Bullet物理引擎不完全指南(Bullet Physics Engine not complete Guide)

write by 九天雁翎(JTianLing) -- blog.csdn.net/vagrxie

讨论新闻组及文件

前言

Bullet据称为游戏世界占有率为第三的物理引擎，也是前几大引擎目前唯一能够找到的支持iPhone，开源，免费(Zlib协议，非常自由，且商业免费）的物理引擎，但是文档资料并不是很好，Demo虽然多，但是主要出于特性测试/展示的目的，会让初学者无从看起，一头雾水。我刚学习Bullet的时候困于没有好的文档及资料，非常没有头绪，折腾了很久，所以就发挥没有就创造的精神，写作及整理此文，（以整理资料为主，自己写 为辅）希望大家在学习Bullet的时候不要再像我开始一样没有头绪。因为我实在没有精力去完成一个包含Bullet方方面面的完全指南，所以本文只能是不完全版本，这个就请大家谅解了，但是期望能够真正的完成一个简单的由浅入深的教程，并提供尽量详尽的额外信息链接，只能说让初学者比看官方的WIKI和Demo效果更好，大家有好的信息和资料而本文没有包含的，也请告诉我，我可以在新版中添加进来。因为我学习Bullet的时间也比较短，有不对的地方请高人指点。

前段时间简单的学习了一下Bullet,牵涉到图形部分的时候主要都是研究Bullet与Ogre的结合，所以使用了OgreBullet这个Ogre的Addon，其实真正的学习当然还是直接利用Bullet本身附带的简单的debug OpenGL绘制就好了。本文就完全以Bullet本身的Debug功能来学习，虽然简陋，但是可以排除干扰，专注于bullet。也许除了本文，会有个额外的文章，稍微研究下Ogre与Bullet的整合和分析一下OgreBullet的源码。

Bullet介绍

Bullet的主页。最新版本在这里下载。简单的中文介绍见百度百科。一些也许可以促使你选择Bullet的小故事在以前的文章中有提及，参考这里的开头--为什么选择Bullet。很遗憾的是前几天看到的一篇很详细的bullet中文介绍找不到了，将来也许

补上。

安装

Bullet作为一款开源物理引擎，你可以选择作者编译好的SDK，或者直接从源码编译自己的版本（Windows版本自带VS工程）。得益于CMake，在其他平台从源码自己编译也非常简单，参考这里。iPhone版本的话参考这里。想要更详细点的图文教程可以参考Creating_a_project_from_scratch。

Hello World Application

在学习之前，没有接触过物理引擎的可以参考一下这个术语表。

这里有个较为详细的教程。也包含在Bullet本身的一个名叫 AppHelloWorld 的Demo中。（注释也很详细，但是和WIKI上的版本略有不同）可以大概的对Bullet有个感觉。

其实Bullet与Ogre走的一条路线，为了灵活，增加了很多使用的复杂性。（真怀念Box2D和Irrlicht的简单啊）其实即使希望通过strategy模式来增加灵活度，让用户可以自由的选择各类算法和解决方案，但是我还是感觉首先提供默认解决方案，用户需要不同方案的时候通过Set方式改变(甚至也可以new的时候修改）但是大牛们研究这些东西那么透，总是会觉得这个世界上不存在默认方案。。。。。因为没有方案是最优的，是适合大多数情况的，所以导致Bullet的HelloWorld程序源代码都已经超过100行。。。。。。。。。。-_-!发了点牢骚。。。。。

通过HelloWorld程序，我们大概可以知道一些东西，比如建立一个Bullet物理世界的步骤，比如Bullet的类以bt（变态-_-!)开头，比如Bullet与Box2D这样的2D物理引擎一样，专注于数据的计算，本身没有图形输出，比如创建一个物理实体的时候也有shape的概念，然后通过一个结构作为参数(BodyConstructionInfo）来创建真实的物体，大概的熟悉一下就好，具体的细节还不懂，没有关系，一步一步来。

另外，建议趁这个机会，确定自己机器使用Bullet的环境，特别是Win32下，我的使用方法是，利用BULLET_HOME环境变量指明Bullet安装的位置，BULLTE_LIBS指明最后编译完的静态库的位置，工程中利用这两个环境变量来确定位置。（这种用法很适合屏蔽各机器的环境不同）最后的Hello World工程见https://bullet-sample.jtianling.googlecode.com/hg/中的Bullet-HelloWorld。

请确保该Hello World程序能够运行（无论是你自己的还是用我的）然后才继续下面的内容。

让你坐在司机的位置上

该怎么学习的问题，向来都是各执一词，有人认为该从最基础的学起，就像建房子一样打好地基，有人会更加推崇自上而下的学习(Top-Down Approach），我属于后一派，能先写有用的可以摸到的程序，然后一层一层的向下学习，这样会更加有趣味性，并且学习曲线也会更加平缓，假如你是前一派，那么推荐你先看完Bullet的User Manual，然后是Bullet所有的Tutorial Articles，然后再自己一个一个看Demo。

