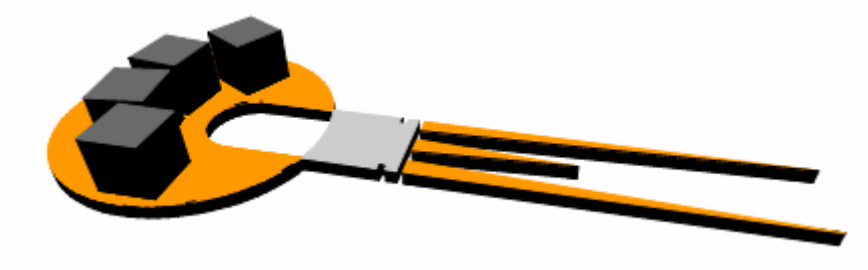
Bullet 物理引擎中文文档



英文文档正在更新中

详情查看 Wiki 和论坛（http://bulletphysics.com）

翻译：五行谦饼
Email：cqw1022@gmail.com

目录

1. 简介

1.1. 类库描述

Bullet 物理引擎是开源的，专业的集刚体、软体和碰撞检测于一身的动力学类库。在 Zlib 授权下用户可以免费用于商业开发。

1.2. 2.74 版新添加的元素

1.3. 计划

1.4. 主要特性

1） 在 Zlib 授权中使用开源的 C++代码，可免费用于包括 PLAYSTATION 3, XBox 360, Wii, PC, Linux, Mac OSX and iPhone 平台的商业开发。

2） 包括射线和凸扫测试在内的离散和连续碰撞检测，可检测的碰撞物体形状包括凹凸网格和所有的基本形状。

3） 快速和稳定的刚体动力约束和求解、动态车辆、人物控制和滑动器、铰链、普通的 6 自由度和针对碎布木偶的圆锥和扭曲约束。

4） 软体动力学方面可用于布料、绳子和双向变容的刚体，包括约束支持。

5） Maya 动态插件，集成 Blender 渲染器，支持 COLLADA 格式导入导出。

1.5. 联系和支持

1） 公开的帮助和回馈论坛 http://bulletphysics.com

2） PS3 合法的开发用户可以从 https://ps3.scedev.net/projects/spubullet 下载到用于 Cell SPU 的优化版本

2. 快速入门

1） 下载

Windows 平台的开发者可以从 http://bulletphysics.com 下载到 BulletZIPPED 格式的源代码。

MacOSX、Linux 和其他平台的开发者可以下载到 GZIPPED TAR 格式的压缩包。

2） 生成

Bullet 能在所有的平台编译，并且包含了所有的依赖项。

Windows Visual Studio 平台的所有项目文件都在 Bullet/msvc 文件夹中，主要的解决方案在 Bullet/msvc/8/wksbullet.sln 中。

3） 测试示例

可以尝试着以运行体验 Demos/AllBulletDemos 来作为学习使用 Bullet 的起点。Bullet 可以用于一下几个方面，完整刚体模拟，碰撞检测类库和（low level snippets such as GJK closest point calculation.）。这方面的帮助支持可以在 Directories 下的 doxygen documentation 中找到。

4） 集成到应用程序

仔细研究 CcdPhysicsDemo 项目中是如何创建一个 btDiscreteDynamicsWorld，btCollisionShape,btMotionState 和 btRigidBody。在动态世界中每帧调用 stepSimulation，同时动态世界能转换你的图形对象。前提条件：

#include “btBulletDynamicsCommon.h “

添加头文件路径 Bullet/Src

添加类库 libbulletdynamics, libbulletcollision, libbulletmath

5） 只整合碰撞检测类库

Bullet 的碰撞检测也可以脱离 Dynamics/Extras 而单独使用，仔细研究起步的碰撞

接口示例，尤其是 CollisionWorld 这个课程。前提条件：

#include “btBulletCollisionCommon.h”

添加路径 Bullet/src

添加类库 libbulletcollision, libbulletmath

6） 只用切片？

3. 类库概述

3.1. 简介

物理引擎的主要任务是碰撞检测，解决碰撞和其他约束，更新虚拟世界中所有物体在世界中的变换。本章将会大体上介绍一下所有部分中共享的刚体动力学管道、基本数据类型和数学类序

