术语表

下面的属于表也是来自bullet的官方wiki。物理引擎用上爱因斯坦的可能会刺激很多~时空穿梭模型、质能转换模型....大都用牛顿。物理引擎做的星际游戏

基础项

**单量 - Scalar**

单量其实就是数字，一个线性的值。比如1.0，1.23.单量可以用来构成其他的项。

比如，两点之间的距离就是一个单量，速度也是一个单量。

**向量，3维向量，4维向量 - Vector**

向量就是有方向的一段距离。

根据空间的维度，向量可分为2维，3维，4维。

三维向量的长度根据勾股定理可以用http://img.blog.csdn.net/20130803204258468来计算，对应的就是向量的lengh方法，求根的算法非常消耗复杂，所以向量类有length2()，得到的就是没有开根的结果。

例如：

**[sql]** [view plain](http://blog.csdn.net/silangquan/article/details/9730963) [copy](http://blog.csdn.net/silangquan/article/details/9730963)

1. #define THRESHOLD 20
2. #define THRESHOLD\_SQ THRESHOLD\*THRESHOLD // Your compiler will optimise this **out** later
4. Vector3 a(12,14,20);
5. if(THRESHOLD > a.length()) { do\_something; } // This one **is** much slower
6. if(THRESHOLD\_SQ > a.length2()) {do\_something; } // Use this version if you can

**欧拉坐标 - Euler co-ordinates**

就是三维坐标，vector3(x,y,z)表示的就是三维空间中的一个点。

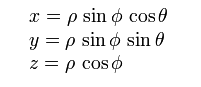
**极坐标 - Polar co-ordinates**

在 平面内取一个定点O， 叫极点，引一条射线Ox，叫做极轴，再选定一个长度单位和角度的正方向（通常取逆时针方向）。对于平面内任何一点M，用ρ表示线段OM的长度，θ表示从Ox到OM的角度，ρ叫做点M的极径，θ叫做点M的极角，有序数对 (ρ,θ)就叫点M的极坐标，这样建立的坐标系叫做极坐标系。

平面中圆的极坐标表达式是：

x  = ρ cos(θ) ;  y = ρ sin(θ)

空间球体的极坐标表达式是：



**欧拉角 - Euler Angle**

欧拉角主要用于描述一个旋转，对空间中的一个点进行旋转通常可以描述为：先绕 x 旋转一个角，再绕 y 轴旋转一个角，再绕 z 轴旋转一个角，最后得到最终坐标。

由于电脑是一只只会线性计算的怪兽，嗷唔～ 在使用欧拉角进行旋转的时候可能会出现一种很蛋疼的情况 - 万圣节锁 [youtube video explaining gimbal lock](http://www.youtube.com/watch?v=rrUCBOlJdt4" \o "http://www.youtube.com/watch?v=rrUCBOlJdt4" \t "_blank)

为了避免万圣节锁，一个方法是设定好旋转轴的旋转顺序。

另一中方法是使用四元组。

**四元组 - Quaternion**

四元数一般定义如下：  
q=w+xi+yj+zk

其中v=(x,y,z)是矢量，w是标量，虽然v是矢量，但不能简单的理解为3D空间的矢量，它是4维空间中的的矢量，也是非常不容易想像的。  
通俗的讲，一个四元数（Quaternion）描述了一个旋转轴和一个旋转角度。这个旋转轴和这个角度可以通过 Quaternion::ToAngleAxis转换得到。当然也可以随意指定一个角度一个旋转轴来构造一个Quaternion。这个角度是相对于单位四元数而言的，也可以说是相对于物体的初始方向而言的。  
当用一个四元数乘以一个向量时，实际上就是让该向量围绕着这个四元数所描述的旋转轴，转动这个四元数所描述的角度而得到的向量。

**轴角 - AxisAngle**

也是用来描述一个旋转。

旋转的轴角表示用两个值参数化了旋转: 一个轴或向量，和描述绕这个轴的旋转量的一个角。它也叫做旋转的指数坐标。  
有时也叫做旋转向量表示，因为这两个参数(轴和角)可用在这个轴上的其模是旋转角的一个向量来表示。

