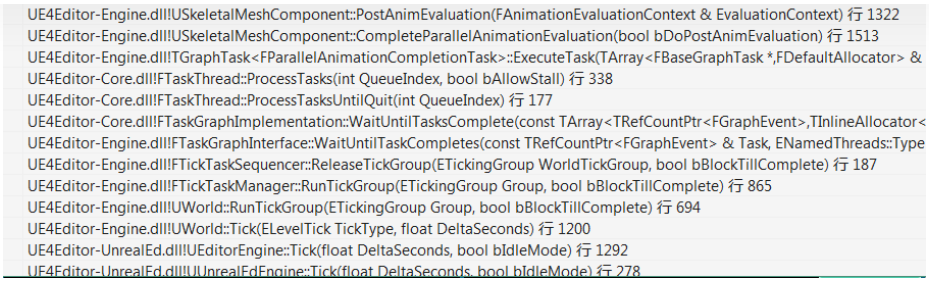


图2-7单独线程来处理动画物理数据调用堆栈图

前面我们提到要选择让物理在需要的时候去生成，而在一般状态下要拿掉。那这是如何做到的？其实我们可以在USkeletalMeshComponent::UpdateKinematicBonesToAnim 去处理，这个函数意义是根据动画的变换去更新当前的物理数据，每一帧都需要执行。基本思路就是，每帧都去检测是否需要骨骼物理数据，如果需要我们创建对应的物理数据（已经创建过了就直接返回）。如果检测到当前不再需要更新物理，就调用USkeletalMeshComponent::TermArticulated()删除物理数据。我们已经知道，运行中的物理数据全部存储在BodyInstance里面，而这个函数就会把我们当前存储在Bodies里面的所有BodyInstance数据全部清除。忘记Bodies的朋友可以回头看一USkeletalMeshComponent的类图。

同时这里还有一段注释可以参考一下：

```
// This below code produces some interesting result here
// - below codes update physics data, so if youdon't update pose, the physics won't ha
// - but if we just update physics bone withoutupdate current pose, it will have stale
// If desired, pass the animation data to thephysics joints so they can be used by mot
// See if we are going to need to update kinematics
```

const bool bUpdateKinematics = (KinematicBonesUpdateType != EKinematicBonesUpdateToPhysics::SkipAllBones); 可以把是否更新物理放到这个位置去处理。

另外，对于USkeletalMeshComponent，初始化物理时会bPhysicsRequiredOnDediServer等属性来控制在服务器模式下是否创建物理数据。

三．物理对象的移动

在UE里面，移动逻辑是通过移动组件来完成的。一般来说，一个可移动角色身上要至少挂一个移动组件（控制移动），一个胶囊体组件（也可以是球形等，作为移动组件控制的对象）。

胶囊体身上挂载着简单类型的物理碰撞，移动时必须要带着其身上的物理一起移动。图3-1是移动组件触发物理移动的调用堆栈。简单描述一下，移动组件通过调用胶囊体的InternalSetWorldLocationAndRotation来真实的更新其位置与旋转，更新后会通过函数OnUpdateTransform来触发其物理对象的更新，即UPrimitiveComponent::SendPhysicsTransform。这一步会执行胶囊体身上对应的BodyInstance位置与旋转的更新，再深入一些就是更新物理引擎里面的PxRigidActor对象了。

