Chaos 物理引擎一览

Chaos是什么这里我就不尬吹了，相信能打开这篇文章的同学都是业内人士。我就直接开门见山的弄点干货吧。

上篇文章介绍了物理引擎的大体流程，积分、碰撞检测、约束求解。Chaos也是如此，它的核心函数

TPBDRigidsEvolutionGBF<Traits>::AdvanceOneTimeStepImpl

清晰明了的实现这些步骤，每一步对应的核心代码分别是

积分：

Integrate(…)

碰撞检测：

CollisionDetector.DetectCollisionsWithStats(…);

碰撞处理：

ApplyConstraints(…);

静止接触：

ApplyPushOut(…);

现在先不介绍这些步骤的实现细节，后面会有单独的文章讲解

本篇文章的重点在于梳理chaos的框架设计思想，帮助大家熟悉chaos的一些命名规范和概念。

同UE4引擎一样，chaos也有对应的刚体物理世界，它的类型是TPBDRigidsEvolutionGBF，之所以叫这个奇怪的名字，猜测chaos的核心思路借鉴了PBD，但又在其基础上进行了扩展(evolution)，至于缩写GBF我也没猜到是啥。缩写PBD是position based dynamic，这个什么后面讲约束的时候会说。

构成游戏物理世界最重要的元素是什么？答案显而易见——刚体，所谓的刚体，就是运动中形状和大小不变的物体，但奇怪的是，chaos没有叫RigidBody之类的对象类型，而是用了一种类似于data-oriented（面向数据编程）的方式来组织世界中的刚体。接触过粒子系统的同学应该会了解，当我们有大量粒子需要操作时，我们通常会将这些粒子放在一块连续的内存中，这样可以提高缓存命中，提高运行效率，类TPBDRigidsSOAs就是这块内存池，缩写SOA是small object allocator表示小对象分配器的意思。我们还可以对数据布局进一步优化，把所有刚体的相同属性放在一起，举个例子

原来表示一堆刚体的方法是

struct RigidBody { FVector P，FRotator R }

RigidBody Bodies[Num]

现在就可以写成 struct RigidBodies { FVector P[Num]，FRotator R[Num] }

两种方法虽然内存占用差不多，但空间排列方式差别很大，当同时操作多个刚体时，显然后者对硬件会更友好些。

Chaos便使用了这种方法，将刚体中一些经常修改的变量放在了TParticles类型以及其子类中。这里大家可能会比较奇怪，刚体为什么会叫TParticles这个名字，刚体跟粒子怎么会扯上关系。其实，这里大可不必较真，可以适当把粒子的概念放宽一些，刚体也就是一个大号的粒子，只不过比传统的粒子多了几何形状和朝向，为了更好的区分刚体和粒子，chaos从TParticles派生出了TGeometryParticlesImp类，用来表示这种有形状的粒子。该类型比较具有代表性，这里截取部分实现

CHAOS\_API TGeometryParticlesImp() : TParticles<T, d>()

{

MParticleType = EParticleType::Static;

TArrayCollection::AddArray(&MUniqueIdx);

TArrayCollection::AddArray(&MR);

TArrayCollection::AddArray(&MGeometry);

TArrayCollection::AddArray(&MSharedGeometry);

TArrayCollection::AddArray(&MDynamicGeometry);

TArrayCollection::AddArray(&MParticleIDs);

TArrayCollection::AddArray(&MShapesArray);

TArrayCollection::AddArray(&ImplicitShapeMap);

TArrayCollection::AddArray(&MLocalBounds);

可以看到MR就是这堆刚体的朝向数组，MGeometry就是几何形状数组。Chaos的几何形状基类型是FImplicitObject，其派生出了很多种形状，比如TSphere，TBox，TPlane，TConvex等等，里面有对应的几何数据以及相交检测算法，后面讲碰撞检测的时候也会涉及到这些类型。

有了形状之后，要想让刚体动起来还需要一些运动属性，比如速度，角速度等，于是从TGeometryParticlesImp派生出了TKinematicGeometryParticlesImp类型，用来表示运动(Kinematic)的物体。当然，一个真实物理刚体还需要力、力矩、质量、冲量等等这些物理属性，TKinematicGeometryParticlesImp又派生出了TRigidParticles用来表示动力学(Dynamic)刚体。最后，chaos从TRigidParticles派生出了TPBDRigidParticles，加入了一些用来方便实现PBD方法的变量，至此，一个完备的刚体类型才算成型了。派生关系如下：

TParticles

TGeometryParticlesImp

TKinematicGeometryParticlesImp

TRigidParticles

TPBDRigidParticles

可能是鉴于把属性分开放在不同数组里面不太直观，也没有单个刚体对象的概念。Chaos又弄了一套跟单个刚体对应的Handle类，名字就是TParticleHandleBase、TGeometryParticleHandleImp、TKinematicGeometryParticleHandleImp、TPBDRigidParticleHandleImp这样的，代指某个特定的刚体，其他类通过这个Handle操作刚体，不用直接操作数据