合成项

**变换 - Transform**

变换就是一个位移加上一个旋转。

**矩阵 - Matrix**

所有的变换都可以用矩阵的运算来实现。但是要注意渲染系统的矩阵算法的实现和bullet中的是否是一样的。

物理项

**刚体 - Rigid Body**

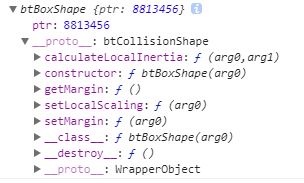
刚体有着固定的质量、尺寸和其他的物理属性，它时物理模拟中最基础的项。

**节点约束 - Joint，constraint**

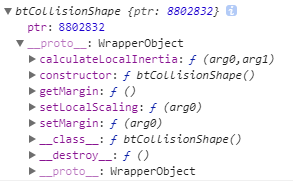
节点约束就是用于连接两个物体的地方，连接的方式又很多中，一种是简单的连接，连接的地方可以旋转，还有一种是绞接，只能单向运动。

**世界 - World**

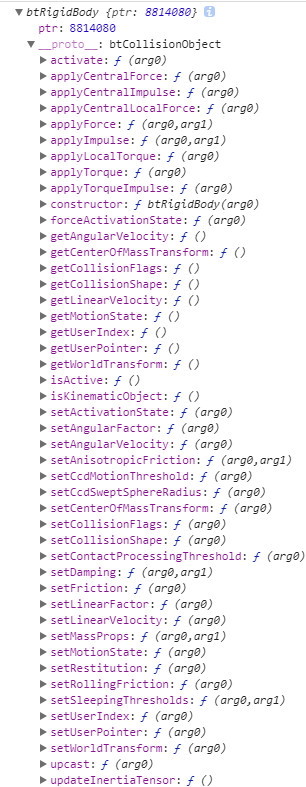
所有的物体，所有的刚体，软体都在世界中，整个模拟系统就是一个世界。



**threeObject.userData.physicsBody.getCollisionShape()**









力，循序渐进——ApplyForce, ApplyForce()是给刚体添加一个创造力，让刚体做加速或减速运动

顾名思义，ApplyForce方法会在刚体上施加一个力。学过物理力学的同学都知道，F=ma，有了力F就有了加速度a，有了加速度，物体就会有速度，就会慢慢动起来。(但是不会立马动起来，因为力不会直接影响速度)。

举个简单的例子，小明推一个静止的箱子，箱子不会立马飞出去，而是慢慢的、越来越快的动起来(减速也一样)。

center Of Mass 是质心的位置，默认为物体的位置

applyImpulse(force) 设置物体的冲力, 创造冲量

setLinearVelocity 设置线速度

setAngularVelocity 设置角速度

setRestitution(1.0f) 设置物体的恢复力 , 设置弹力

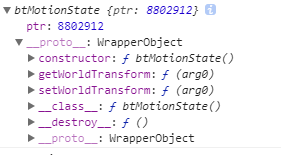
恢复力和摩擦力要成对使用，即使你把恢复力设置为1.0而不设置摩擦力为0，依旧不是完全弹性碰撞，自己可以修改调试就可以知道正确与否。关于弹性碰撞可以参考一些物理书。对于恢复力，在Chipmunk中文手册有“0.0表示没有弹性，1.0b表示“富有”弹性。然而由于存在模拟误差，不推荐使用1.0或更高的值，碰撞的弹性是由单个形状的弹性相乘得到”。对于摩擦力，在Chipmunk中文手册有“Chipmunk使用的是库仑摩擦力模型，0.0值表示无摩擦。碰撞间的摩擦是由单个形状的摩擦相乘找到”

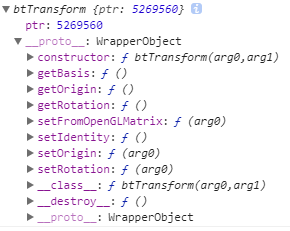
setFriction(0.0f) 设置物体的摩擦力 \_material.friction = friction;