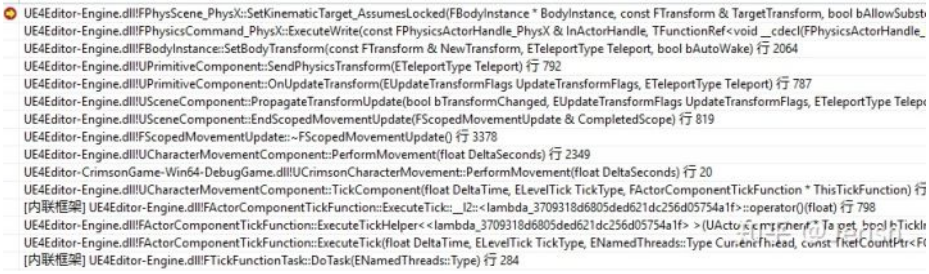


图3-1 一般的物理移动调用堆栈

动等，因为移动过程中可能出现移动不合法重置的情况。这个功能有助于提高移动性能。

```
UE4Editor-Engine.dll!FBodyInstance::SetBodyTransform(const FTransform & NewTransform, bool bTeleport) 行 2328
UE4Editor-Engine.dll!UPrimitiveComponent::SendPhysicsTransform(bool bTeleport) 行 615
UE4Editor-Engine.dll!UPrimitiveComponent::OnUpdateTransform(bool bSkipPhysicsMove) 行 608
UE4Editor-Engine.dll!USceneComponent::EndScopedMovementUpdate(FScopedMovementUpdate & CompletedScope) 行 232
UE4Editor-Engine.dll!FScopedMovementUpdate::~FScopedMovementUpdate() 行 1715
UE4Editor-Engine.dll!UCharacterMovementComponent::PerformMovement(float DeltaSeconds) 行 2178
UE4Editor-Engine.dll!UCharacterMovementComponent::TickComponent(float DeltaTime, ELevelTick TickType, FActorComponentT
UE4Editor-ShooterGame.dll!UShooterCharacterMovement::TickComponent(float DeltaTime, ELevelTick TickType, FActorCompone
```

图3-2 开启延迟更新后的调用堆栈

前面描述的是胶囊体这种简单类型的物理碰撞，如果是一个SkeletalMeshComponent组件呢？他的身上有与骨骼数量相等的BodyInstance，如何移动?

其实本质上差不多，通过堆栈可以看出USkeletalMeshComponent重写了函数OnUpdateTransform，随后会调用UpdateKinematicBonesToPhysics函数更新所有的物理数据。

```
UE4Editor-Engine.dll!USkeletalMeshComponent::UpdateKinematicBonesToPhysics(bool bTeleport, bool bNeedsSkinning, bool bI
UE4Editor-Engine.dll!USkeletalMeshComponent::OnUpdateTransform(bool bSkipPhysicsMove) 行 1440
UE4Editor-Engine.dll!USceneComponent::UpdateComponentToWorldWithParent(USceneComponent * Parent, bool bSkipPhysicsI
UE4Editor-Engine.dll!USkeletalMeshComponent::UpdateComponentToWorld(bool bSkipPhysicsMove) 行 2196
UE4Editor-Engine.dll!USceneComponent::UpdateChildTransforms() 行 981
UE4Editor-Engine.dll!USceneComponent::UpdateComponentToWorldWithParent(USceneComponent * Parent, bool bSkipPhysicsI
UE4Editor-Engine.dll!USceneComponent::InternalSetWorldLocationAndRotation(FVector NewLocation, const FQuat & RotationQu
UE4Editor-Engine.dll!UPrimitiveComponent::MoveComponentImpl(const FVector & Delta, const FQuat & NewRotationQuat, bool
UE4Editor-Engine.dll!UMovementComponent::MoveUpdatedComponentImpl(const FVector & Delta, const FQuat & NewRotation
UE4Editor-Engine.dll!UMovementComponent::SafeMoveUpdatedComponent(const FVector & Delta, const FQuat & NewRotation
UE4Editor-Engine.dll!UCharacterMovementComponent::PhysFalling(float deltaTime, int Iterations) 行 4223
UE4Editor-Engine.dll!UCharacterMovementComponent::StartNewPhysics(float deltaTime, int Iterations) 行 2530
UE4Editor-ShooterGame.dll!UShooterCharacterMovement::StartNewPhysics(float deltaTime, int Iterations) 行 821
UE4Editor-Engine.dll!UCharacterMovementComponent::PerformMovement(float DeltaSeconds) 行 2123
UE4Editor-Engine.dll!UCharacterMovementComponent::TickComponent(float DeltaTime, ELevelTick TickType, FActorComponent
```

图3-3 SkeletalMeshComponent更新移动

在更新动画的时候也会触发UpdateKinematicBonesToPhysics函数。

```
void USkeletalMeshComponent::PostAnimEvaluation(FAnimationEvaluationContext& EvaluationContext)
{
    AnimEvaluationContext.Clear();

    SCOPE_CYCLE_COUNTER(STAT_PostAnimEvaluation);
    if (EvaluationContext.bDuplicateToCacheBones)
    {
        CachedSpaceBases = SpaceBases;
        CachedLocalAtoms = LocalAtoms;
    }

    if (EvaluationContext.bDoInterpolation)
    {
        SCOPE_CYCLE_COUNTER(STAT_InterpolateSkippedFrames);

        const float Alpha = 0.25f + (1.f / float(FMath::Max(AnimUpdateRateParams.GetEvaluationRate(), 2) * 2));
        FAnimationRuntime::LerpBoneTransforms(LocalAtoms, CachedLocalAtoms, Alpha, RequiredBones);
        FAnimationRuntime::LerpBoneTransforms(SpaceBases, CachedSpaceBases, Alpha, RequiredBones);
    }

    // Transforms updated, cached local bounds are now out of date.
    InvalidateCachedBounds();

    // update physics data from animated data
    UpdateKinematicBonesToPhysics(false, true);
    UpdateRBJointMotors();
```