在Hello World的例子中你已经可以看到文本数据的输出，能够看到球/Box的落下了，但是很明显太不直观了，得益于Bullet良好的debug输出支持，我们要先能直观的通过图形看到球的落下！先坐在司机的位置上才能学会开车^^你也不至于被乏味的汽车/交通理论闷死。

Bullet像Ogre一样，提供了一个DemoApplication类，方便我们学习，我们先看看Bullet的DemoApplication是怎么样的。先看看Bullet自己提供的AppBasicDemo这个Demo。忽略那些作者用#if 0关闭的内容和hashmap的测试内容，看看DemoApplication的用法。

首先是BasicDemo类，从class BasicDemo : public PlatformDemoApplication可以看到，DemoApplication是给你继承使用的，这里的PlatformDemoApplication实际是GlutDemoApplication。（Win32那个作者好像只是预留的）

怎么去实现这个类先放一边，看看整个类的使用：

```
GLDebugDrawer    gDebugDrawer;
BasicDemo ccdDemo;
ccdDemo.initPhysics();
ccdDemo.getDynamicsWorld()->setDebugDrawer(&gDebugDrawer);
glutmain(argc, argv,640,480,"Bullet Physics Demo. http://bulletphysics.com",&ccdDemo);
```

实际就这5句，很简单，构造debug,BasicDemo，调用initPhysics函数，设定debug，调用glutmain这个函数，参数也一目了然。这里就不看了。看实现一个有用的DemoApplication的过程。

大概看看DemoApplication这个基类和GlutDemoApplication知道必须要实现的两个纯虚函数是

```
virtual    void initPhysics() = 0;
virtual void clientMoveAndDisplay() = 0;
```
看BasicDemo的实现后，知道还需要实现displayCallback这个现实回调，基本上就没有其他东西了，理解起来也还算容易。

initPhysics的部分，一看就知道，与HelloWorld中过程几乎一致，也就是实际构建物理世界的过程。只是多了
```
setTexturing(true);
setShadows(true);
setCameraDistance(btScalar(SCALING*50.));
```
这三个与显示有关的东西（其实这些代码放到myinit中去也可以，毕竟与物理无关）
最后还多了个clientResetScene的调用，我们知道这个过程就好，具体函数的实现先不管。

clientMoveAndDisplay和displayCallback部分
其实非常简单，几乎可以直接放到glutExampleApplication中去。（事实上不从灵活性考虑，我觉得放到glutExampleApplication中更好）
原来的程序有些代码重复，其实只要下列代码就够了：（一般的程序也不需要修改）
```
void BasicDemo::clientMoveAndDisplay()
{
    //simple dynamics world doesn't handle fixed-time-stepping
    float ms = getDeltaTimeMicroseconds();

    ///step the simulation
    if (m_dynamicsWorld)
    {
        m_dynamicsWorld->stepSimulation(ms / 1000000.f);
    }

    displayCallback();
}

void BasicDemo::displayCallback(void) {

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
    renderme();

    //optional but useful: debug drawing to detect problems
    if (m_dynamicsWorld)
        m_dynamicsWorld->debugDrawWorld();

    glFlush();
    swapBuffers();
}
```

运行该程序能够看到中间一个很多Box堆起来的大方块，点击鼠标右键还能发射一个方块出去。
了解这个Demo以后，我们就可以直接来用Bullet构建我们自己的物理世界了，暂时不用考虑图形的问题，甚至不用知道Bullet使用GLUT作为debug图形的输出，GLUI做界面，都不用知道，只需要知道上面demoApplication的使用和在initPhysics函数中完成构建物理世界的代码。另外，你愿意的话，也可以先看看exitPhysics的内容，用于分配资源的释放，作为C++程序，一开始就关注资源的释放问题是个好习惯。虽然对于我们这样简单的demo程序来说是无所谓的。

