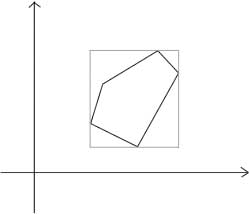
碰撞检测

上一篇文章中我们介绍了如何通过几何算法来解决碰撞检测，最后在分析三角网格的时候引出了BVH等空间加速技术SpatialAcceleration的概念，这类技术的主要目的是为了节省计算量，加速检测过程。不仅能加速单个复杂网格体，还可以用来加速场景中大量的刚体，在上篇文章介绍的几何算法执行之前，就可以利用空间加速技巧剔除掉很多不必要的检测。该优化过程被称作粗略碰撞检测，而之前介绍的几何算法属于精确碰撞检测。所以说，本篇内容更偏向于程序算法优化。对于理解整个碰撞检测过程并无大碍。鉴于本节内容与计算机算法和实现关系密切，我会着重介绍unreal chaos的源码。不关心实现的同学可以酌情跳过。

空间加速算法依赖于空间数据结构，根据不同的空间划分方法演变成了很多不同的算法，比如四叉树、八叉树、kd树、BVH等等，而且不同的树形结构可以组合使用，加速效果更佳。

在介绍空间算法之前，我们需要了解包围体的概念，因为无论何种空间算法，其管理的都不是原本的形状，而是形状的简化——包围体。包围体有很多种，引擎中最常使用的是轴对齐包围盒（AABB），简单来说就是一个跟坐标轴对齐的且大小正好可以包裹住模型的盒子。下图可以直观的看到这种盒子的样子



虽然用这种盒子来进行碰撞检测的结果会很不精确，但好在算法复杂度很低，易懂易实现，无论是拿来做碰撞检测还是构造空间树形结构都能极大的提高效率。在粗略碰撞检测阶段，AABB盒子就已经足够胜任了。

用盒子做碰撞检测的算法如下：

for (int32 i = 0; i < d; ++i)

{

if (Other.MMax[i] < MMin[i] || Other.MMin[i] > MMax[i])

return false;

}

生成AABB盒子的算法也比较简单，针对每一种形状的生成算法都不同，这些形状包括 FKSphereElem（球体）、FKBoxElem（盒子）、FKSphylElem（胶囊体）、FKConvexElem（凸包）、FKTaperedCapsuleElem（两头不一样大的胶囊体）以及FKAggregateGeom（组合体）

生成代码是FBox CalcAABB(const FTransform& Transform) const;

函数实现都比较简单，这里只介绍生成凸包的AABB的方法：

简单来说就是一句话，遍历所有顶点，然后在每个轴向上分别找出最大最小值来构成AABB盒子的最大最小顶点

截取部分UE的代码实现：

void FKConvexElem::UpdateElemBox()

{

for(int32 j=0; j<VertexData.Num(); j++)

{

ElemBox += VertexData[j];

}

}

FORCEINLINE FBox& FBox::operator+=( const FVector &Other )

{

if (IsValid)

{

Min.X = FMath::Min(Min.X, Other.X);

Min.Y = FMath::Min(Min.Y, Other.Y);

Min.Z = FMath::Min(Min.Z, Other.Z);

Max.X = FMath::Max(Max.X, Other.X);

Max.Y = FMath::Max(Max.Y, Other.Y);

Max.Z = FMath::Max(Max.Z, Other.Z);

}

}

因为我们假设物体的形状在游戏中不变，所以AABB盒子可以在游戏一开始提前算好，但我们还忽略了一个的问题，因为刚体是可以旋转的，只要朝向一变，在世界空间中的AABB盒子就会发生变化，提前计算的AABB就失效了，解决方法很简单，我们只需要在运行过程中变换提前生成的AABB盒子到世界空间中，再对其生成一遍AABB盒子就可以了，这样无论多复杂的凸包都只需要遍历8次（也就是盒子的8个顶点），比遍历凸包所有顶点效率高得多。UE重新生成世界空间AABB盒子的代码在这里：

inline TAABB<T, d> TransformedAABBHelper(const TAABB<T, d>& AABB, const TTRANSFORM& SpaceTransform)

chaos目前使用了四种空间加速结构，分别是TAABBTree、TBoundingVolume、TAABBTree和TBoundingVolume的组合，以及TBoundingVolumeHierarchy。

下面将依次介绍这些树形结构

TAABBTree

构造树的过程：假设场景中有一堆刚体

1. 分别为这些刚体生成世界空间AABB盒，创建包含所有刚体的根节点
2. 生成可以包围节点内所有刚体AABB盒的节点AABB盒
3. 找到节点包围盒在哪个轴向上长度最长，以该轴方向为基准将节点包围盒平均分成两个子AABB盒，作为两个子节点的包围盒
4. 依次遍历节点中的所有刚体AABB，判断其属于哪个子节点，如果刚体AABB正好在交界处，则根据跟哪个子节点重叠的体积大来决定。
5. 如果两个子节点都有所属刚体，则分别把两个子节点作为父节点从第2部开始下一次循环，如果有一个子节点没有刚体，则创建叶子节点，并跳出循环。