图3-4 更新动画时更新物理

如果开启了物理托管，那么角色的移动就完全交给物理引擎去处理。通过下面这个接口获取物理引擎返回的Transform并更新自己的位置。

```
void UPrimitiveComponent::SyncComponentToRBPhysics()
{
    if(!IsRegistered())
    {
        UE_LOG(LogPhysics, Log, TEXT("SyncComponentToRBPhysics : Component not registered (%s)"), *GetPathName());
        return;
    }

     // BodyInstance we are going to sync the component to
    FBodyInstance* UseBI = GetBodyInstance();
    if(UseBI == NULL || !UseBI->IsValidBodyInstance())
    {
        UE_LOG(LogPhysics, Log, TEXT("SyncComponentToRBPhysics : Missing or invalid BodyInstance (%s)"), *GetPathName());
```

对于SkeletalMeshComponent，上面的操作只能让组件与根骨骼位置匹配。其他的骨骼还需要通过USkeletalMeshComponent::BlendInPhysics进一步计算，

```
void USkeletalMeshComponent::BlendInPhysics()
{
    SCOPE_CYCLE_COUNTER(STAT_BlendInPhysics);

    // Can't do anything without a SkeletalMesh
    if( !SkeletalMesh )
    {
        return;
    }

    // We now have all the animations blended together and final relative transforms for each bone.
    // If we don't have or want any physics, we do nothing.
    if( Bodies.Num() > 0 )
    {
        BlendPhysicsBones( RequiredBones );

        // Update Child Transform - The above function changes bone transform, so will need to update child transform
        UpdateChildTransforms();

        // animation often change overlap.
        UpdateOverlaps();

        // New bone positions need to be sent to render thread
        MarkRenderDynamicDataDirty();
    }
}
```

图3-6

四．UE4与PhysX

前面我们已经了解到BodyInstance在UE逻辑里是一个运行时的物理的基本单位。而实际在PhysX引擎中，也同样存在一个物理基本单位，这个物理单位就PxRigidActor。一个BodyInstance对应一个PxRigidActor（实际上就是BodyInstance::InitBody时创建一个对应的PxRigidActor），这样我们就可以将UE引擎与PhysX引擎结合起来使用了。

> 这个时候，我再提出一个问题，真正的物理碰撞是如何检测的呢？

这个问题确实值得我们深思，而且不同情况下检测的方法是不一样的。举个例子，想知道两个球是否产生碰撞，那么只要判断两个球心的距离就可以了。而两个复杂模型的碰撞，可能需要通过判断两个三角面是否有交集来判断。我这里提出这个问题，只是想提醒大家，物理引擎里面的Actor也一样需要知道其本身的形状，然后进一步来处理碰撞逻辑。所以，在创建一个基本物理单位PxRigidActor之后，我们还需要给其创建基本的几何形状（在引擎里面叫做Shape），这个逻辑的处理就在函数UBodySetup::AddShapesToRigidActor（新版本叫UBodySetup::AddShapesToRigidActor_AssumesLocked）。看到这个函数，我们就知道Shape是通过UBodySetup来创建的，同时这个几何形状的数据也是存储在UBodySetup里面的。

PhysX里面提供的类型有下面几种，官方声称前四种是简单碰撞，第五种是复杂碰撞，而实际上凸面体碰撞的处理与三角面相似，所以也可以理解为复杂碰撞：

1. PxSphereGeometry 球形
2. PxBoxGeometry 盒子
3. PxCapsuleGeometry 胶囊体（SkeletalMesh常用）
4. PxConvexMeshGeometry 凸面体
5. PxTriangleMeshGeometry 三角面

4.1 简单碰撞的物理创建

这5种类型里面，前4种的生成好的物理数据都存储在UBodySetup的FKAggregateGeom AggGeom里面，按照官方文档的分类，我们称他们为简单碰撞类型。实际上，凸面体的碰撞处理并不像前面几个那样简单。