3.2. 软件设计

Bullet 是定制式和模块化开发的。开发者可以按如下方式自由使用

1） 只使用碰撞检测组件

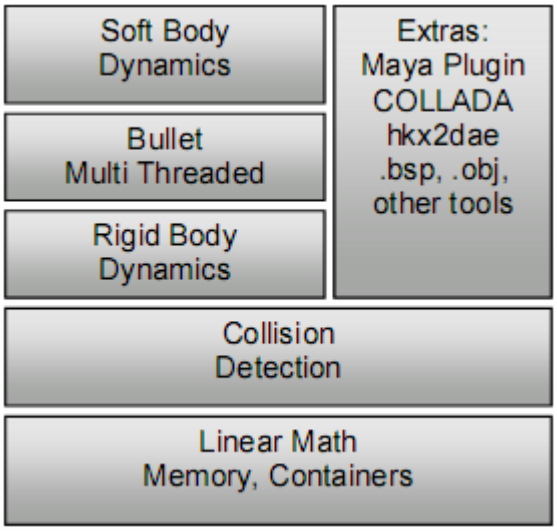
2） 使用刚体动力学组件而不使用软体动力学组件

3） 以众多方式使用类库的一部分和扩展类库

4） 选个单精度或者双精度版本的类库

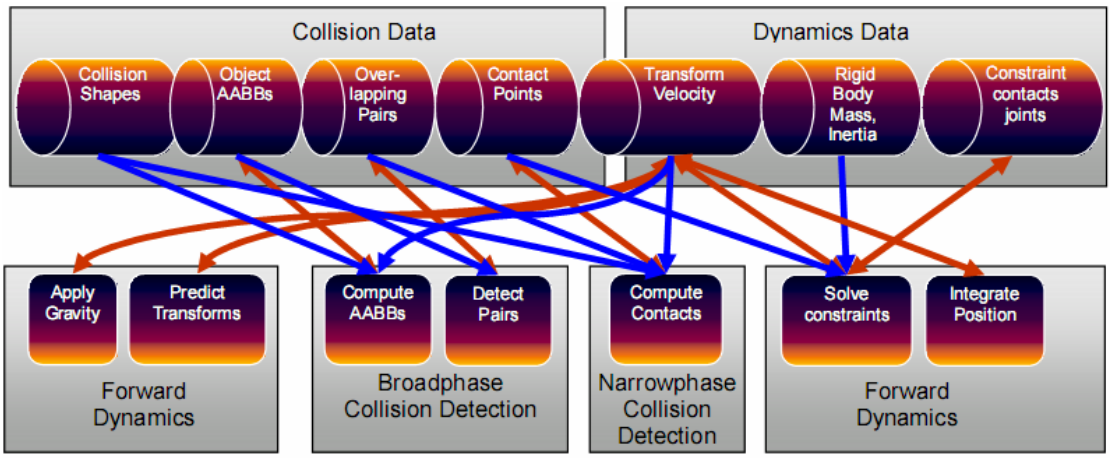
5） use a custom memory allocator, hook up own performance profiler or debug drawer

主要组件的机构如下：



3.3. 刚体物理管线

在介绍细节前，下图展示了在 Bullet 引擎动态管线中的主要的数据结构和计算阶段。管线从左到右执行，以接受重力作用开始，位置整合结束，然后更新物体在世界中的变换。



整个物理管线的计算过程和它的数据结构在 Bullet 的动态世界中完全展现出来。当在动态世界中执行 stepSimulation 时，所有上面的阶段都会执行。默认的动态世界实现是 btDiscreteDynamicsWorld。

Bullet 让开发者明确地选择动态世界的几个部分，如初测阶段（broadphase）碰撞检测，细测阶段（narrowphase）碰撞检测（分配器）和约束解决程序。

3.4. 整体概貌

如果你想在你自己的 3D 程序中使用 Bullet，你最好按照 Bullet/Demos/HelloWorld 文件夹中 HelloWorld 中的步骤来。简单地说：

创建一个 btDiscreteDynamicsWorld 或者 btSoftRigidDynamicsWorld

这写来自 btDynamicsWorld 的类提供了一个高水平管理动态物体和约束的接口，也实现了每帧更新所有物体状态的功能。

创建 btRigidBody 然后添加到 btDynamicsWorld

要创建 btRigidBody 或者 btCollisionObject，你还要提供：

1） 质量，动态移动的物体质量为正数，静态物体则为 0；
2） 碰撞形状，想盒子、球体、圆锥体、三角网格、凸壳
3） 材料的属性如摩擦系数和 restitution

每帧更新仿真情况

stepSimulation

在动态世界中调用 stepSimulation，btDiscreteDynamicsWorld 会自动通过插值取代小仿真 stepSimulation 来计算 timestep 变量，btDiscreteDynamicsWorld 使用内置的大小有 60Hertz 的 timestep。stepSimulation 会进行碰撞检测和物理仿真，通过调用 btMotionState 的 setWorldTransform 来更新主动物体在世界中的变换。

下一章将提到更多关于碰撞检测和刚体运动学的内容。许许多多的细节在 Bullet/Demos 都展示出来。如果你不能找到某个函数，请去 http://bulletphysics.com 论坛查询。

3.5. 基本数据类型和数学类库

基本数据类型，内存管理和内存容器都保存在 Bullet/src/LinearMath 文件中。

btScalar

btScalar 是 Bullet 中用来表示浮点数类型的词。为了能用当精度浮点数和双精度浮点数

编译类库，Bullet 在类库中使用 btScalar 数据类型代替浮点数类型。默认的 btScalar 是定义为 float，用户可以在自己创建的系统中或者在 Bullet/src/LinearMath/btScalar.h 文件头部定义 BT_USE_DOUBLE_PRECISION 来更改 btScalar 表示的类型。

btVector3

3D 的位置和向量都可以用 btVector3 来表示。btVector3 有 3 个 btScalar 类型的数据 x,y,z 变量组成。实际上它也有第四个因为定向和 SIMD 兼容的原因没有使用的变量 w。很多操作都很用在 btVector3 上，比如加法、减法和或获取向量长度。

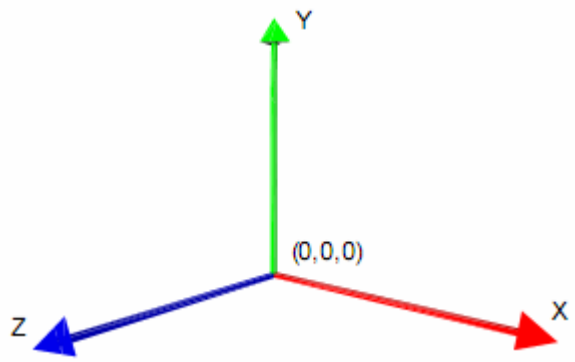
btQuaternion 和 btMatrix3x3

btQuaternion 和 btMatrix3x3 都可以用来 3D 定向和旋转。

btTransform

btTransform 是位置和方向的组合，点和方向可以用它进行坐标系转换。不允许进行裁剪和放大缩小。

Bullet 采用的是右手系坐标轴，如图：



btTransformUtil 和 btAabbUtil 提供常用的函数来进行转换和 AABBs

3.6. 内存管理、分配和容器

很多情况下数据是按 16 位分配的，比如在 CellSPU 中使用用 SIMD 或者 DMA 传输的时候。Bullet 提供默认的内存分配器来处理内存分配，用户也可以自由使用其他的内存分配器。Bullet 中使用以下方法来分配内存：