Chaos中的一些实现技巧：

1. 判断刚体跟哪个子节点AABB重叠的体积大，可以通过判断刚体AABB跟哪个子节点的交集体积小来决定。可以看下图
2. 因为世界中的刚体非常多，构造树的过程会花费很多时间，而且因为经常会发生刚体的增添删除等操作，树结构也会发生大幅度变化，更新树结构还不如重新构造来的方便，chaos采用的方法是不断的构造该树形结构，但如果在主线程每帧都执行构造树的方法，游戏帧率势必会大幅度降低。Chaos又将该构造过程放在了另一个线程，而且构造过程是分时间片的，每帧可以划分的刚体数量有上限，大约有2000个，如果超了就只能下一帧再继续了。该方法还有一点需要特别注意，另一个线程在构造树的过程中，如果世界中的刚体发生了变化，需要更新树结构，那该怎么办？chaos使用的方法是类似双缓冲的方法，主线程和构造线程都有树形结构，在构造线程构造过程中，可以先更新主线程的树形结构（注意这里并不是真的更新，而是将这些刚体缓存下来，作为独立于树的个体DirtyElements，这也意味着每次碰撞检测都要考虑这些额外的刚体，当额外刚体数量过多时，需要考虑重新构造树形结构），等另一个线程构造完了，再一股脑把这些缓存的更新用在新树上，最后将新树copy到主线程。

TBoundingVolume

这不是树形结构，而是网格体。它将空间划分成均匀的立方体单元格，刚体AABB跟哪些单元重叠，就加入这些单元的数组中。

构造过程如下：

1.遍历所有的刚体，计算所有刚体的包围盒以及立方单元格的长度，计算长度的算法既需要考虑所有刚体AABB的大小，还需要考虑这些刚体AABB的长宽高。所以，chaos采用的算法是

for 所有刚体

{

Dx += DotProduct(Vector(刚体包围盒长宽高), Vector(1,1,1)) / 3;

}

Dx = Dx / 刚体数量;

2.将包围盒按照计算出的长度划分成均匀的立方格，相当于引入了新的坐标系统，坐标的粒度就是单元格长度，遍历所有刚体AABB，计算这个AABB覆盖了哪些单元格，遍历这些单元格，将刚体的索引加入单元格的数组中，完成刚体的空间划分。

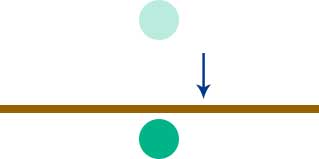
该结构比较容易理解，Chaos的实现比较简单，没有多线程时间片构造，这里就不再展开实现细节。

TAABBTree和TBoundingVolume的组合

TAABBTree作为主要数据结构，TBoundingVolume作为叶子节点。该复合结构集成了两种结构的优点。

TAABBTree在处理相隔比较远的刚体簇的时候比较有优势，但在处理一些均匀分布的情况下明显不如TBoundingVolume。TAABBTree要尽量控制层级不要过多，否则会增加递归调用次数，降低查询效率。TBoundingVolume也要保证划分粒度不能太大，所以，TAABBTree用来处理大场景，TBoundingVolume用来处理小场景，两者结合会非常有效

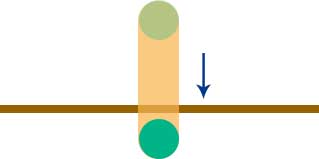
到此为止，碰撞检测部分已经基本介绍完毕。不过，我们还忽略了一种特殊情况。假设刚体的长度小于其在一帧内移动的距离，就有可能出现检测不到碰撞的情况。来看下图中的例子



还是小球自由下落，因为小球的速度很快，经过一帧的移动就直接越过了挡板。到目前为止，我们都是通过检测两个刚体是否重叠(overlap)来判断是否发生碰撞，且小球在帧开始和结束位置都不会跟挡板重叠，所以只能任由小球穿过挡板，这种现象叫做隧道效应。

为了避免这种bug，一种方法是增加检测的次数，在小球这一帧的移动路线上增加多个检测点，总会有位置可以检测到overlap。当然，为了提高检测效率，可以用二分法，这样可以减少不少检测次数。

另外一种方法比起第一种要准确高效的多，简单来说就是把刚体连续移动所覆盖的空间(扫掠体)当作新的形状作overlap检测。还是拿小球举例，其扫掠体如下图所示，正好是一个胶囊体



计算出该胶囊体与挡板的接触点后，可以直接将小球移动到该位置。同理，将小球换成其他形状也可以得到对应的扫掠体，大部分扫掠体都是不规则的凸体，使用上篇文章中介绍的GJK算法就可以算出对应的接触点。

这类解决运动中碰撞检测的方法叫做连续碰撞检测Continuous collision detection (CCD)，对应的方法名是Sweep，在成熟的物理引擎中都有比较完备的实现。