CSc 165
Computer Game Architecture

13 - Physics Engines & Collisions

Overview

Collision Detection

- Broad- vs. Narrow-phase

Collision Handling & Physics

Physics Engines

- Rigid Bodies and Joints
- Collision Spaces

Integrating Physics & Game Engines

Naïve Collision Detection

```
// Check for collisions between world objects by comparing
// all possible combinations of objects  (as done in CSc-133)

void checkCollisions()
{   Iterator iter1 = theWorld.iterator();
    while (iter1.hasNext())
    {   //get a world object
        ICollider curObj = (ICollider) iter1.next();
        Iterator iter2 = theWorld.iterator();
        while (iter2.hasNext())
        {   //get a second object
            ICollider otherObj = (ICollider) iter2.next();
            //insure it's not the SAME object
            if (otherObj != curObj)
            {   //check for collision and handle it
                if (curObj.collidesWith(otherObj))
                    curObj.handleCollision(otherObj);
            }
        }
    }
}
```

Two-Phase Collision Detection

Broad-phase:

select pairs of objects that *might* have collided

Goals:

- Try to avoid selecting pairs that *can't* collide
- Try to make these sections quickly

Narrow-phase:

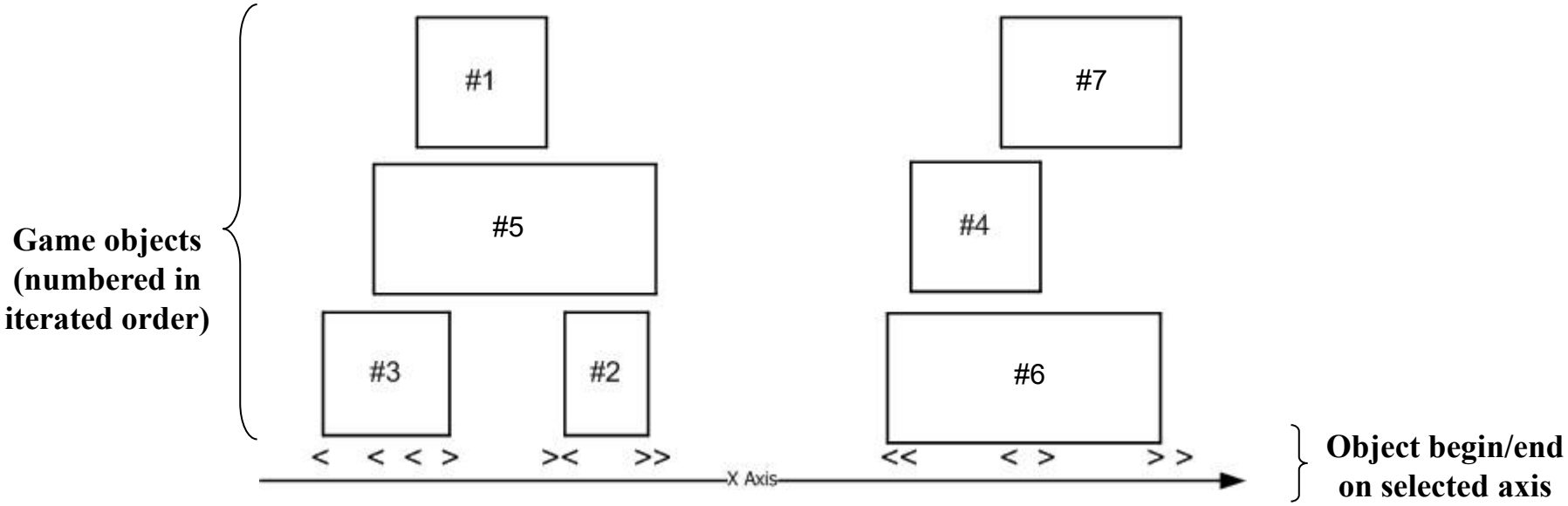
compare those pairs to see if they actually collided

Goals:

- *Accurate* comparisons
- *Efficient* comparisons

Broad-phase (step 1)

"*Sweep And Prune" (SAP)* Algorithm



Unsorted:

<1	>1	<2	>2	<3	>3	<4	>4	<5	>5	<6	>6	<7	>7

After sorting:

<3	<5	<1	>3	>1	<2	>2	>5	<6	<4	<7	>4	>6	>7

after: Broadphase Collision Detection, John Wells

Sweep And Prune (cont.)