btAlignedAlloc，允许指定分配内存的大小并且和分配内存

btAlignedFree，释放由 btAlignedAlloc 分配的内存

用户可以选择下面的两个方法之一来覆盖默认的内存分配器：

btAlignedAllocSetCustom 当用户的分配器不支持内存分配的时候调用

btAligneAllocSetCustomAligned 可以用来设置用户自己的内存分配器

用户可以使用下面的宏来确保结构和类对象内存的自动分配：

ATTRIBUTE_ALIGNED16(type) variablename 创建一个 16 位的变量

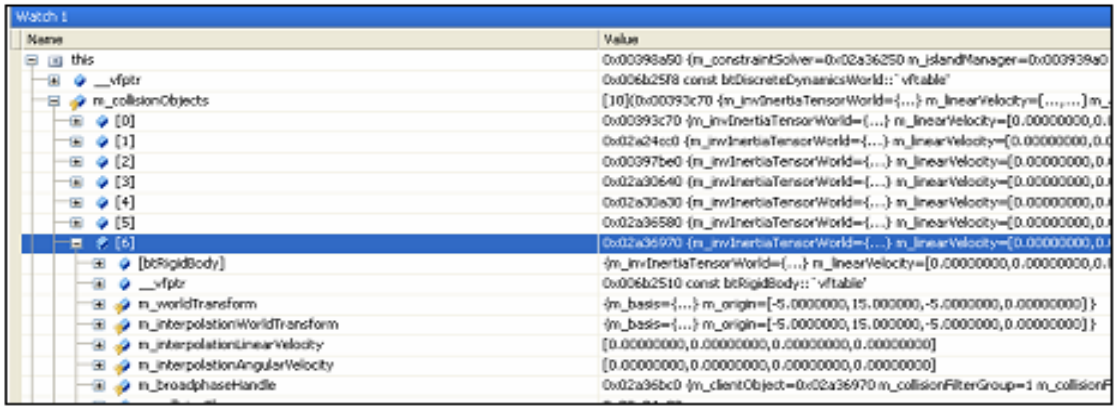
很多情况下我们都需要维护数组对象，开始的时候 Bullet 类库是用 STL std::vector 数据

结构给数组，但是为了可移植性和兼容性作者已经更换为使用自己的数组类了

btAlignedObjectArray 和 std::vector 非常相似，它使用对齐分配器来保证分配，它还提供数组排序、快速排序、堆排序的接口。

To enable Microsoft Visual Studio Debugger to visualize `btAlignedObjectArray` and `btVector3`, follow the instructions in Bullet/msvc/autoexp_ext.txt

为了使得微软的 Visual Studio 的调式器能监视到 btAlignedObjectArray 和 btVector3，请按照 Bullet/msvc/autoexp_ext.txt 中的指示去做。



3.7. 时间和性能分析

为了找出性能瓶颈,Bullte 使用宏来进行分级性能测量

1）btClock 用微秒的精度来计算时间

2）BT_PROFILE(section_name)标识一个分析的开始

3）CProfileManager::dumpAll(); 将一个分级性能完全输出在控制台中，并调用后面的仿真步骤

4）CProfileIterator 遍历性能分析树的类

当然也可以通过在 Bullet/src/LinearMath/btQuickProf.h 中定义#define BT_NO_PROFILE 1 来取消性能分析

3.8. 调试画图

视觉调试仿真数据结构相当有用，例如它可以让你验证物理仿真数据和图形数据的吻合程度，也可以现实缩放比例问题，错误的约束帧和限制问题。

btIDebugDraw 是用于调试画图的接口，来源于用户实现的 drawLine 等其他方法。

4. Bullet 的碰撞检测

4.1. 碰撞检测

Bullet 的碰撞检测系统提供的算法和加速结构来进行最近点（距离和渗透）查询、射线检测和凸扫面检测，主要的数据结构如下：

btCollisionObject：包含对象在世界中的变换情况和碰撞图形的对象

btCollisionShape： 描述碰撞图形和碰撞对象，如盒子、球体、凸壳或者三角网格。一个简单的碰撞图形可以在很多的碰撞对象中共享使用

btGhostObject：一个特殊的 btCollisionObject，用来进行局部快速碰撞查询

btCollisionWorld：保存所有的 btCollisionObject 并且提供接口来进行查询碰撞图形

在碰撞检测的初测阶段，Bullte 提供了基于重叠包围盒的加速结构去快速排除对象对（pairs of object）。Bullte 有几个不同的初测阶段的加速结构可以好使用：