GenerateSphylAsSimpleCollision分别将碰撞数据添加AggGeom的SphereElems，BoxElems，SphylElems里面。正如我前面所说的那样，判断两个球体是否碰撞很容易，所以这几种碰撞类型不需要很复杂的数据来记录与处理，PhysX引擎可以很容易的获取到这些碰撞类型对应的数据并做处理。

凸面体与前面三种碰撞类型都不同。由于其可以通过配置生成一个较为复杂的碰撞，而且碰撞体的顶点都是通过算法生成的，所以他需要经过一个物理Cook的过程。这个过程类似渲染，把所有的三角面的顶点信息和索引提供给PhysX引擎随后PhysX利用这些数据Cook出一个完整的碰撞模型，不过这个过程需要一定的时间来执行。一般来说，我们在游戏编辑器添加AutoConvexCollision碰撞并执行Apply的时候，就会执行这个凸面体的Cook过程。Cook的过程与三角面的Cook过程相似，后面再详细分析。（下图是简单类型碰撞的添加）

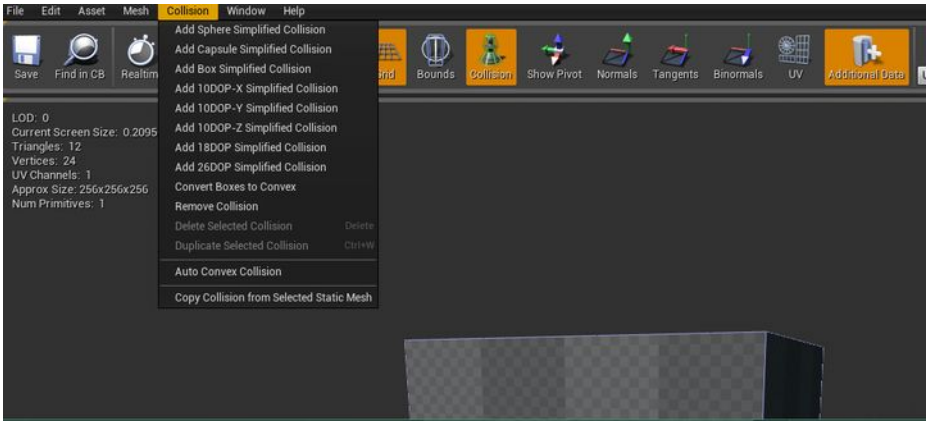


图4-1

我们知道SkeletonMeshComponent里面使用PhysicsAsset来创建骨骼动画的物理，那PhysicsAsset里面的BodySetup里面的数据是如何初始化的？他里面的shape类型是什么？看下面的两个堆栈，当我们导入一个带有动画的骨骼资产时，首先会判断该SkeletalMesh有没有物理资产。

如果没有就会调用图3-2创建物理资产，随后执行第二步，根据每个骨骼初始化对应的物理。当前引擎中，会针对没有物理资产的SkeletalMesh的每个骨骼默认初始化一个胶囊体类型的简单碰撞。这个类型是通过FPhysAssetCreateParams NewBodyData;初始化后作为参数传递给物理资产的，所以一般来说我们的角色的物理资产都是胶囊体的。

图4-2

图4-3

当然，当你导入模型之后。你也可以根据你的骨骼资产创建一个新的PhysicsAsset，在创建的时候右键骨骼资产文件——Create——CreatePhysicsAsset，随后会弹出下面的界面，可以根据需求针对每个骨骼创建一个指定类型的Shape碰撞。如果想单独调整个别骨骼的碰撞，就要打开PhysicsAsset在编辑界面里单独处理了。（SkeletonMeshComponent的物理并不一定就是简单的胶囊体碰撞，也可能复杂碰撞，下个小结分析）

图4-4

图4-5

4.2 复杂碰撞的物理创建

最后一种碰撞类型是复杂类型，那么有多复杂呢？其实就是根据Mesh的网格信息（也就是三角面的数量）来进行物理的生成，所以模型面数越多那自然就越复杂。生成好的三角面的物理数据都存储在UBodySetup的TriMesh与TriMeshNegX里面。

图4-6

看过官方文档碰朋友肯定知道，我们可以通过StaticMesh里面Collision Complexity设置来改变其碰撞的复杂度，当我们标记为UseComplexAsSimple的时候，其实就会在此时去除简单碰撞，并给对应的Mesh资产创建一份复杂的物理碰撞，这个时候就会执行三角面的Cook过程。