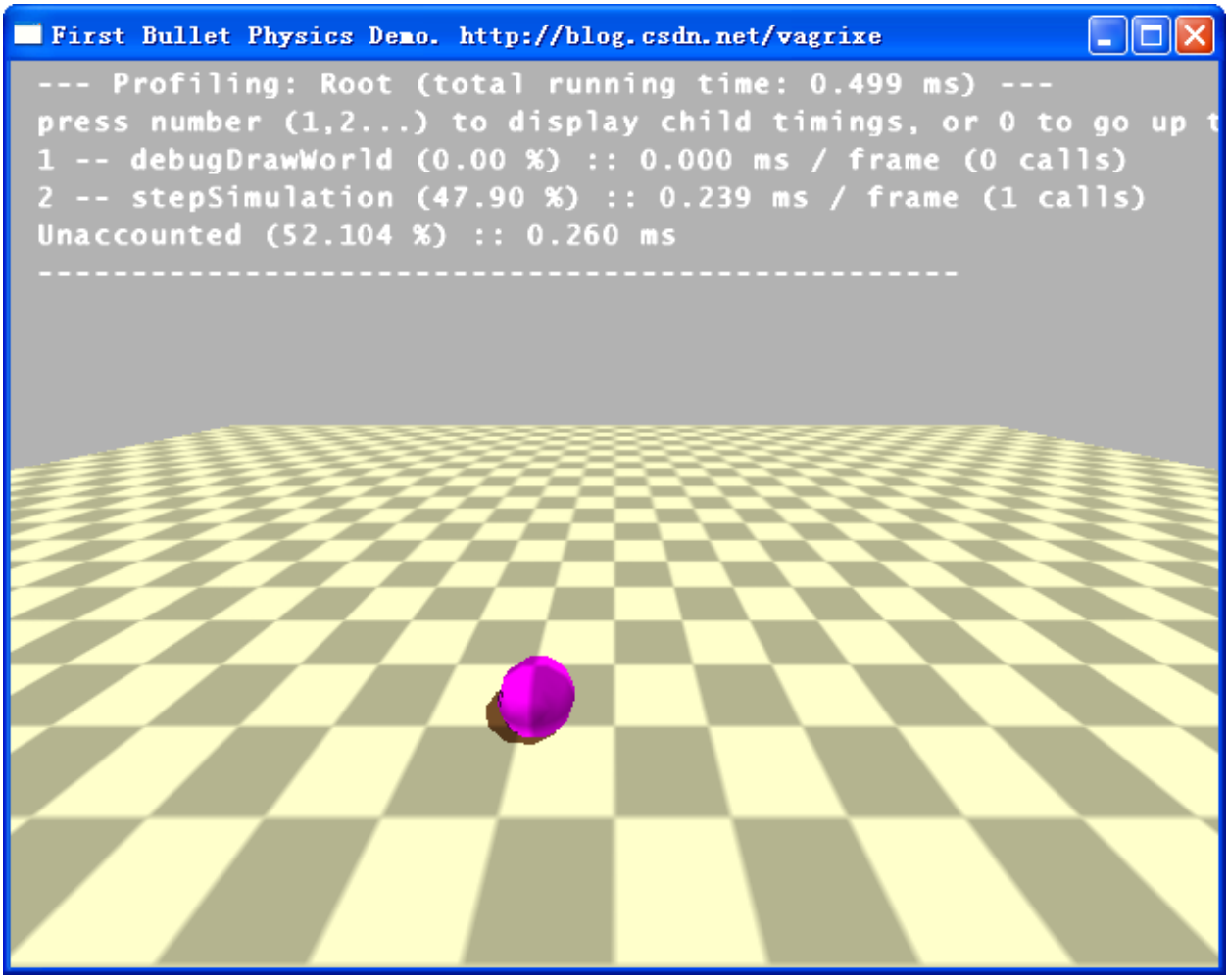
看过上面Demo后，也许你已经有些了解，也许你还是一头雾水，不管怎么样，Bullet的Demo毕竟还是别人的东西，现在，从零开始，构建一个HelloWorld程序描述的世界。先自己尝试一下！假如你成功了，那么直接跳过一下的内容，失败了，再回头了看看，提醒你步骤：
1.继承DemoApplication，拷贝上面clientMoveAndDisplay和displayCallback部分的代码，实现这两个函数。
2.在initPhysics函数中完成构建物理世界的代码。（构建过程参考HelloWorld）
3.Main中的使用代码：

```
    GLDebugDrawer    gDebugDrawer;
    BasicDemo ccdDemo;
    ccdDemo.initPhysics();
    ccdDemo.getDynamicsWorld()->setDebugDrawer(&gDebugDrawer);
    glutmain(argc, argv,640,480,"Bullet Physics Demo.
http://bulletphysics.com",&ccdDemo);
```

4.注意工程需要多包含$(BULLET_HOME)/Demos/OpenGL的头文件目录

和库：
$(BULLET_HOME)/Glut/glut32.lib
opengl32.lib
glu32.lib
麻烦点的是glut是个动态库，你需要将dll文件拷贝到你工程的运行目录。