Identifying "interference" (potential collisions):

- *Prune* (skip) non-interfering objects

After sorting:

<3	<5	<1	>3	>1	<2	>2	>5	<6	<4	<7	>4	>6	>7

```
for (each world object x)
{   i = indexOf(x.beginDelimiter);                    //start of cur obj in array
    for (j=i+1; j<indexOf(x.endDelimiter); j++)       //check all locs btwn xBegin/xEnd
    {   if (array[j] instanceof BeginDelimiter)       //check for new object start
        {   checkForCollision(array[i], array[j]);    //found interference
        }
        else if (array[j] instanceof EndDelimiter)          //check for obj end
        {   start = indexOf(array[j].obj.beginDelimiter);   //find matching start
            if (start < i)                  //check if new obj start was before cur obj
                checkForCollision(array[i], array[j]);      //found interference
        }
    }
}
```

Narrow-phase (step 2)

For hierarchical objects, using *Bounding Volumes:*

```
public class SceneNode
{  ...
   boolean collidesWith (SceneNode otherNode)
   {  if ( ! this.getWorldBound().intersects(otherNode.getWorldBound() )
      {  return false ;                    //world BV's don't intersect
      }
      else if (this.hasChildren())
      {    for (each child of this node)
           {  if (child.intersects(otherNode))
                return true ;              //found a child that intersects
           }
           return false ;                  //no child intersects
      }
      else if (otherNode.hasChildren())
      {    for (each child of otherNode)
           {  if (child.collidesWith(this))
                return true ;
           }
           return false ;
      }
   }
}
```

Collision Handling

(Some) factors to consider

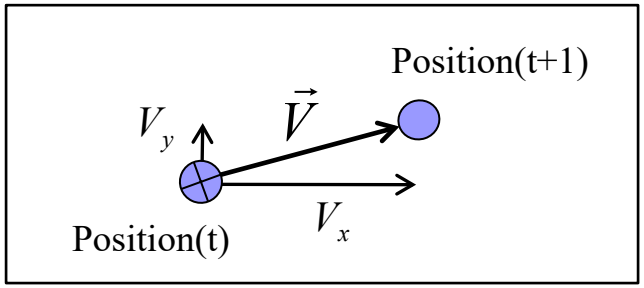
- Position
- Orientation
- Linear velocity (change in position)
- Angular velocity (change in orientation)
- Friction
- Air lift/drag
- Water resistance/buoyancy
- Gravity
- Elasticity

Using Physics In Games

Ad-hoc solutions:

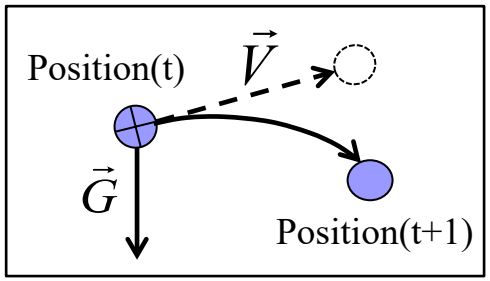
- Constant velocity

```
newPos = curPos + velocity * elapsedTime ;
```



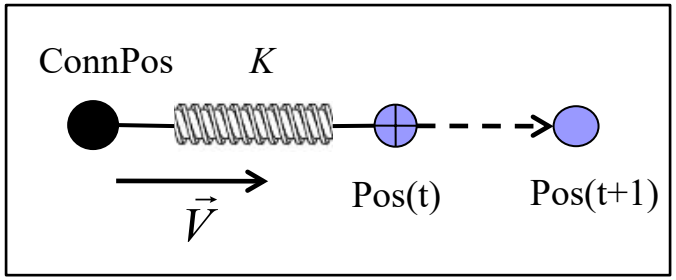
- Gravity

```
velocityY = velocityY + GRAVITY * elapsedTime ;
newPosY = curPosY + velocityY * elapsedTime ;
```



- Springs

```
k = 0.2;  //spring constant
springVector = mass.pos - connectionPos;
mass.pos = mass.applyForce(-springVector*k);
```



(Ordinary) Differential Equations

- Equations defining a relationship between a single-variable function $x(t)$ *and its derivatives*

$$\frac{d\,x(t)}{dt}, \quad \frac{d^2x(t)}{dt^2}, \quad \ldots$$

- ex. : Newton's Second Law of motion

$$F = ma; \quad F(x(t)) = m\frac{d^2x(t)}{dt^2}$$

Force on particle at position X at time t

Acceleration (2nd derivative of position)

Physics Engines

Integrate laws of physics into a game

- Laws are represented by ODE's
- Physics Engines contain " ODE solvers"

Independent of "gameplay"

- Analogous to how a Game Engine supplies independent "renderers"
- Usable for a wide number of genres:
  - Race simulation
  - FPS action-shooter
  - Virtual world dynamic structures
  - Space travel, planets, etc.
  - Sports

