CS 378: Computer Game Technology

Using Bullet Physics Library

Spring 2012

Building an app with Ogre and Bullet

- Make sure they all link similarly
  - On Windows, make sure they all use the same C runtime library
  - This means you'll probably need to build at least one of them from source
- OgreBullet may not be the simplest way to go
- Examples without OgreBullet

Ogre-Bullet Interface

```
class OgreMotionState : public btMotionState {
protected:
    Ogre::SceneNode* mVisibleobj;
    btTransform mPos1;
public:
    OgreMotionState(const btTransform& initialpos, Ogre::SceneNode* node) {
        mVisibleobj = node;
        mPos1 = initialpos;
    }
    virtual ~OgreMotionState() { }

    void setNode(Ogre::SceneNode* node) {
        mVisibleobj = node;
    }
    void updateTransform(btTransform& newpos) {
        mPos1 = newpos;
    }
```

Communicating with Bullet

```
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());
   }
};
```

Simulation

```
class Simulator {
protected:
        btDefaultCollisionConfiguration* collisionConfiguration;
        btCollisionDispatcher* dispatcher;
        btBroadphaseInterface* overlappingPairCache;
        btSequentialImpulseConstraintSolver* solver;
        btDiscreteDynamicsWorld* dynamicsWorld;
        btConstraintSolver* mConstraintsolver;
//      btCollisionWorld* mWorld;
        Ogre::SceneManager* sceneMgr;
        std::deque<GameObject*> objList;
public:
        Simulator();
        ~Simulator();

        void addObject(GameObject* o);
        bool removeObject(GameObject* o);
        void stepSimulation(const Ogre::Real elapsedTime,
                int maxSubSteps = 1, const Ogre::Real fixedTimestep = 1.0f/60.0f);

};
```

Simulation

```
Simulator::Simulator() {
  ///collision configuration contains default setup for memory, collision setup.
  collisionConfiguration = new btDefaultCollisionConfiguration();
  ///use the default collision dispatcher. For parallel processing you can use a diffent dispatcher
  dispatcher = new btCollisionDispatcher(collisionConfiguration);
  ///btDbvtBroadphase is a good general purpose broadphase. You can also try out btAxis3Sweep.
  overlappingPairCache = new btDbvtBroadphase();
  ///the default constraint solver. For parallel processing you can use a different solver (see Extras/BulletMultiThreaded)
  solver = new btSequentialImpulseConstraintSolver();
  dynamicsWorld = new btDiscreteDynamicsWorld(dispatcher,overlappingPairCache,solver,collisionConfiguration);
  dynamicsWorld->setGravity(btVector3(0,-0.098, 0));
  //keep track of the shapes, we release memory at exit.
  //make sure to re-use collision shapes among rigid bodies whenever possible!
  btAlignedObjectArray<btCollisionShape*> collisionShapes;
}
void Simulator::addObject (GameObject* o) {
        objList.push_back(o);
        // use default collision group/mask values (dynamic/kinematic/static)
        dynamicsWorld->addRigidBody(o->getBody());
}
void Simulator::stepSimulation(const Ogre::Real elapsedTime, int maxSubSteps, const Ogre::Real fixedTimestep) {
        // do we need to update positions in simulator for dynamic objects?
        dynamicsWorld->stepSimulation(elapsedTime, maxSubSteps, fixedTimestep);
}
```

Game Objects

```
class GameObject {
protected:
        Ogre::String name;
        Ogre::SceneManager* sceneMgr;
        Simulator* simulator;
        Ogre::SceneNode* rootNode;
        Ogre::Entity* geom;
        btCollisionShape* shape;
        btScalar mass;
        btRigidBody* body;
        btTransform tr;
        btVector3 inertia;
        OgreMotionState* motionState;
```

Game Objects

```
GameObject::GameObject(Ogre::String nym, Ogre::SceneManager* mgr, Simulator* sim) {
        name = nym;
        sceneMgr = mgr;
        simulator = sim;
        rootNode = sceneMgr->getRootSceneNode()->createChildSceneNode(name);
        shape = NULL;
        tr.setIdentity();
        mass = 0.0f;
        inertia.setZero();
}
void GameObject::updateTransform() {
        Ogre::Vector3 pos = rootNode->getPosition();
        tr.setOrigin(btVector3(pos.x, pos.y, pos.z));
        Ogre::Quaternion qt = rootNode->getOrientation();
        tr.setRotation(btQuaternion(qt.x, qt.y, qt.z, qt.w));
        if (motionState) motionState->updateTransform(tr);
}
void GameObject::addToSimulator() {
        //using motionstate is recommended, it provides interpolation capabilities, and only synchronizes 'active' objects
        updateTransform();
        motionState = new OgreMotionState(tr, rootNode);
        //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);
        body = new btRigidBody(rbInfo);
        simulator->addObject(this);
}
```