```
//BulletContactCallback.h - the callback code

#pragma once

#include <btBulletDynamicsCommon.h>
class GameObject;

struct CollisionContext {
	bool hit;
	const btCollisionObject* body;
	const btCollisionObject* lastBody;
	GameObject* theObject;
	float distance;
	float velNorm;
	btVector3 point;
	btVector3 normal;
	btVector3 velocity;

	CollisionContext() {
		reset();
	}

	void reset() {
		hit = false;
		body = NULL;
		theObject = NULL;
		distance = 0.0;
		velNorm = 0.0;
		point.setZero();
		normal.setZero();
		velocity.setZero();
	}
};

struct BulletContactCallback : public btCollisionWorld::ContactResultCallback {

	//! Constructor, pass whatever context you want to have available when processing
contacts
	/*! You may also want to set m_collisionFilterGroup and m_collisionFilterMask
	 *  (supplied by the superclass) for needsCollision() */
	BulletContactCallback(btRigidBody& tgtBody , CollisionContext& context /*, ... */)
		: btCollisionWorld::ContactResultCallback(), body(tgtBody), ctxt(context)
{ }

	btRigidBody& body; //!< The body the sensor is monitoring
	CollisionContext& ctxt; //!< External information for contact processing

	//! If you don't want to consider collisions where the bodies are joined by a
constraint, override needsCollision:
	/*! However, if you use a btCollisionObject for #body instead of a btRigidBody,
	 *  then this is unnecessaryócheckCollideWithOverride isn't available */
	virtual bool needsCollision(btBroadphaseProxy* proxy) const {
		// superclass will check m_collisionFilterGroup and m_collisionFilterMask
		if(!btCollisionWorld::ContactResultCallback::needsCollision(proxy))
```

```
                        return false;
                // if passed filters, may also want to avoid contacts between constraints
                return
body.checkCollideWithOverride(static_cast<btCollisionObject*>(proxy->m_clientObject));
        }

        //! Called with each contact for your own processing
        virtual btScalar addSingleResult(btManifoldPoint& cp,
                const btCollisionObject* colObj0, int partId0, int index0,
                const btCollisionObject* colObj1, int partId1, int index1) {

                ctxt.hit = true;
                ctxt.lastBody = ctxt.body;
                if(colObj0 == &body) {
                        ctxt.point = cp.m_localPointA;
                        ctxt.body = colObj1;
                } else {
                        assert(colObj1 == &body && "body does not match either collision
object");
                        ctxt.point = cp.m_localPointB;
                        ctxt.body = colObj0;
                }
                ctxt.theObject = static_cast<GameObject*>(ctxt.body->getUserPointer());
                ctxt.normal = cp.m_normalWorldOnB;
                ctxt.velocity = body.getLinearVelocity();
                ctxt.velNorm = ctxt.normal.dot(ctxt.velocity);

                return 0;
        }
};

// declarations in game object class

class GameObject {
protected:
        Ogre::String name;
        Ogre::SceneManager* sceneMgr;
        Ogre::SceneNode* rootNode;
        Ogre::Entity* geom;
        OgreMotionState* motionState;

        Simulator* simulator;
        btCollisionShape* shape;
        btRigidBody* body;
        btTransform tr;
        btVector3 inertia;

        btScalar mass;
        btScalar restitution;
        btScalar friction;
        bool kinematic;
        bool needsUpdates;

        CollisionContext* context;
```

```
        BulletContactCallback* cCallBack;
…

// game object class add to simulator

void GameObject::addToSimulator() {
        //using motionstate is recommended, it provides interpolation capabilities, and
only synchronizes 'active' objects
        updateTransform();
        //rigidbody is dynamic if and only if mass is non zero, otherwise static
        if (mass != 0.0f) shape->calculateLocalInertia(mass, inertia);
        btRigidBody::btRigidBodyConstructionInfo rbInfo(mass, motionState, shape,
inertia);
        rbInfo.m_restitution = restitution;
        rbInfo.m_friction = friction;
        body = new btRigidBody(rbInfo);
        body->setUserPointer(this);
//    context = new SimContext();
        if (kinematic) {
                body->setCollisionFlags(body->getCollisionFlags() |
btCollisionObject::CF_KINEMATIC_OBJECT);
                body->setActivationState(DISABLE_DEACTIVATION);
        }
        simID = simulator->addObject(this);
}

// specific game object update routine

void Ball::update(float elapsedTime) {
    while (simulator->checkHit(simID)) {
        if (context->velNorm > 5.0 || context->velNorm < -5.0) {
            if (context->theObject->getName() != "Player") soundMgr->playClip(bounceClip);
        }
    }

// do time step and check for hits in simulator

void Simulator::stepSimulation(const Ogre::Real elapsedTime, int maxSubSteps, const
Ogre::Real fixedTimestep) {
        // do we need to update positions in simulator for dynamic objects?
    for (int i = 0; i != objList.size(); i++) idList[i] = 0;
        dynamicsWorld->stepSimulation(elapsedTime, maxSubSteps, fixedTimestep);
        for (unsigned int i = 0; i < objList.size(); i++)
                if (objList[i].gObject->doUpdates()) objList[i].gObject-
>update(elapsedTime);
}

bool Simulator::checkHit(int o) {
    for (int i = idList[o]; i < objList.size(); i++) {
        if (i != o) {
            objList[o].gObject->context->hit = false;
            dynamicsWorld->contactPairTest(objList[o].gObject->getBody(),
objList[i].gObject->getBody(), objList[o]);
            if (objList[o].gObject->context->hit) {
```

```
                idList[o] = ++i;
                return true;
            }
        }
    }
    return false;
}
```