Popular Physics Engines

Havok (www.havok.com)

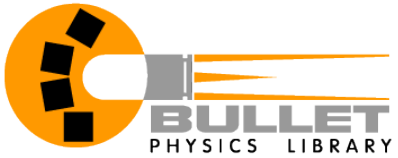
Newton (www.newtondynamics.com)

PhysX (www.nvidia.com/physX)

- jPhysX (www.jphysX.com)

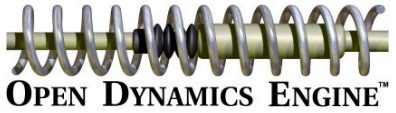
Bullet (www.bulletphysics.org)

- JBullet (jbullet.advel.cz)



Open Dynamics Engine (ODE) (www.ode.org)

- ODEJava (odejava.dev.java.net)



Chipmunk (http://chipmunk-physics.net)

- Chipmunk-for-Java (https://github.com/johang/chipmunk-for-java)

Physics Engine Concepts

`World` ("Physics Space")

A container for **`Bodies`**, **`Constraints`**, **`CollisionShapes`**, **`CollisionHandlers`**, and **`Solvers`**

`Body`

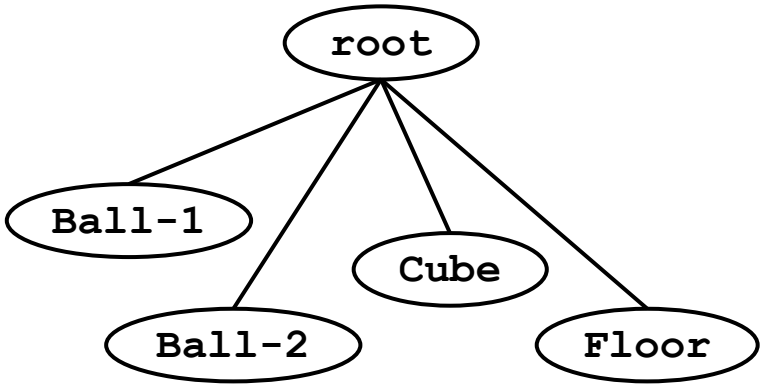
Represents a single rigid or soft body.
Most common is rigid:
Fixed attributes: Mass, Mass distribution
Dynamic Attributes: Position, Orientation, Velocity, Angular velocity

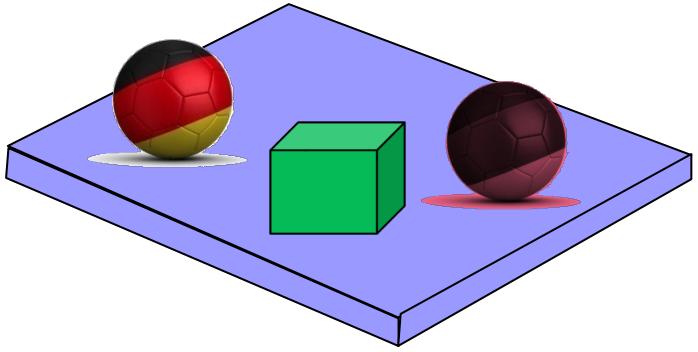
`Constraint` ("joint")