现在我们知道了chaos世界的刚体类型，那它们又是如何跟UE4世界的Actor对应起来的呢？

我们先来看看UE4如何初始化一个刚体的

游戏开始运行后，会先加载资源和数据，实现细节在UBodySetup中，这个类可以看做是一个物理资源类，它从asset里读取到一些物理数据，比如形状数据FKAggregateGeom，这是一个组合形状，里面包含FKSphereElem、FKBoxElem、FKConvexElem等形状数组。这些形状跟前面介绍的TSphere，TBox等类型是一一对应的关系。

根据UBodySetup里面的数据信息，UE4为每个Actor创建了属于自己的物理实体FBodyInstance，该实体保存在UPrimitiveComponent中。BodyInstance内有所有的物理配置属性以及跟物理引擎交互的接口。所以可以这么理解，BodyInstance就是UE4引擎层面的刚体对象。该对象内有一个关键的句柄FPhysicsActorHandle ActorHandle指向物理引擎内部的刚体对象。该句柄类型的定义是：

using FPhysicsActorHandle = Chaos::TGeometryParticle<float, 3>\*;

这里的TGeometryParticle是不是看起来跟前面的TGeometryParticlesImp以及TGeometryParticleHandleImp命名非常相似，别把它们弄混了，这个新的刚体类型相当于UE这边对chaos那边刚体类型的镜像。游戏逻辑层可以通过这个镜像，来操控chaos世界内的刚体，跟遥控器有些类似。

我们继续看初始化刚体的过程，引擎通过BodyInstance的FBodyInstance::InitBody方法将ActorHandle句柄在chaos内注册。调用顺序是

1. FBodyInstance::InitBody
2. FInitBodiesHelperBase::InitBodies()
3. FChaosScene::AddActorsToScene\_AssumesLocked()
4. FChaosEngineInterface::AddActorToSolver()
5. TPBDRigidsSolver<Traits>::RegisterObject()

上面出现了几个比较陌生的类，

FChaosScene : 可以看做是UE和chaos的中间层，所以，从某种角度来看，它可以算做UE引擎这边的物理世界

TPBDRigidsSolver : 主要负责UE和chaos的数据交互以及多线程发起调用等。

在最后一步RegisterObject中，chaos在给ActorHandle生成一个唯一的UniqueIdx之后，又创建了一个新的类型IPhysicsProxyBase，这只是个接口类，我们看它的主要派生类FSingleParticlePhysicsProxy，简单翻译就叫做刚体代理，抽象的来说，它可以看做是连通UE这边的刚体和chaos那边的刚体的桥梁，看里面的定义就会一目了然。

几个关键属性：

// UE这边的刚体，PARTICLE\_TYPE就是TGeometryParticle

PARTICLE\_TYPE\* Particle;

// chaos这边的刚体，FParticleHandle就是TGeometryParticleHandleImp

FParticleHandle\* Handle;

// 从chaos取刚体数据时的双缓冲

TUniquePtr<Chaos::IBufferResource<FStorageData>> BufferedData;

再来看几个方法：

// 把UE这边刚体数据推送到chaos那边

void PushToPhysicsState(…);

// 从chaos那边通过双缓冲机制取数据

bool PullFromPhysicsState(…);

UE其他线程和chaos线程的数据交互涉及到多线程编程，属于UE代码中相对比较复杂的部分，如何高效的实现大规模刚体的增添、修改、删除，而不会产生严重的多线程竞争问题，是一项很有挑战性的任务。为了了解chaos是怎么处理这部分的，我们来看tick的流程

Chaos的tick由游戏主线程发起，从World的StartPhysicsSim开始，依次调用

1. FChaosScene::StartFrame()
2. FPhysicsSolverBase::AdvanceAndDispatch\_External()
3. FPhysicsSolverAdvanceTask::AdvanceSolver()

这一步开始就进入物理线程了，chaos通过UE4的Task机制实现了多线程，可以主线程执行也可以视情况选择合适的线程执行。

1. TPBDRigidsSolver<Traits>::AdvanceSolverBy()
2. AdvanceOneTimeStepTask<Traits>(this, DeltaTime).DoWork()
3. TPBDRigidsEvolutionGBF<Traits>::AdvanceOneTimeStep()
4. TPBDRigidsEvolutionGBF<Traits>::AdvanceOneTimeStepImpl()

到了这里才是我们前面说的chaos核心代码，可见chaos工程之复杂

整个过程大概可以概括为主线程准备需要更新的刚体数据FPushPhysicsData，chaos线程接收到这些脏数据，如果物理世界中刚体未创建，则创建对应的刚体以及其TParticleHandleBase对象，也有可能是逻辑层控制刚体发生了位移等属性变化，这些也会作用到物理世界中，物理世界经过仿真模拟后，会把新的刚体数据写回主线程，因为在写回过程中，有可能其他线程会读取这些数据，为了能不阻塞其他线程，更好的并发执行，chaos使用了双缓冲和SRWLock读写锁的等技巧。具体实现细节这里不多介绍。

看到这里，可以发现chaos的设计结构竟然如此复杂，涉及到的类型非常多，而本文介绍的东西只是冰山一角。作为一个成熟的商业物理引擎，仅仅实现逼真的物理效果是不够的，框架设计的灵活易扩展，代码实现的运行效率高，而要想这3方面都做好，基本上是一件不可能的事，现实往往充斥着各种妥协和平衡，面对这些情况时，想必chaos的工程师们也是很无奈的吧。