btDbvtBroadphase：使用一个基于 AABB 树的动态层次包围体

btAxisSweep3 和 bt32BitAxisSweep3：实现增量三维扫描和裁剪

btCudaBroadphase：实现了使用 GPU 图形硬件的快速均匀网格

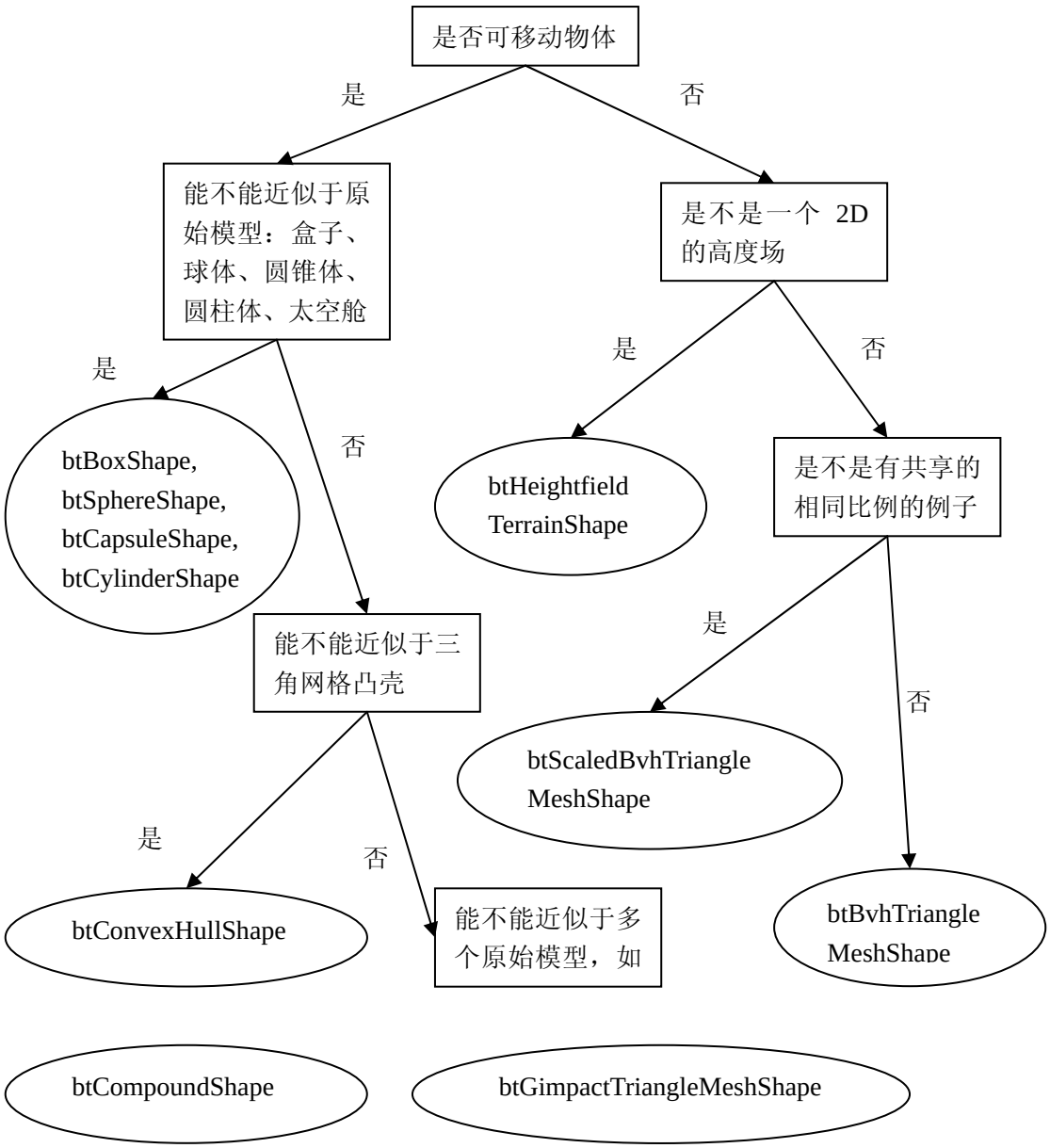
初测阶段在对象对缓存中添加和移除重叠对象对，用户可以选择对象对缓存的类型。

一个碰撞分配器迭代每个对象对，查找匹配基于所包含对象的类型的碰撞算法并且执行碰撞算法来计算接触点。

btPersistentManifold：一个用来保存给定的对象对的接触点的缓存

4.2. 碰撞图形

Bullet 支持很多不同种类的碰撞图形，也可以加入用户自己碰撞图形。为了让程序拥有更好的性能和质量，根据需要选择适当的碰撞图形很重要。下图能帮助你进行选择：



4.3. 凸原始图元

很多原始图元形状是围绕起源的局部坐标框定义的：

btBoxShape：由盒子边长的一半定义

btSphereShape：通过半径定义

btCapsuleShape::：绕着 Y 轴定义的胶囊，同理可定义绕着 X 和 Z 轴。

btCylinderShape：绕着 Y 轴定义的圆柱，同理也可定义绕着 X 和 Z 轴的

btConeShape：绕着 Y 轴的圆锥，同理也可定义绕着 X 和 Z 轴的圆锥

btMultiSphereShape：多个球体组成的凸壳，比如由两个球体组成的胶囊

4.4. 复合图形

多个凸图形使用 btCompoundShape 可以组合到一个合成图形或者复合图形，这是一个有凸图形的部件组成的凹图形，有叫子图形，每个 btCompoundShape 子图形相对于 btCompoundShape 都有自己的局部偏移转换方法。

4.5. 凸核图形

Bullet 有好几个方法支持表示凸三角网格，最早的方法是创建一个 btConverHullShape 并传给一个向量数组。大多数情况下，图形网络都包含很多的向量让 btConverHullShape 直接使用。

4.6. 凹三角网格

在静态环境中，有个非常高效的表示静态三角网格的方法是用 btBvhTriangleMeshShape。这个碰撞图形会从 btTriangleMesh 或者 btStridingMeshInterfase 中创建一个内部的加速器结构。除了在运行时创建树外，它还可以序列化为二进制树保存在硬盘当中。请看 Demos/ConcaveDemo 是怎么样保存和加载这些 btOptimizeBvh 树加速器结构的。当你有几个相同的三角网格而大小不同的示例时，你可以用 btScaledBvhTriangleMeshShape 多次实例化 btBvhTriangleMeshShape 实例。

4.7. 凸分解

理想的情况下，凹网格最好只用于做静态的品，不然的话它的凸壳应该在它的网格转换成 btConverhullshape 后使用，如果单个凸图形不够详尽，可以使用多个凸图形组合成复合物体（btCompoundShape）。凸分解是用来将凹网格分解成几个凸图形。请看 Demos/ConverDecompositionDemo 中的自动分解方法。

4.8. 高度场

Bullet 通过 btHeighfieldTerrainShape 来支持特殊情况下的平面二维凹地形，请在 Demos/TerrainDemo 查看它的用法。

4.9. Buttle 的静态平面（btStaticPlane）图形

顾名思义，btStaticPlaneShape 是表示无限的平面或者半空间，只能用于静态的不会移动的物体

4.10. 碰撞图形缩放

有些碰撞图形能有局部的缩放应用。使用 btCollisionShape::setScaling(vector3)。不同轴的缩放比例不一致的非均匀缩放值能在 btBoxShape,btMultiSphereShape,btConvexShape，btTriangleMeshShape 中能使用，不同轴使用一致的缩放比例的均匀的缩放值能在 btSphereShape 中使用。要说明的是非均匀缩放的球体能用 btScaledBvhTriangleMeshShape 来创建。前面提到国，btScaledBvhTriangleMeshShape 允许实例化一个在非均匀缩放因素下的 btBvhTriangleMeshShape 对象。btUniformScalingShape 允许实例化不同规模的凸图形来减少内存的使用总量。

4.11. 碰撞边框

Bullet 在碰撞图形中使用碰撞边框来提高碰撞检测的性能和准确度。最好不要更改默认

的碰撞边框，而且必须使用正值，边框为 0 可能会出错误。默认的碰撞边框是 0.04，也就是 4 厘米（如果你使用米为单位的话）。

4.12. 碰撞矩阵

Bullet 用分配器给每对碰撞图形分配一个特定的碰撞算法。如下图，分配器的矩阵中默认由下面的算法填充。要解释的是 GJK 算法，GJK 实际上是三个人名的缩写，分别是 Gilbert, Johnson 和 Keerthi，他们是开发凸距离算法的人，它结合了 EPA 的穿透深度的计算。EPA 全称是“Expanding Polythope Algorithm “。Bullet 有自己的免费的 GJK 和 EPA 的实现。

	box	sphere	convex,cylinder cone,capsule	compound	triangle mesh
box	boxbox	spherebox	gjk	compound	concaveconvex
sphere	spherebox	spheresphere	gjk	compound	concaveconvex
convex, cylinder, cone, capsule	gjk	gjk	gjk	compound	concaveconvex
compound	compound	compound	compound	compound	compound
triangle mesh	concaveconvex	concaveconvex	concaveconvex	compound	gimpact