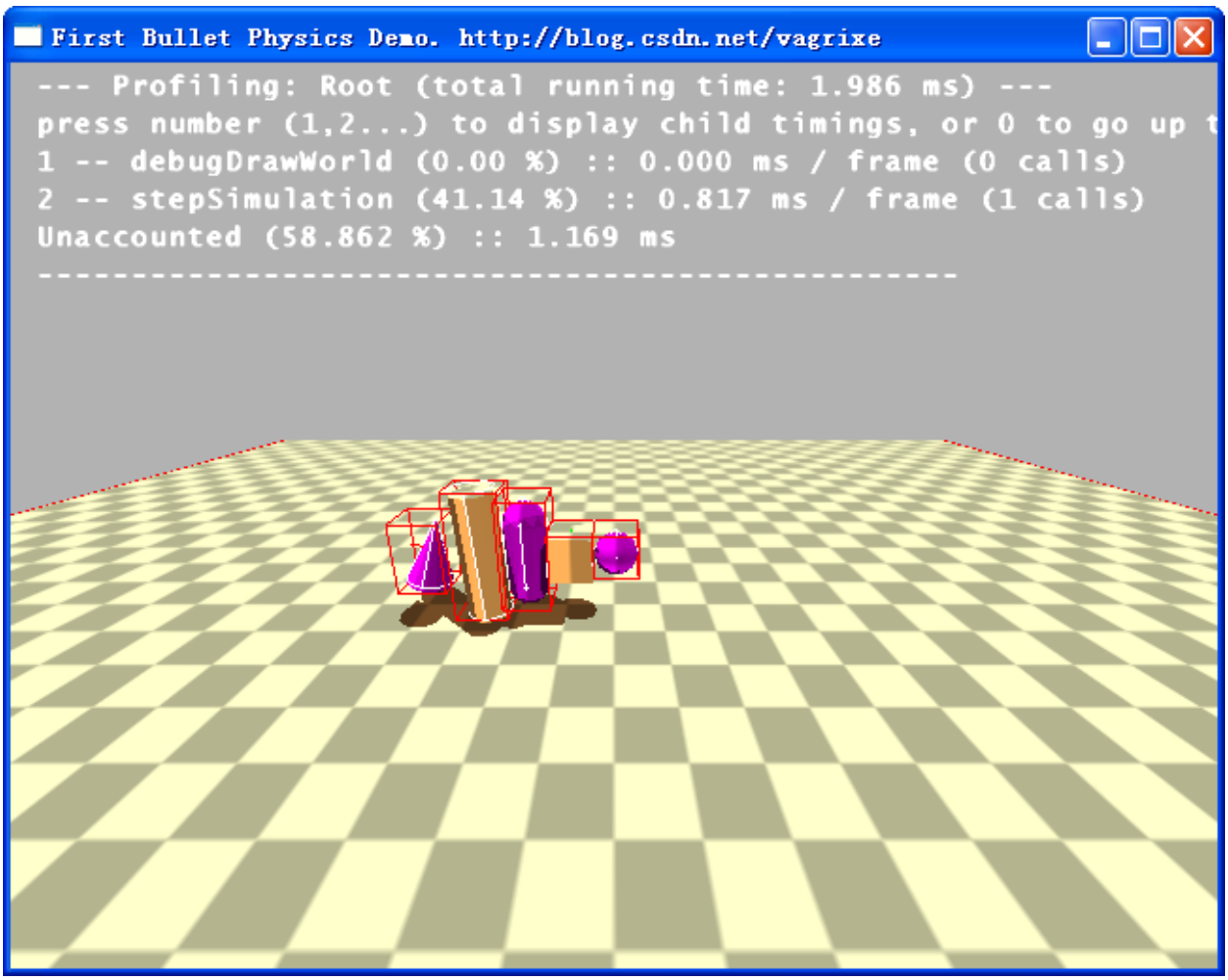
现在应该成功了吧？
我实现的工程见https://bullet-sample.jtianling.googlecode.com/hg/中的Bullet-WithGL。
于是乎，现在你已经可以看到Hello World中那个不曾显示的电脑凭空现象的球了。大概是下面这个样子滴：(因为现在Google Docs写完文章后很久就不能够直接post到CSDN的博客中去了，所以每次写完文章后都得到新建文章中去复制粘贴，图片还需要重新上传然后插入，非常麻烦，所以最近的文章都尽量的减少了图片的使用，见谅。其实说来，只有看热闹的人才需要截图，真的看教程的人估计自己的程序都已经运行起来了，也没有必要看我的截图了)



到目前为止，你已经有了可以自己happy一下的程序了，你可以立即深入的学习，去研究ExampleApplication的源代码，去了解Bullet是怎么与图形交互的，但是在这个之前，特别是对于以前没有使用过其他物理引擎的人，先多在这个图形版的HelloWorld的程序的基础上玩玩，比如现在球很明显没有弹性，调整一下反弹的系数看看，比如多生成几个球，比如加上速度，演示两个球的碰撞，比如因为现在没有设置debugmode，所以实际没有debug信息输出，尝试输出aabb等debug信息，有助于进一步学习。有了图形，就有了丰富的世界，先对Bullet的各个概念先熟悉一下（特别是btRigidBodyConstructionInfo中的各个变量，还有各个shape）然后再前进吧。事实上，得益于ExampleApplication，现在甚至可以用鼠标左键拖拽物理，右键发射箱子的功能也还在，还能按左右键调整camera。（其实还有一堆功能，自己尝试一下吧）因为比较简单，在本教程中不会再有关于这些基础信息的内容，只能自己去找资料或者尝试了。其实就我使用物理引擎的经验，学习并使用一款物理引擎并不会太难，最麻烦的地方在于后期游戏中各个参数的调整，尽量使游戏的效果变得真实，自然，或者达到你