图4-7

图4-8

看上面的函数堆栈，在更新上面的配置的时候需要重新创建PhysicsMeshs并获取到CookData。GetCookedData就会调用FDerivedDataPhysXCooker里面的FDerivedDataPhysXCooker::BuildConvex处理凸面体物理或者FDerivedDataPhysXCooker::BuildTriMesh处理三角面的物理数据。更深入一步，在这个两个函数里又分别通过PhysX引擎IPhysXFormat接口里面的PxCooking调用cookConvexMesh函数以及cookTriangleMesh函数。

另外，想要执行Cook，我们一定要准确的获取到碰撞模型的所有顶点信息，对于StaticMesh三角面碰撞。这个顶点信息就是通过渲染的Lod信息来取到的，具体的操作在函数GetPhysicsTriMeshData里面，执行堆栈如下：

图4-9

复杂物理的生成与渲染很像，如果你需要动态的去生成与删除物理，那么一定要慎重考虑这个动态创建的过程消耗如何？我们平时对StaticMesh物理的Cook过程都是在编辑器里面就完成了。如果游戏中做这个操作很有可能造成卡顿，如果非要这么做也可以考虑使用异步线程FNonAbandonableTask来执行这个过程。

官方文档上可以看到静态模型是如何创建复杂物理的，但是好像没有说骨骼资产如何创建，骨骼模型是不是不可以创建？并不是。在4.8以后的版本里，我们打开骨骼资源文件找到bEnablePerPolyCollision属性并勾选既可。在旧版本4.5里面需要到角色蓝图找到对应的SkeletalMesh组件里勾选bEnablePerPolyCollision属性。

> 注：新版本skeletalMesh组件里也有这个属性，但是勾选无效，这是引擎的一个Bug

4.3 物理创建的后续工作

前面的操作是将函数UBodySetup::CreatePhysicsMeshes()展开，将物理Cook的过程执行完毕。随后，UBodySetup::AddShapesToRigidActor函数会获取AggGeom以及TriMesh里面已经Cook好的数据，创建对应的物理Shape。

另外，我们在创建物理的时候还分为静态与动态两种，他们通过组件上的OwnerComponent->Mobility
!= EComponentMobility::Movable来控制。很明显，静态碰撞与动态碰撞的消耗是不同的。

```
//创建静态PxRigidActor
GPhysXSDK->createRigidStatic(PTransform);
//创建动态PxRigidActor
GPhysXSDK->createRigidDynamic(PTransform);
```

截止到这里，我们已经基本上完成了物理数据的初始化。然而，我们知道在游戏里面，还有很多详细的设置，比如碰撞通道，碰撞类型等。这些数据也必须要及时更新与处理，这些逻辑与相关标记的处理在

csShapeFilterData()

实际上我们的平时做的碰撞设置CollisionEnabled对应到Physx里面就是这两个操作。

```
PShape->setFlag(PxShapeFlag::eSCENE_QUERY_SHAPE,true);
PShape->setFlag(PxShapeFlag::eSIMULATION_SHAPE, false);（参考后面的结构体）
```

图4-10

PhysX里面的Shape标记。

```
struct PxShapeFlag
{
  enum Enum
  {
    eSIMULATION_SHAPE = (1<<0),
    eSCENE_QUERY_SHAPE = (1<<1),
    eTRIGGER_SHAPE = (1<<2),
    eVISUALIZATION = (1<<3),
    ePARTICLE_DRAIN = (1<<4)
  };
};
```

关于UE与PhysX之间的交互，就简单介绍到这里。至于更详细的内容，大家有兴趣的话就去源码里面进一步了解吧。

五．物理约束Constraint

物理约束系统作为物理引擎的一项重要组成部分能够提供更真实与更丰富的模拟表现。

前面通过Mesh，胶囊体等组件总结了物理是以一个什么形式存在于游戏世界中的，UE中的物理对象与PhysX的对象是怎样的关系。但是我们没有分析游戏世界是如何模拟真实的物理现象的，比如我们常见的钟摆，弹簧，关门等。

对于两个完全分离的刚体，他们之间的作用是比较容易模拟的，只要在发生碰撞的位置给各自施加一个力就可以了，他们会根据受力情况各自进行模拟移动，不会相互影响。不过，如果我们想模拟一个钟摆的效果，目前的条件就无法完成了。所以，引擎提供了物理约束（Constraint）功能，可以将两个对象绑定在一起根据参数进行物理模拟，方便我们快速的模拟类似的效果。

在Unity中，物理约束更多的是以**关节joint**的概念存在。本质上物理约束的概念更大一些，关节是针对两个物体的约束，而物理约束可能只对一个物体产生约束。物理约束通常以关节的方式存在于引擎中。