Represents a connection between two bodies

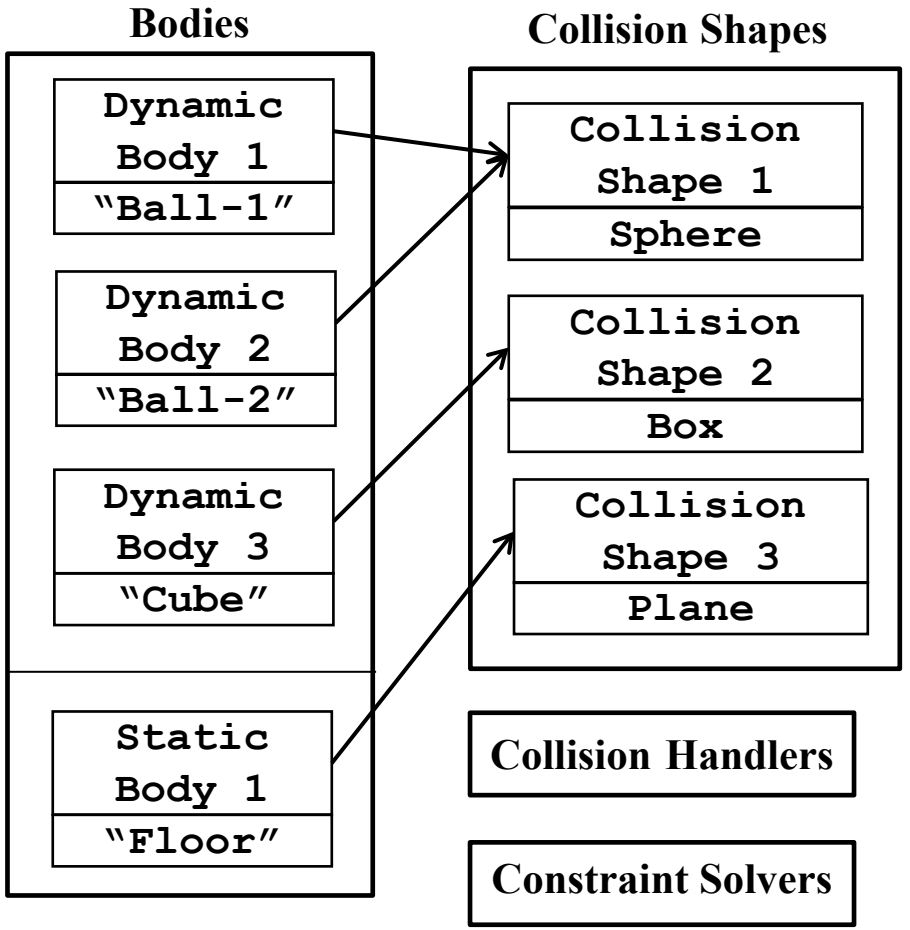
Physics Spaces

GameWorld





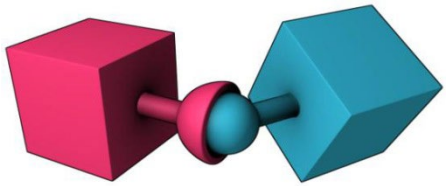
Physics World



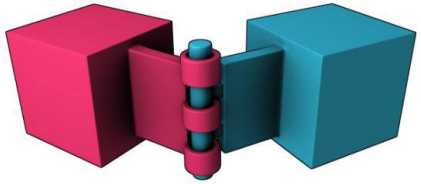
Constraints (a.k.a. *Joints*)

Enforced relationships between *bodies.*
(created at start-up)

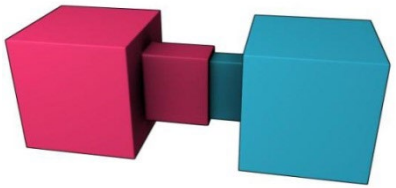
Common types of constraints:



Ball & Socket
("Point-to-Point")