想要的效果，这种调整很多时候需要靠你自己的感觉，而这个感觉的建立，那就是多多尝试一个物理引擎中的在各个参数下呈现的不同效果了，这种感觉的建立，经验的获得，不是任何教程，文档或者演示程序能够给你的。

比如，下面是Bullet支持的5种基本物体形状：



其实上面的内容是最关键的，此时学开车的你已经在司机的位置了，学游泳的你已经在水里了，剩下的Bullet相关的内容虽然还有很多，但是其实已经完全可以自己独立折腾了，因为每个折腾的成果你都已经能够实时的看到，能够很哈皮的去折腾了。

与显示的整合，**MotionState**

一个只有数据运算的物理引擎，一般而言只能为显示引擎提供数据，这就牵涉到与图形引擎整合的问题，像Box2D这样的物理引擎就是直接需要直接向各个物理实体去查询位置，然后更新显示，这种方式虽然简单，但是我感觉非常不好，因为难免在update中去更新这种东西，导致游戏逻辑部分需要处理物理引擎+图形引擎两部分的内容。（可以参考Box2D与Cocos2D for iPhone的整合）而且，对于完全没有移动的物体也会进行一次查询和移动操作。（即使优化，对不移动物体也是进行了两次查询）

Bullet为了解决此问题，提供了新的解决方案，MotionState。其实就是当活动物体状态改变时提供一种回调，而且就Bullet的文档中说明，此种回调还带有适当的插值以优化显示。通过这种方法，在MotionState部分就已经可以完成显示的更新，不用再需要在update中添加这种更新的代码。而且，注意，仅仅对活动物体状态改变时才会进行回调，这样完全避免了不活动物体的性能损失。

首先看看ExampleApplication中是怎么利用default的MotionState来显示上面的图形的，然后再看看复杂点的例子，与Ogre的整合。
先看看回调接口：

```
///The btMotionState interface class allows the dynamics world
to synchronize and interpolate the updated world transforms with
graphics
///For optimizations, potentially only moving objects get
synchronized (using setWorldPosition/setWorldOrientation)
class   btMotionState
{
    public:

        virtual ~btMotionState()
        {

        }

        virtual void  getWorldTransform(btTransform& worldTrans
) const =0;

        //Bullet only calls the update of worldtransform for
```

```
active objects
        virtual void  setWorldTransform(const btTransform&
worldTrans)=0;


};
```

很简单，一个get接口，用于bullet获取物体的初始状态，一个set接口，用于活动物体位置改变时调用以设置新的状态。
下面看看btDefaultMotionState这个bullet中带的默认的MotionState类。

```
///The btDefaultMotionState provides a common implementation to
synchronize world transforms with offsets.
struct  btDefaultMotionState : public btMotionState
{
    btTransform m_graphicsWorldTrans;
    btTransform m_centerOfMassOffset;
    btTransform m_startWorldTrans;
    void*       m_userPointer;

    btDefaultMotionState(const btTransform& startTrans =
btTransform::getIdentity(),const btTransform& centerOfMassOffset
= btTransform::getIdentity())
        : m_graphicsWorldTrans(startTrans),
        m_centerOfMassOffset(centerOfMassOffset),
        m_startWorldTrans(startTrans),
        m_userPointer(0)

    {
    }

    ///synchronizes world transform from user to physics
    virtual void   getWorldTransform(btTransform&
centerOfMassWorldTrans ) const
    {
```

```
            centerOfMassWorldTrans =
m_centerOfMassOffset.inverse() * m_graphicsWorldTrans ;
    }

    ///synchronizes world transform from physics to user
    ///Bullet only calls the update of worldtransform for active
objects
    virtual void   setWorldTransform(const btTransform&
centerOfMassWorldTrans)
    {
            m_graphicsWorldTrans = centerOfMassWorldTrans *
m_centerOfMassOffset ;
    }
};
```

这个默认的MotionState实现了这两个接口，但是还引入了质心（center Of Mass 应该是指质心吧）的概念，与外部交互时，以质心位置表示实际物体所在位置。
在一般rigitBody的构造函数中可以看到下列代码：

```
    if (m_optionalMotionState)
    {
        m_optionalMotionState-
>getWorldTransform(m_worldTransform);
    } else
    {
        m_worldTransform =
constructionInfo.m_startWorldTransform;
    }
```