4.13. 注册常用的碰撞图形和算法

用户能使用 btDispatcher::registerCollisionAlgorithm 来注册一个特定的碰撞检测算法，也能使用它来覆盖任意碰撞矩阵中的入口。Demos/UserCollisionAlgorithm 例子中注册了一个 SphereSphere 碰撞检测算法。

5. 碰撞过滤（选择碰撞）

Bullet 提供三种方法去保证只有特定的物体才有可能相互碰撞：掩码、初测阶段过滤回调和细测阶段的 nearcallbacks。值得一提的是，基于掩码的碰撞选择比回调函数发生得更多。简而言之，如果掩码能充分达到你的目的，就用掩码，它们性能更好而且用起来更加简单。

当然千万不要因为性能高那么一点点而尝试着强行将某些明显不适合与系统的对象硬加去基于掩码的碰撞选择系统中。

5.1. 用掩码来过滤碰撞

Bullet 通过掩码来判断并选择物体是否能和其他物品发生碰撞或者接受碰撞。

以飞船游戏（普通的 2D 射击游戏）为例：飞船之间是可以完全穿透，它们不会发生碰撞也不可能接收到其他飞船的碰撞；而飞船和墙之间就会发生碰撞，不过墙只能接收碰撞而不会主动碰撞其他物品，并且它在接收碰撞后没有什么影响；对于像“武器加强”的道具物品来说，他们能接收到飞船的碰撞但是不能碰撞飞船。掩码就是按照这样设计来选择碰撞的。

为了利用 Bullet 的这个功能，用户需要一个用位表示的掩码来设置墙、飞船和道具：

```
#define BIT(x) (1<<(x))
enum collisiontypes {
    COL_NOTHING = 0, //<Collide with nothing
    COL_SHIP = BIT(1), //<Collide with ships
    COL_WALL = BIT(2), //<Collide with walls
    COL_POWERUP = BIT(3) //<Collide with powerups
}
int shipCollidesWith = COL_WALL;
int wallCollidesWith = COL_NOTHING;
int powerupCollidesWith = COL_SHIP | COL_WALL;
```

设置完后，你只需要简单地把物体添加到虚拟世界中，还有第二和第三个参数中的物体碰撞类型和碰撞掩码。

```
btRigidBody ship; // Set up the other ship stuff
btRigidBody wall; // Set up the other wall stuff
btRigidBody powerup; // Set up the other powerup stuff
mWorld->addRigidBody(ship, COL_SHIP, shipCollidesWith);
mWorld->addRigidBody(wall, COL_WALL, wallCollidesWith);
mWorld->addRigidBody(powerup, COL_POWERUP, powerupCollidesWith);
```

值得一提的是如果你使用了掩码并且它们完全符合你的需求，那么你就不用使用通常的碰撞过滤器了。

如果你有比上述掩码中位容量更多类型的物体，或者是有写碰撞是根据某些因素来判断是否激活，那么 Bullet 中有几种方法去注册回调来实现通常的逻辑并且只会在用户想要的碰撞检测上调用。

5.2. 用初测阶段的回调过滤器来过滤碰撞

过滤碰撞的一个高效方法是注册一个初测阶段的回调过滤器。回调过滤器在碰撞管线的早期就被调用，同时预防碰撞对的产生。

```
struct YourOwnFilterCallback: public btOverlapFilterCallback
{
// return true when pairs need collision
virtual bool needBroadphaseCollision(btBroadphaseProxy* proxy0,btBroadphaseProxy*
proxy1)
const
  {
    bool collides = (proxy0->m_collisionFilterGroup & proxy1->m_collisionFilterMask)!= 0;
    collides = collides && (proxy1->m_collisionFilterGroup & proxy0-
    >m_collisionFilterMask);
    //add some additional logic here that modified 'collides'
    return collides;
  }
};
```

然后创建这个类对象并且注册这个回调使用。

```
btOverlapFilterCallback * filterCallback = new YourOwnFilterCallback();
dynamicsWorld->getPairCache()->setOverlapFilterCallback(filterCallback);
```

5.3. 用细测阶段的 NearCallBack 来过滤碰撞

在碰撞对经过初测阶段的初步赛选后，可以在细测阶段注册另一个回调函数。btCollisionDispatcher::dispatchAllCollisionPairs 会在 btCollisionDispatcher::needsCollision 检测中调用细测阶段的 NearCallBack 来检测每个碰撞对。用户也可以添加自己的过滤回调函数，方法如下：

```
void MyNearCallback(btBroadphasePair& collisionPair,
   btCollisionDispatcher& dispatcher, btDispatcherInfo& dispatchInfo) {
      // Do your collision logic here
      // Only dispatch the Bullet collision information if you want the physics to continue
      dispatcher.defaultNearCallback(collisionPair, dispatcher, dispatchInfo);
}
mDispatcher->setNearCallback(MyNearCallback);
```

5.4. 从 btCollisionDispatcher 派生你自己的类

为了使得碰撞分配拥有更细粒的控制，你可以从 btCollisionDispatcher 派生出自己的类并覆盖一个或者多个下列方法：

```
virtual bool needsCollision(btCollisionObject* body0,btCollisionObject* body1);
virtual bool needsResponse(btCollisionObject* body0,btCollisionObject* body1);
virtual void dispatchAllCollisionPairs(btOverlappingPairCache* pairCache,const
                btDispatcherInfo& dispatchInfo,btDispatcher* dispatcher);
```