5.1 简单理解物理约束的原理

拟效果。一个普通的刚
旋转。我们可以分别或

不过在游戏中，我们更对的是对两个对象进行物理约束（也就是关节），那么对两个物体产生物理约束有什么特点？答案是多个Actor的约束需要有特定的参照对象。一旦我们对两个对象进行约束，那么二者就必须有一个统一的约束参照对象，然后根据参照对象的坐标系来进行模拟。通常来说，这个参照对象就是ConstraintActor或者ConstraintComponent。

比如说，我用一个ConstraintActor对两个Actor进行约束，限制他们只能绕X轴旋转。不过，这两个Actor绕谁的X轴旋转？难道是世界坐标系的X轴？显然我们应该选择一个合适的可以配置的参考对象，这个对象就是上面的ConstraintActor，完成配置后Actor就会绕着ConstraintActor的X轴旋转了。

5.2 物理对象自身约束

物理自身约束比较容易理解，他只是单纯的限制自身的物理模拟的位置，不会对其他对象产生影响。所有包含物理数据的对象都可以进行设置，根据配置限制对象只能在某个轴或某个面产生模拟移动。如图5-1

图5-1

5.3 物理约束Actor

他本身是一个Actor对象，利用它可将两个Actors 连接起来（假定成一个物理模拟体），并应用限制和力度。引擎拥有一些默认关节类型 （球窝式ball-and-socket、铰链式hinge、棱柱式prismatic），区别只存在于它们的对Actor的6个自由度的限制差异。可任选一种关节开始，自行进行调整试验。里面的参数比较多，除了基本的限制操作外还可以通过Motor参数添加驱动力，后面会对一些常见的应用场景进行简单进行分析，其他的建议大家查阅相关资料后多去尝试。

图5-2

上图就是物理约束Actor的配置，一个物理约束Actor能且只能绑定两个Actor对象，这两个对象至少有一个要**开启物理模拟**。如果需要的话，我们也可以将一个SkeletalMesh的骨骼与另一个Actor绑定，甚至我们还可以指定一个Actor的某个组件与另一个Actor的某个组件绑定。具体的操作草考官方文档。

我们还可以对一个Actor进行多次约束绑定来模拟更多效果，举个例子：假如我要模拟一个秋千，intActor绑定秋千上面
or都设置允许3个自由度

其两个方向都不能旋转）。当玩家用道具砍掉一边的绳子后，对一个ConstraintActor解绑，就可以模拟一边被砍断的情形了。

5.4 物理约束组件

物理约束组件（Physics Constraint Components）的使用方法和 物理约束 Actors 相同，不同之处是其在蓝图中使用，可在 C++ 中进行创建。物理约束组件结合了蓝图的灵活和 C++ 的强大，您可利用它对项目中的任意物理形体设置约束。官方文档也有案例，不再赘述。

5.5 SkeletalMesh中的物理约束调整

打开物理资源文件（PhysicsAsset），默认是Body模式，点击按钮我们会看到有一个Constraint Mode，点击就会进入物理约束模式。

图5-3

图5-4

进入物理约束模式后，首先注意红色标记，我们有19个骨骼18个关节，同时也有18个物理约束。其实，这里可以猜出来，UE在创建对应的PhysicsAsset的同时会对每一个骨骼关节创建一个对应的物理约束，这正好符合我们的常识。

那是不是说，我们每次导入一个SkeletalMesh就会完美的创建一个带有物理约束的PhysicsAsset？并不是，每当我们根据一个SkeletalMesh创建一个物理体的时候，是会创建对应

图5-5 默认创建的物理资产

在上面的菜单位置，我们可以看到4个默认的物理约束方式，下面一一描述。

图5-6

球窝式ball-and-socket：类似上面图片的效果，一个球状的骨骼塞到凹槽里面，可以在一定空间内旋转，类似人的肘关节。下面完全开放了3个旋转自由度，但通常情况我们经常要对各个方向做一定的限制。

图5-7

铰链式hinge：类似上面图片的效果，两个物体通过铰链相连，只能绕着固定方向旋转，如门的开关。这里完全打开了绕着X轴方向的旋转。

图5-8

棱柱式prismatic：两个刚体间的角度是固定的，只能在一个固定的轴上滑动。这里我们看到只能沿着X轴产生位移。

图5-9

角色关节 Skeletal：其实与球窝式很相似，但是在各个旋转方向上都有限制。

图5-10

下图是官方根据实际情况调整过后的物理约束：可以看出来他把沿着X轴方向的旋转给禁掉了，这样更符合现实情况（可以自己试一下）。

当然，我们可以根据自己的情况做特殊处理，比如角色的骨骼上可能绑定了一个武器，那么这个武器的物理约束就可以设置成6个自由度的。不过这些全都是我们在开启角色物理模拟时才会出现的效果。