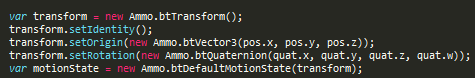
setDensity(1.0f) 设置物体密度

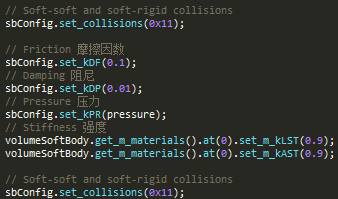
setLinearVelocity 与ApplyImpulse一样，直接影响刚体的速度。不一样的是，setLinearVelocity添加的速度会覆盖刚体原有的速度。

**threeObject.userData.physicsBody.getMotionState()**

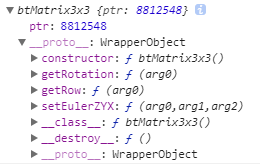




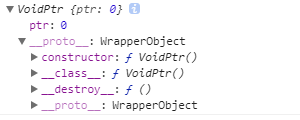




**body.getWorldTransform().getBasis()**

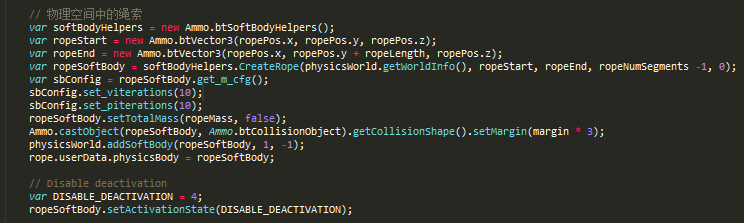


**threeObject.userData.physicsBody.getUserPointer()**



**threeObject.userData.physicsBody.getUserIndex() = 0**





**怎么得到mass的值**

**怎么修改mass的值**

[**http://studiofreya.com/game-maker/bullet-physics/bullet-physics-how-to-change-body-mass/**](http://studiofreya.com/game-maker/bullet-physics/bullet-physics-how-to-change-body-mass/)

**怎么初始化物理世界？**

**Rigid刚体**

**function initPhysics() {**

**// bullet基本场景配置**

**collisionConfiguration = new Ammo.btDefaultCollisionConfiguration();//设置碰撞配置和调度程序**

**dispatcher = new Ammo.btCollisionDispatcher(collisionConfiguration);**

**broadphase = new Ammo.btDbvtBroadphase();//构建broadphase**

**solver = new Ammo.btSequentialImpulseConstraintSolver();//objects 正确交互，考虑到重力、游戏逻辑、碰撞、铰链约束，The actual physics solver**

**physicsWorld = new Ammo.btDiscreteDynamicsWorld(dispatcher, broadphase, solver, collisionConfiguration);//建立动态物理世界**

**physicsWorld.setGravity(new Ammo.btVector3(0, gravityConstant, 0));//设置重力在y轴方向上**

**//程序执行完 js会自动删除这些占用的内存**

**}**

**Soft软体**

**function initPhysics() {**

**// bullet基本场景配置**

**// collisionConfiguration = new Ammo.btDefaultCollisionConfiguration();**

**/\*\***

**\* 启动支持肉体的物理引擎**

**\* config，solver和physicsWorld要用softbody版本**

**\* 还要单独创建softBodySolver**

**\*/**

**collisionConfiguration = new Ammo.btSoftBodyRigidBodyCollisionConfiguration();**

**dispatcher = new Ammo.btCollisionDispatcher(collisionConfiguration);**

**broadphase = new Ammo.btDbvtBroadphase();**

**solver = new Ammo.btSequentialImpulseConstraintSolver();**

**softBodySolver = new Ammo.btDefaultSoftBodySolver();**

**physicsWorld = new Ammo.btSoftRigidDynamicsWorld(dispatcher, broadphase, solver, collisionConfiguration, softBodySolver);**

**physicsWorld.setGravity(new Ammo.btVector3(0, gravityConstant, 0));**

**physicsWorld.getWorldInfo().set\_m\_gravity(new Ammo.btVector3(0, gravityConstant, 0));**

**}**