6. 刚体动力学

6.1. 简介

刚体动力学实现在碰撞检测模块上层，它给物体添加作用力、质量、惯性速度和约束。

btRigidBody 是主要的刚体类型，移动的刚体惯性和质量不能为 0，btRigidBody 派生与 btCollisionObject，它继承了 btCollisionObject 中的物体在世界中的变换情况、摩擦系数和 restitution 并且线速度和角速度。

btTypedConstraint 是用于刚体约束的基类，包括 btHingeConstraint,btPoint2PointConstraint ,btConeTwistConstraint, btSliderConstraint 和 btGeneric6DOFconstraint。

btDiscreteDynamicsWorldis 派生与 btCollisionWorld，并且是给 stepSimulation 进行仿真的容器

6.2. 静态、动态和运动学上的刚体

Bullet 中一共有三种不同的物体类型

1） 动态的刚体

拥有大小为正的质量

每个仿真帧都会更新自己的在世界中的变换

2） 静态的刚体

0 质量

不能移动但可以碰撞

3） 运动学上的刚体

0 质量

用户能用它们制作动画，但它只有一方面的作用：推动刚体但是刚体对它没有任何影响。

它们都必须要手动添加到动态的虚拟世界中才能使用。刚体能分配到一个碰撞图形，利用这个图形能计算质量分布也叫惯性张量。

6.3. 质心在虚拟世界中的变换

在 Bullet 引擎中，刚体在虚拟世界中的变换等同于质心的变换，根据质心还定义了物体的局部惯性参考系。局部惯性张量取决于物体的形状，同时如果给定质量那 btCollisionShape 就可以提供计算局部惯性的方法。

虚拟世界变换必须是指一个刚体的变换，这意味着它应该不包含缩放和裁剪。如果希望缩放一个物体对象，你可以缩放碰撞图形，如果必要的话，其他的转换信息如裁剪可以应用到三角网格的向量中。

为了防止碰撞图形不符合质心变换，你可以用 btCompoundShape 将他转化到吻合，同时利用子变换来转换自碰撞图形。

6.4. 什么是运动状态（MotionStates）

运动状态是 Bullet 用来获取即将在程序中渲染的物体在虚拟世界中的变换。

很多时候，在渲染每帧之前，你的游戏循环会迭代所有的物体，对于即将渲染的物体来说，游戏会根据动力学原理来更新它的位置。Bullet 就是用 MotionStates 来保存你的程序在这方面的劳动成果。

在程序中利用运动状态有很多好处：

1） 每帧中计算移动物体变换只是针对移动过的物体，而不会理会那些即将渲染但是确分毫不动的物体。

2） You don't just have to do render stuff in them. They could be effective for notifying network code that a body has moved and needs to be updated across the network.

3） 一般来说，插值只是使得在屏幕显示运动物体时更加平滑，Bullet 会利用 MotionStates 来管理物体的运动插值。

4） 可以监视图形对象和质心变换之间的转换。

5） MotionStates 用起来相当简单。

6.5. 插值

Bullet 知道怎么在物体运动时添加插值，上面提到过，Bullet 是通过 MotionStates 来处理插值的。

如果你想知道物体的位置，通过 btCollisionObject::getWorldTransform 或者 btRigidBody::getCenterOfMassTransform 可以得到物体最后一次物理运动最终状态的位置（没有插值），这很有用，但是为了渲染得更好看，用户要获取一些插值。在变换传递给 setWorldTransform 之前 Bullet 会在物体的变换中插值。

6.6. 如何使用 MothonStates

Bullet 里有两个地方用到 MotionStates：

1） 第一次是刚开始创建物理的时候，在物体进入虚拟世界前，Bullet 会从 MotionState 中获取物体的初始位置

Bullet 会调用 getWorldTransform ，getWorldTransform 会引用一个让用户填写的变换信息的变量。

Bullet 也会对在运动学上的物体调用 getWorldTransform。详情参阅下一节

2） 在创建完物体后，在每次仿真之间 Bullet 会调用 Motionstate 来让物体运动。

Bullet 调用 setWorldTransform，其中包含物体的变换信息，这是为了让用户适当地更新物体

实现一个 MotionState，只需要简单地继承 btMotionState 并且覆盖 getWorldTransform 和 SetWorldTransform。

6.7. DefaultMotionState 介绍

尽管推荐用户没有必要从 btMotionState 接口中派生自己的 MotionState，Bullet 还是提供了一个默认的 motionstate 接口来让用户实现自己的 MotionState。用户可以简单地用物体的默认变换来构建 motionstate：

```
btDefaultMotionState* ms =new
btDefaultMotionState(btTransform(btQuaternion(0,0,0,1),btVector3(0,10,0)));
/* The constructor has default parameters that are the identity.
```

如果你只是想新创建一个物体，你可以构建一个没有参数的 btDefaultMotionState。

6.8. Ogre3D 的 MotionState 示例

Since Ogre3d seems popular, here's a full implementation of a motionstate for Bullet. Instantiate it with a the initial position of a body and a pointer to your Ogre SceneNode that represents that body. As a bonus, it provides the ability to set the SceneNode much later. This is useful if you want an object

现在 Ogre3d 引擎非常火，下面将展示一个用 Ogre3d 完整实现 Bullet 的 Motionstat 的例子。其中的 btTransform mPos1 是物体的初始位置，Ogre::SceneNode *mVisibleobj 是用来渲染物体的。另外它还提供了一个 Ogre::SceneNode *mVisibleobj 对象的 set 函数，用它你可以根据需要来设置。

```
class MyMotionState : public btMotionState {
public:
    MyMotionState(const btTransform &initialpos, Ogre::SceneNode *node) {
        mVisibleobj = node;
        mPos1 = initialpos;
    }
    virtual ~MyMotionState() {      }
    void setNode(Ogre::SceneNode *node) {
        mVisibleobj = node;
    }
    virtual void getWorldTransform(btTransform &worldTrans) const {
        worldTrans = mPos1;
    }
    virtual void setWorldTransform(const btTransform &worldTrans) {
        if(NULL == mVisibleobj) return; // silently return before we set a node
        btQuaternion rot = worldTrans.getRotation();
        mVisibleobj->setOrientation(rot.w(), rot.x(), rot.y(), rot.z());
        btVector3 pos = worldTrans.getOrigin();
        mVisibleobj->setPosition(pos.x(), pos.y(), pos.z());
    }
protected:
    Ogre::SceneNode *mVisibleobj;
    btTransform mPos1;
};
```