在实际应用中，我们很多情况下需要固定一个物体，然后另一个物体相对进行移动或者旋转，比如门，钟摆等的实现。所以只需要开启一个物体的simulates即可。

图5-11

关于约束限制的驱动力，可以通过Motor来添加，有位移驱动与旋转驱动。具体可以参考官方的ContentExample的PhysicsMap。

图5-12

5.6 UE中的物理约束

在UE源码里面，物理约束Actor的类是APhysicsConstraintActor函数，物理约束组件是UPhysicsConstraintComponent。APhysicsConstraintActor本身也是使用约束组件的功能。

约束组件里面最重要的数据就是FConstraintInstance ConstraintInstance，该对象包含了我们在编辑器中所见的各项参数，同时会将相关的约束数据保存到PhysX引擎中的PxD6Joint类型的数据里面。具体细节请查看源码分析。

图5-13

六．物理材质

> 物理材质在官方上这样定义：用于定义当物理对象和世界进行动态交互时它所做出的反应。

说白了，就是他在游戏世界应该是什么材料的，虽然我们通过材质的表现可以看出来他是一块木头还是一块铁，但实质上普通的材质只是在视觉上达到了效果。真正要在木头上跑步，敲击，采集等等，你肯定还需要其他的逻辑去处理。UE里面提供了物理材质便于你去定义你的游戏世界里面的物理类型（即物理材质），物理材质的创建很简单，编辑器——AddNew——Physics——PhysicalMaterial。创建之后打开就是这样的，参数很少，基本上就是设置一下摩擦系数和物理表面类型，具体可以参考官方文档的介绍。

图6-1

关于表面类型，可以打开编辑器Edit——ProjectSettings——Physics——PhysicalSurface来查看与添加。

图6-2

物理材质添加完之后，需要赋给对应的材质、材质实例、StaticMesh、SkeletonMesh等等，在各个蓝图搜索physicalMaterial既可。另外，对于每一个Mesh（实际上BodyInstance）都存在一个PhysMaterialOverride，可以覆盖你前面设置的物理材质。

最后，简单说一下一般游戏里面的使用场景。

- 第一点是走路与落地的音效，在PrimalCharacter（玩家与动物通用）里面会通过GetFootPhysicalSurfaceType函数获取脚下地面的材质类型，进而根据检测到的类型播放事先预制好的音效。
- 第二点是武器击中不同物体对应的特效与音效，思路基本上相同。

编辑于 2021-03-09 16:13

写评论 | 大钊 关注了作者

27 条评论 默认 时间

申犇
您好，补充相关知识点
1.碰撞题除您阐述的部分，还有个部分可以从外部制作的方式导入，我们一般使用UCX_StaticMesh来命名与StaticMesh同时制作，导入引擎，此类Collision针对比较复杂的形态，具体制作规范UE官方有详细文档，项目中不太建议使用默认collision！
2.关于PhyX外部导入碰撞体方面，针对布料系统，碰撞体导入UE4有个规则就是15个为上限，超过15个碰撞，对不了就不会产生作用（既显示为灰色，拥有属性的为红色现实）
3.地形碰撞方面多数使用BlockAll等自身属性，多数情况使用自定义形态使用
4.角色collision方面，根据项目需求对角色（也就是骨骼模型）对碰撞进行定义，多数情况由胶囊体，部分游戏针对性有物理属性角色，以及死亡后角色之体会产生肢体物理状态掉落也比较常见！
楼主文章很赞，学习很多知识，拙见勿怪！
2018-04-30 回复 3