这就是get函数的使用，也就是决定物体初始坐标的函数回调。

set函数的回调如下：

```
void    btDiscreteDynamicsWorld::synchronizeSingleMotionState(btRigidBody*
body)
{
```

```
    btAssert(body);

    if (body->getMotionState() && !body-
>isStaticOrKinematicObject())
    {
        //we need to call the update at least once, even for
sleeping objects
        //otherwise the 'graphics' transform never updates
properly
        ///@todo: add 'dirty' flag
        //if (body->getActivationState() != ISLAND_SLEEPING)
        {
            btTransform interpolatedTransform;
            btTransformUtil::integrateTransform(body-
>getInterpolationWorldTransform(),
                body->getInterpolationLinearVelocity(),body-
>getInterpolationAngularVelocity(),m_localTime*body-
>getHitFraction(),interpolatedTransform);
            body->getMotionState()-
>setWorldTransform(interpolatedTransform);
        }
    }
}
```

也就是同步状态的时候调用。此过程发生在调用bullet的btDynamicsWorld::stepSimulation函数调用时。

然后可以参考DemoApplication的DemoApplication::renderscene(int pass)函数：

```
    btScalar    m[16];
    btMatrix3x3 rot;rot.setIdentity();
    const int  numObjects=m_dynamicsWorld-
>getNumCollisionObjects();
    btVector3 wireColor(1,0,0);
```

```
    for(int i=0;i<numObjects;i++)
    {
        btCollisionObject*  colObj=m_dynamicsWorld->getCollisionObjectArray()[i];
        btRigidBody*        body=btRigidBody::upcast(colObj);
        if(body&&body->getMotionState())
        {
            btDefaultMotionState* myMotionState = (btDefaultMotionState*)body->getMotionState();
            myMotionState->m_graphicsWorldTrans.getOpenGLMatrix(m);
            rot=myMotionState->m_graphicsWorldTrans.getBasis();
        }
    }
}
```

实际也就是再通过获取motionState然后获取到图形的位置了，这种defaultMotion的使用就类似Box2D中的使用了。

既然是回调，那么就可以让函数不仅仅做赋值那么简单的事情，回头来再做一次轮询全部物体的查询，官网的WIKI中为Ogre编写的MotionState就比较合乎推荐的MotionState用法，代码如下：

```
lass 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;
};
```

注意，这里的使用直接在set回调中直接设置了物体的位置。如此使用MotionState后，update只需要关心逻辑即可，不用再去手动查询物体的位置，然后更新物体的位置并刷新显示。

碰撞检测

物理引擎不仅仅包括模拟真实物理实现的一些运动，碰撞，应该还提供方式供检测碰撞情况，bullet也不例外。

AppCollisionInterfaceDemo展示了怎么直接通过btCollisionWorld来检测碰撞而不模拟物理。

而官方的WIKI对于碰撞检测的描述也过于简单，只给下列的示例代码，但是却没有详细的解释。

```
    //Assume world->stepSimulation or world->performDiscreteCollisionDetection has been called

    int numManifolds = world->getDispatcher()->getNumManifolds();
    for (int i=0;i<numManifolds;i++)
    {
        btPersistentManifold* contactManifold =  world->getDispatcher()->getManifoldByIndexInternal(i);
        btCollisionObject* obA = static_cast<btCollisionObject*>(contactManifold->getBody0());
        btCollisionObject* obB = static_cast<btCollisionObject*>(contactManifold->getBody1());

        int numContacts = contactManifold->getNumContacts();
        for (int j=0;j<numContacts;j++)
        {
            btManifoldPoint& pt = contactManifold->getContactPoint(j);
            if (pt.getDistance()<0.f)
            {
                const btVector3& ptA = pt.getPositionWorldOnA();
                const btVector3& ptB = pt.getPositionWorldOnB();
                const btVector3& normalOnB = pt.m_normalWorldOnB;
            }
        }
    }
```

以上代码的主要内容就是

```
int numManifolds = world->getDispatcher()->getNumManifolds();
btPersistentManifold* contactManifold =  world->getDispatcher()-
```

>getManifoldByIndexInternal(i);
两句。

而btPersistentManifold类表示一个Manifold，其中包含了body0,body1表示Manifold的两个物体。

这里特别提及的是,Manifold并不直接表示碰撞，其真实的含义大概是重叠，在不同的情况下可能表示不同的含义，比如在Box2D中，手册的描述大概是（凭记忆）为了快速的检测碰撞，在2D中一般先经过AABB盒的检测过滤，而只有AABB盒重叠的才有可能碰撞，而Manifold在Box2D中就表示AABB盒重叠的两个物体，而我看Bullet有不同的Broadphase,在实际中，也重叠也应该会有不同的情况，因为我没有看源码，所以不能确定，但是，总而言之，可以理解Manifold为接近碰撞的情况。

所以无论在Box2D还是Bullet中，都有额外的表示碰撞的概念，那就是contact（接触）。上述示例代码：

```
int numContacts = contactManifold->getNumContacts();
```