6.9. 运动体

如果您打算做动画或移动静态对象，您应标记这些对象为运动体，同时在动画中禁用它们的睡眠/钝化功能。也就是说 Bullet 动态虚拟世界每个仿真帧都会从 btMotionState 中获得新的世界变换。

```
body->setCollisionFlags( body->getCollisionFlags() |
btCollisionObject::CF_KINEMATIC_OBJECT);
body->setActivationState(DISABLE_DEACTIVATION);
```

如果你使用运动体，那么 getWorldTransform 就会每帧调用一次，也就意味这你的运动体的 MotionState 应该有将自己当前位置信息添加到 MotionState 的功能。

6.10.仿真帧和插值帧

Bullet 里面默认的虚拟帧速率是 60Hertz（60 帧/秒，0.016666 秒/帧），但是游戏或者应用程序可能会采用不同的甚至是多个虚拟帧速率，为了将应用程序的帧速率和 Bullet 中的虚拟帧速率分离，Bullet 在 stepSimulation 内嵌了一个自动插值方法，当应用程序的时间增量小于 Bullet 内部的时间步长时，Bullet 给世界一个插值变换，然后发送插值变换给 btMotionState 但不进行物理模拟。如果应用程序的时间增量大于 bullet 的时间步长，那么每次调用 stepSimulation 就会有超过 1 个的虚拟步运行。用户可以通过传递一个最大的值作为第二个参数限制模拟步骤的最大数目。

刚体创建后能调用 btMotionState::getWorldTransform 从 btMotionState 中.取得初始世界变换。如果仿真已经开始运行，也就是调用 stepSimulation 时，通过调用 btMotionState::setWorldTransform 来更新已经激活的刚体的世界转换。

动态刚体都有大小是正数的质量，它们的运动由虚拟系统决定，静态和运动学上的刚体没有质量。静态物体无论什么时候都不能被用户移动。

7. 约束

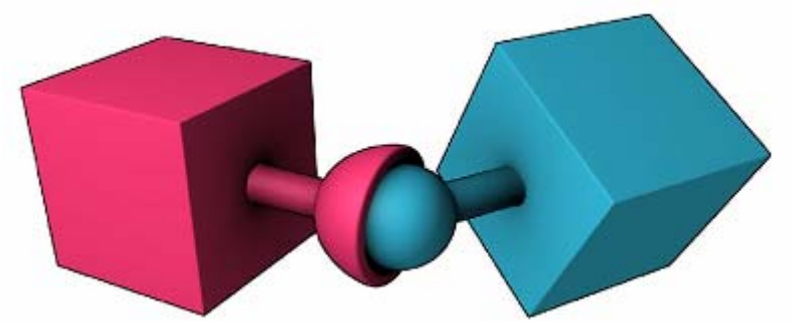
Bullet 实现了几个约束，用户可以从 Demos/ConstraintDemo 中看到所有约束的实例。包括 btRaycastVehicle 在内的所有的约束都是从 btTypedConstraint 中派生。约束只能用在两个刚体之间，而且必须至少有一个是动态的。

7.1. 点对点约束

点对点约束，也叫球窝关节（球窝接头）限制了两个刚体的平移，所以它们的局部枢纽点能在世界空间中相互吻合。使用这个约束能将一连串的刚体联系起来。

btPoint2PointConstraint(btRigidBody& rbA,const btVector3& pivotInA);

btPoint2PointConstraint(btRigidBody& rbA,btRigidBody& rbB, const btVector3& pivotInA,const btVector3& pivotInB);



点对点约束

7.2. 铰链约束

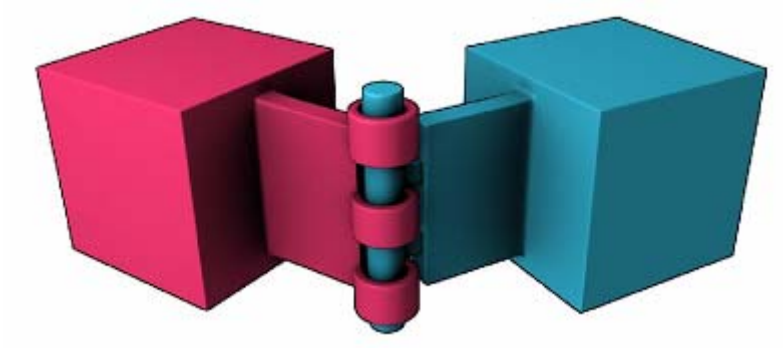
铰链约束，也叫选择关节限制了另外两个角的自由度，因此物体只能绕着一个轴——铰链轴旋转。这个可以应用于模拟门、齿轮的旋转。用户可以指定铰链的限制和动力（limits and motor）

btHingeConstraint(btRigidBody& rbA,const btTransform& rbAFrame, bool useReferenceFrameA = false);

btHingeConstraint(btRigidBody& rbA,const btVector3& pivotInA,btVector3& axisInA, bool useReferenceFrameA = false);

btHingeConstraint(btRigidBody& rbA,btRigidBody& rbB, const btVector3& pivotInA,const btVector3& pivotInB, btVector3& axisInA,btVector3& axisInB, bool useReferenceFrameA = false);

btHingeConstraint(btRigidBody& rbA,btRigidBody& rbB, const btTransform& rbAFrame, const btTransform& rbBFrame, bool useReferenceFrameA = false);

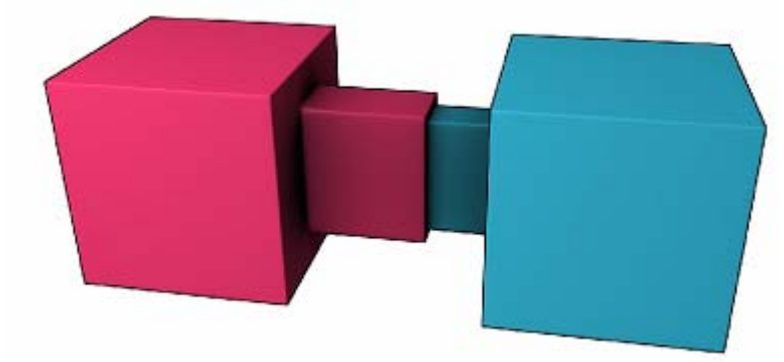


铰链约束

7.3. 滑动约束

滑动约束允许物体绕着一个轴旋转和沿着这个轴平移

btSliderConstraint(btRigidBody& rbA, btRigidBody& rbB, const btTransform& frameInA, const btTransform& frameInB ,bool useLinearReferenceFrameA);