Jerish 作者
感谢补充，我也有很多不了解的知识，大家一起学习进步~
2018-04-30 回复 1

申犇 ‣ **二毛**
测试为准！
2021-11-19 回复 赞

展开其他 2 条回复 >

遗落战境
谢谢分享
2018-04-16 回复 1

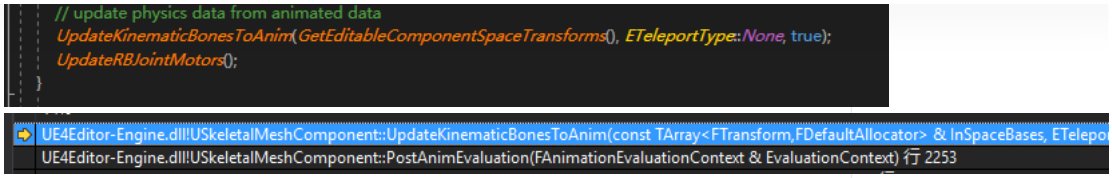
Asynchronous
请教一个问题，对于skeletalmeshcomponent，我们可以设置这个component的collision preset等配置，但是对于其绑定的physics asset中每个bone的shape（例如每个骨骼有一个胶囊体），怎么设置这些shape的collision preset，我找了一下physics asset 编辑器，没找到，最后打断点看数据发现好像这些胶囊体的碰撞预设是每个trace channel等默认的值？ 那这样来说对skeletalmeshcomponent中设置的collision preset其实并没有作用？ 因为最后的碰撞是physics asset中骨骼的shape的碰撞。
另外有办法设置physics asset中每个shape的collision preset吗
2021-02-20 回复 1

Jerish 作者 ‣ **Asynchronous**
神奇，我上次测试的时候正好项目里面有个bug，导致设置无效。今天我又用默认的原始引擎设置，发现是可以的。skeletalmesh里面的数据bodies数组的碰撞设置一直都是一个默认的值，但是检测时候用的是mesh上的配置
2021-03-09 回复 1

Jerish 作者
确实，我之前都没有发现这个问题。这些physics asset里面的碰撞都是引擎默认初始化的，与skelatalmesh里面的配置无关
2021-02-24 回复 1

展开其他 2 条回复 >

Debug
你好,在看过文章我自己进行调试时可能是版本差距我目前使用4.19.我发现官方好像将UpdateKinematicBonesToAnim与UpdateKinematicBonesToPhysics合并了.



2018-07-04 回复 赞

Jerish 作者
UE版本更新太快了，我看的应该是4.12版本，有时间我再看一下
2018-07-04 回复 1

Eagle Won
bEnablePerPolyCollision哪个引擎版本是有效的呢？似乎从来都没成功过
2018-04-15 回复 赞

Jerish 作者
很老的版本里面只有SkeletalMeshComponent上才有这个配置，更新之后SkeletalMesh资源上也有这个配置了，而且只有SkeletalMesh资源上的bEnablePerPolyCollision的这个配置有效果。这应该就是个Bug，而且一直没修复。
2018-04-16 回复 1

杨杭
老哥你加油！ 捂脸 物理模块究极有趣，PhysX业界用的最纯熟当属 FrozenByte制作的 三位一体系列
2018-04-15 回复 赞

Jerish 作者
加油！
2018-04-16 回复 赞

Asynchronous
2.1 小节这里是不是笔误了 “其实除了UStaticMeshComponent以外，所有继承自UPrimitiveComponent的组件（第一部分提到的那5种都是）都会在注册后就创建物理数据（对于直接继承自UActorComponent的组件，如移动组件就不会执行该操作）。因此除了SkeletalMeshComponent以外（这个后面再分析），其他继承自UPrimitiveComponent的组件物理创建的时机都很明确，” 应该是除了SkeletalMeshComponent以外?
2021-02-04 回复 赞

Jerish 作者
笔误了，谢谢
2021-02-24 回复 赞

Jerish 作者
谢谢，已修改
2021-02-05 回复 赞

司虎虎
一个小细节。对于BP和植被系统Instance用到的模型，他们的物理设置的优先级是高于StaticMesh本身的
2021-01-27 回复 赞

知乎用户3FNYaL
有帮助，谢谢

2020-11-10 回复 赞

Melaw
你好想问下 我将一个Actor SetHiddenInGame(true) 之后 用ShowCollison仍然能看到碰撞

回复 赞

SetCollisionEnabled

03-04 回复 赞

Melaw ‣ Jerish

那有什么办法可以把渲染和物理一起隐藏了吗🤦

03-04 回复 赞

展开其他 2 条回复 >

写评论 | 大钊 关注了作者

文章被以下专栏收录

Exploring in UE4
总结UE4使用经验，深入引擎架构原理

推荐阅读

上海买房最优解！普通人如何通过买房多赚100万?

方块说房 发表于公众号：方...

Excel VBA工作表引用方式，傻傻也分得清

Steve... 发表于SUT事务...

一次性满足你对所有幻想（vol3

Bruce