就表示查看接触点的数量，假如接触点为0，那么自然表示两个物体接近于碰撞，而实际没有碰撞。而上述代码中的Distance的判断应该是防止误差，因为我输出了一个盒子和地面发生碰撞的全部过程的distance，发现绝大部分情况，只要有contact，那么距离就小于0，可是在一次盒子离开地面的过程中，distance还真有过一次0.00x的正值。。。。。。。

当你开始放心大胆的使用上述代码后，也许你总是用来模拟物体的其他效果，也许都不会有问题，直到某一天你希望在碰撞检测后删除掉发生碰撞的问题，你的程序crash了。。。。你却不知道为什么。用前面的demo来展示碰撞检测的方法，并且删除掉发生碰撞的物体。一般先写出的代码都会类似下面这样：

```
    int numManifolds = m_dynamicsWorld->getDispatcher()->getNumManifolds();
    for (int i=0;i<numManifolds;i++)
    {
        btPersistentManifold* contactManifold = m_dynamicsWorld->getDispatcher()->getManifoldByIndexInternal(i);
        btCollisionObject* obA = static_cast<btCollisionObject*>(contactManifold->getBody0());
        btCollisionObject* obB = static_cast<btCollisionObject*>(contactManifold->getBody1());

        int numContacts = contactManifold->getNumContacts();
        for (int j=0;j<numContacts;j++)
```

```
        {
            btManifoldPoint& pt = contactManifold-
>getContactPoint(j);
            if (pt.getDistance()<0.f)
            {
                RemoveObject(obA);
                RemoveObject(obB);
            }
        }
    }
```

但是上面这样的代码是有问题的，这在Box2D的文档中有详细描述，Bullet文档中没有描述，那就是obA和obB可能重复删除的问题（也就相当于删除同一个对象多次，自然crash）在本例中有两个问题会导致重复，很明显的一个，当两个物体多余一个Contact点的时候，在遍历Contacts点时会导致obA,obB重复删除。另外，稍微隐晦点的情况是，当一个物体与两个物体发生碰撞时，同一个物体也可能在不同的manifold中，所以，真正没有问题的代码是先记录所有的碰撞，然后消除重复，再然后删除。这是Bullet文档中没有提到，WIKI中也没有说明的，初学者需要特别注意。。。。。。下面才是安全的代码：

```
    int numManifolds = m_dynamicsWorld->getDispatcher()-
>getNumManifolds();
    for (int i=0;i<numManifolds;i++)
    {
        btPersistentManifold* contactManifold =
m_dynamicsWorld->getDispatcher()->getManifoldByIndexInternal(i);
        btCollisionObject* obA =
static_cast<btCollisionObject*>(contactManifold->getBody0());
        btCollisionObject* obB =
static_cast<btCollisionObject*>(contactManifold->getBody1());

        int numContacts = contactManifold->getNumContacts();
        for (int j=0;j<numContacts;j++)
        {
            btManifoldPoint& pt = contactManifold-
```

```
>getContactPoint(j);
            if (pt.getDistance()<0.f)
            {
                m_collisionObjects.push_back(obA);
                m_collisionObjects.push_back(obB);
            }
        }
    }

    m_collisionObjects.sort();
    m_collisionObjects.unique();
    for (CollisionObjects_t::iterator itr =
m_collisionObjects.begin();
        itr != m_collisionObjects.end();
        ++itr) {
        RemoveObject(*itr);
    }

    m_collisionObjects.clear();
```

上述m_collisionObjects是std::list类型的成员变量。

碰撞过滤

Bullet的wiki 提到了3个方法，这里只讲述最简单的mask（掩码）过滤方法。

mask的使用相信大家基本都接触过，无非就是通过一个整数各个2进制位来表示一些bool值。比如Unix/Linux中文件权限的掩码。在bullet中的碰撞mask的使用非常简单，主要在addRigidBody时候指定。（需要注意的是，只有btDiscreteDynamicsWorld类才有这个函数，btDynamicsWorld并没有，所以demoApplication中的成员变量dynamicWorld不能直接使用。）
WIKI中的代码已经很能说明问题了：

```
#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;


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);
```

特别是那个`#define BIT(x) (1<<(x))`宏用的很有意思。
不要特别注意的是，两个物体要发生碰撞，那么，两个物体的collidesWith参数必须要互相指定对方，假如A指定碰撞B，但是B没有指定碰撞A,那么还是没有碰撞。就上面的例子而言，虽然ship和powerup想要撞墙，但是墙不想撞它们，那么事实上，上面的例子就相当于过滤了所有墙的碰撞，其实仅仅只有ship和power的碰撞，这真所谓强扭的瓜不甜啊，等双方都情愿。