滑动约束

7.4. 锥扭约束

创建一个碎布木偶，锥扭约束对模拟想上肢这样的肢体非常有用，他是一个给轴加了圆锥和扭曲约束限制的特殊的点对点约束，其中 X 轴作为扭曲轴。

btConeTwistConstraint(btRigidBody& rbA,const btTransform& rbAFrame);

btConeTwistConstraint(btRigidBody& rbA,btRigidBody& rbB,const btTransform& rbAFrame, const btTransform& rbBFrame);

7.5. 通用 6 自由度约束

通过配置六个自由度，这个通用的约束可以模仿很多其他标准的约束。前面 3 个自由度是线性自由度，是用来表示物体的平移，后面 3 个则是表示物体的转动。每个自由度都有 3 种状态：locked（锁死）、free（自由）和 limited（限制）。新建的 btGeneric6DofConstraint 对象所有的自由度都处于 locked 状态。

btGeneric6DofConstraint(btRigidBody& rbA, btRigidBody& rbB, const btTransform& frameInA, const btTransform& frameInB ,bool useLinearReferenceFrameA);

下面是范例：

```
btVector3 lowerSliderLimit = btVector3(-10,0,0);
btVector3 hiSliderLimit = btVector3(10,0,0);
btGeneric6DofConstraint* slider = new
btGeneric6DofConstraint(*d6body0,*fixedBody1,frameInA,frameInB);
slider->setLinearLowerLimit(lowerSliderLimit);
slider->setLinearUpperLimit(hiSliderLimit);
```

对于每个自由度：

Lowerlimit == Upperlimit -> axis is locked.

Lowerlimit > Upperlimit -> axis is free

Lowerlimit < Upperlimit -> axis it limited in that range

8. Action：车辆和人物控制器

8.1. Action 接口

有些时候，将通用的动力学方面的代码放到动力学管道里面去处理会很有用。当需要更新几个物体时，对于用户来说从接口 btActionInterface 中派生出自己的常用类比用回调函数更方便，同时实现 btActionInterface::updateAction(btCollisionWorld* world, btScalar deltaTime)。这些可以在 btRaycastVehicle 和 btKinematicCharacterController 这两个应用了 btAcionInterface 的例子中看到。

8.2. 车辆投射

对于大多数车辆仿真，Bullet 更推荐使用像 btRaycastVehicle 例子的简单车辆模型，它并不是将车辆的轮子和底盘分别用刚体表示后用约束连接起来，而是只用一个简单的模型，这样可以有很多便利，并且已经在商业的驾驶游戏中广泛使用。

整个车辆用单个刚体来表示，车轮的碰撞检测则是由射线投射来近似判断，而轮胎的摩擦系数则是一个基本的各向异性的摩擦模型。

详情请看 src/BulletDynamics/Vehicle 和 Demos/VehicleDemo，或者在论坛中询问。

要改变车辆向上的轴，请看 VehiceDemo 例子中的#define FORCE_ZAXIS_UP

8.3. 角色控制器

游戏中的玩家和 NPC 角色可以用一个胶囊、球体或者其他形状来创建。如果不要旋转的话，可以将“angular factor”设置为 0，这样一来在碰撞和其他约束中的旋转就不会影响到物体，请看 btRigidBody::setAngularFactor。其他选项，包括设置方向为上的轴的反惯性张量为 0，或者只用铰链约束，Bullet 都不推荐使用。

btKinematicCharacterController 是专注于角色控制的类，它用 btGhostShape 来创建能爬梯子和在墙壁上流畅滑动的角色。详情请看 src/BulletDynamics/Character 和 Demos/CharacterDemo。

9. 软体动力学

9.1. 简介

软体动力学有绳子、布的仿真还有测定体积的软体。软体、软体和碰撞物体之间有两方面的联系：

1） btSoftBody 派生于 btCollisionObject 是主要的软体物体。和刚体不一样的是，软体没有单独的世界转换，所有的节点和向量都是定义在世界坐标系上。

2） btSoftRigidDynamicsWorld 是软体、刚体和碰撞物体三者的共同容器。

读者最好从 Demos/SoftBodyDemo 中学习如何使用软件虚拟。

下面给读者一些基本的指导：

9.2. 从三角网格创建软体

btSoftBodyHelpers::CreateFromTriMesh 能自动从三角网格中创建软体对象。

9.3. 碰撞集群

软体默认使用向量（节点）和三角形（面）来检测碰撞，这需要密集镶嵌（dense tessellation），否则碰撞可能丢失。有个更好的方法是使用自动分解成凸变形集群，为了激活碰撞集群，请按下面步骤进行：

```
psb->generateClusters(numSubdivisions);//enable cluster collision between soft body and rigid body
psb->m_cfg.collisions += btSoftBody::fCollision::CL_RS;//enable cluster collision between soft body and soft body
```

Softbody 和 AllBulletDemos 示例里有个可以查看凸变形集群的调试选项

9.4. 给软体添加作用力

Bullet 有给整个软体或者给软体的节点添加作用力的方法

```
softbody ->addForce(const btVector3& forceVector);
softbody ->addForce(const btVector3& forceVector,int node);
```

9.5. 软体约束

Bullet 提供了固定一个或者多个节点的方法：

```
softbody->setMass(node,0.f);
```

Bullet 有将软体一个或者多个节点粘附到刚体的方法：

```
softbody->appendAnchor(int node,btRigidBody* rigidbody,
                       bool disableCollisionBetweenLinkedBodies=false);
```

还可以在两个软体间添加约束，让它们接触，详情请看 Bullet/Demos/SoftBody

10. Bullet 示例介绍

10.1. AllBulletDemos

10.2. CCD Physics Demo

10.3. COLLADA Physics Viewer Demo

10.4. BSP Demo

10.5. Vehicle Demo

10.6. Fork Lift Demo

11. 高级低层次的技术示例

11.1. Collision Interfacing Demo

11.2. Collision Demo

11.3. User Collision Algorithm