仿照上面的例子，假如你希望在碰撞检测的时候过滤掉地板，只让物体间发生碰撞然后删除物体，为demo添加下列代码：

```
#define BIT(x) (1<<(x))
    enum collisiontypes {
        COL_NOTHING = 0, //<Collide with nothing
        COL_GROUND = BIT(1), //<Collide with ships
        COL_OBJECTS = BIT(2), //<Collide with walls
    };
```

```
    short GroundCollidesWith = COL_OBJECTS;
    short ObjectsCollidesWith = COL_GROUND;
```

但是当你将上述方法应用到demo中，想要过滤掉你想要的碰撞，你会发现碰撞检测的确是过滤掉了，同时过滤掉的还有碰撞，球直接传地板而过，掉进了无底的深渊。注意，这里的过滤是指碰撞过滤，而不是碰撞检测的过滤，假如希望实现碰撞检测的过滤，你可以在碰撞检测中直接进行。比如前面地板的例子，因为地板是静态物体，你可以通过调用rigidBody的isStaticObject来判断是否是地板，然后进行删除，就如下面的代码这样：

```
            if (pt.getDistance()<0.f) {
                if (!obA->isStaticObject()) {
                    m_collisionObjects.push_back(obA);
                }

                if (!obB->isStaticObject()) {
                    m_collisionObjects.push_back(obB);
                }
            }
```

假如希望与地面碰撞并不删除物体，只有物体与物体的碰撞才删除物体，这也简单：

```
                if (!obA->isStaticObject() && !obB-
>isStaticObject()) {
                    m_collisionObjects.push_back(obA);
                    m_collisionObjects.push_back(obB);
```

至于更加复杂的情况，还可以借助于rigidBody的UserPointer，这在WIKI中没有提及，

```
    ///users can point to their objects, userPointer is not used
by Bullet
    void*  getUserPointer() const
    {
        return m_userObjectPointer;
    }
```

```
    ///users can point to their objects, userPointer is not used
by Bullet
    void   setUserPointer(void* userPointer)
    {
        m_userObjectPointer = userPointer;
    }
```

但是就我的经验，这两个函数的作用是巨大的，你可以将你需要的一切都设置进去。。。。。。。。然后取出来，就上面的碰撞检测过滤而言，你完全可以实现自己的一套碰撞检测mask，只要你想，一切皆有可能。这些例子的完整源代码见https://bullet-sample.jtianling.googlecode.com/hg/中的Bullet-CollideDetection工程。

约束(Constraints)和连接(Joints)

一个一个单独的物理实体已经可以构建一个有意思的物理世界了，但是显示世界有很多东西（最典型的就是绳子连接的物体）不是单独的物理实体可以模拟的，物理引擎中使用约束来模拟类似的东西/现象。

（待补充）

软体

因为我的使用暂时不需要用到软体，暂时未学习此部分内容，欢迎大家补充。

有用的工具

1. MAYA,3D Max的插件
2. Blender，开源的3D建模工具，内建的Game Engine有直接的Bullet支持，还有Erwin提供的改版可以直接导出.bullet文件。

使用了**Bullet**的其他有用工程

1. GameKit,Erwin Coumans自己发起的整合Ogre/Irrlicht和Bullet的游戏引擎，与Blender结合的也很好。
2. oolongengine,乌龙引擎，wolfgang.engel（他的博客）这个大牛（到底有多牛可以参考这里）发起的iPhone平台引擎项目，使用了Bullet，也为Bullet能够流畅的运行于iPhone平台做出了很大贡献。（优化了浮点运算）为什么写乌龙引擎？wolfgang自己有个解释，见这里。
3. Dynamica, Erwin建立的工程，开发bullet的Maya,3D Max插件。
4. bullet-physics-editor, Erwin自己发起的一个bullet编辑器工程，目前还处于前期开发阶段。但是项目中同时包含一些能够到处.Bullet文件的Blender改版。

Bullet的实现原理

1.Physics Pipeline Implementation,应该是一个在爱尔兰的中国人写的，并发布在WIKI上，这里是他的博客。
2.SIGGRAPH 2010 course slides, high-level overview on Bullet collision detection
3.GDC 2010 presentation about contact generation
4.最大的资料自然就是Bullet的源代码啦。。。。。。慢慢研究吧。

参考资料

1.Bullet 2.76 Physics SDK Manual，Bullet的项目发起人，目前的负责人Erwin Coumans所写，（就WIKI资料显示，这哥们现在在SONY)算是官方的Manual了，源码包中就有pdf。
2.Bullet WIKI Tutorial Articles，算是第2个能够找到的稍微好点的关于Bullet的东西了，就是有点散乱。
3.Bullet Bullet Documentation，Bullet的文档，自动生成的，也就只能在写代码的时候可能有些用，很难靠这个学习。

我写的关于**Bullet**的文章

1.Ogre与Bullet的整合(on iPhone)
2.Ogre的3D Max插件介绍