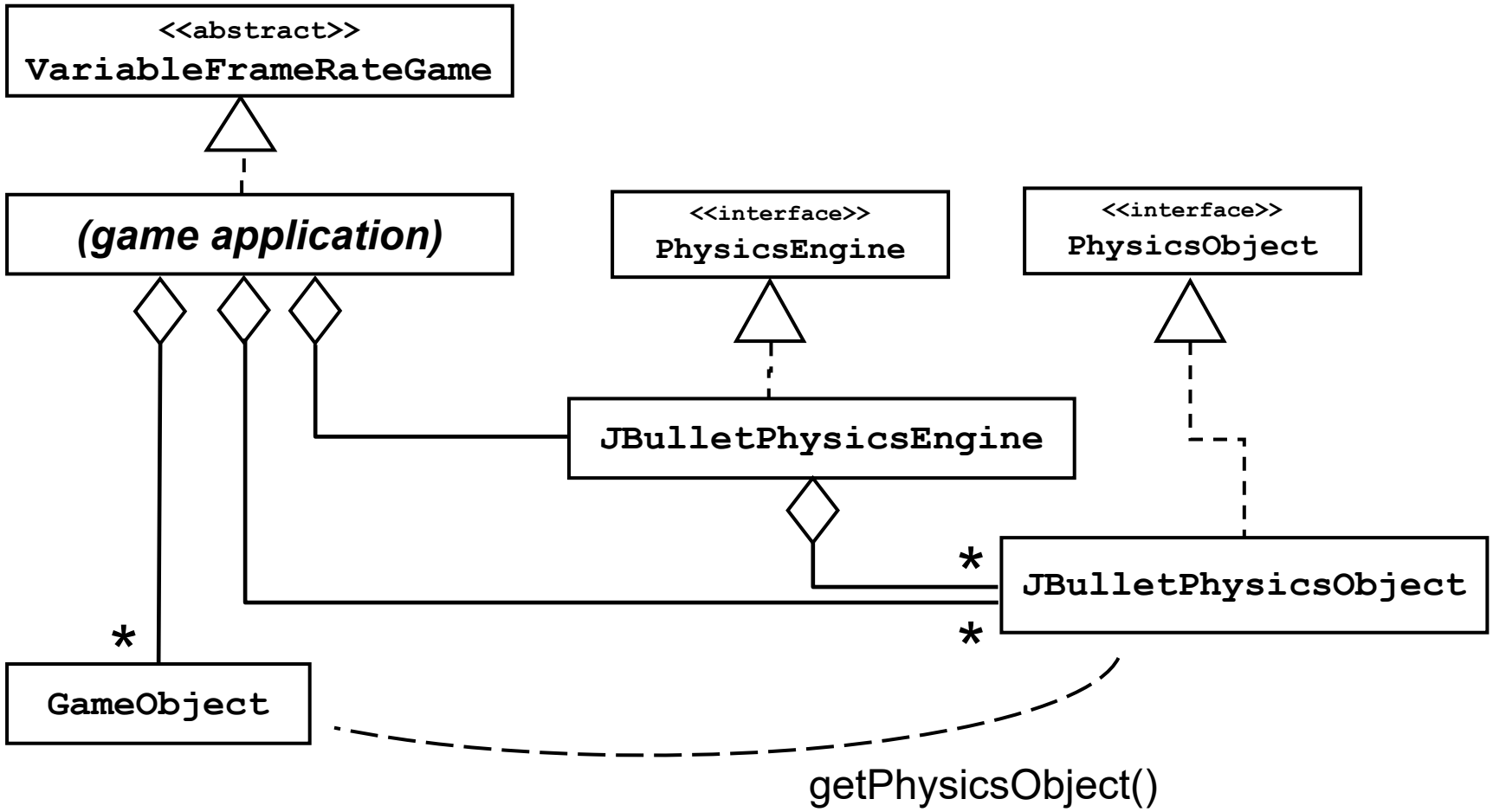
Hinge



Slider
("Piston")

Source: Bullet 2.76 Physics SDK Manual, Erwin Coumans

TAGE Physics World classes



TAGE Physics Engine Interface

```
/**  TAGE interface  */
public interface PhysicsEngine
{   public void initSystem();              //sets defaults for gravity, collisions, etc.
    public void setGravity(float[] gravity_vector);     //sets gravity explicitly
    public int nextUID(); // unique identifier to keep track of physics objects

    public PhysicsObject addBoxObject(int uid, float mass, double[] transform,
                                      float[] size);
    public PhysicsObject addSphereObject (int uid, float mass, double[] transform,
                                        float radius);
    public PhysicsObject addConeObject (int uid, float mass, double[] transform,
                                      float radius, float height);
    public PhysicsObject addCapsuleObject (int uid, float mass, double[] transform,
                                        float radius, float height);
    public PhysicsObject addCylinderObject (int uid, float mass, double[] transform,
                                        float[] halfExtents);
    public PhysicsObject addStaticPlaneObject (int uid, double[] transform,
                                          float[] up_vector, float plane_constant);
    public void removeObject(int uid);

    public void addBallSocketConstraint(int uid, PhysicsObject bodyA, PhysicsObject bodyB)
    public void addHingeConstraint(int uid, PhysicsObject bodyA, PhysicsObject bodyB)
    public void update(float nanoseconds);   //steps the physics simulation
}
```

Physics Object Interface

```
/** Defines the interface implemented by all Physics Objects */

public interface IPhysicsObject

{  public int getUID();

   public void setTransform(double[] transform);
   public double[] getTransform();

   public float getFriction();
   public void setFriction(float friction);

   public float getLinearDamping();
   public float getAnglularDamping();
   public void setDamping(float linearDamping, float angularDamping);

   public float getBounciness();
   public void setBounciness(float value);

   public float[] getLinearVelocity();
   public void setLinearVelocity(float[] velocity);

   public float[] getAngularVelocity();
   public void setAngularVelocity(float[] velocity);

   public void applyForce(float fx, float fy, float fz, float px, float py, float pz);
   public void applyTorque(float fx, float fy, float fz);

   public boolean isDynamic();
}
```

Collision Detection

```
private void checkForCollisions()
{   com.bulletphysics.dynamics.DynamicsWorld dynamicsWorld;
    com.bulletphysics.collision.broadphase.Dispatcher dispatcher;
    com.bulletphysics.collision.narrowphase.PersistentManifold manifold;
    com.bulletphysics.dynamics.RigidBody object1, object2;
    com.bulletphysics.collision.narrowphase.ManifoldPoint contactPoint;

    dynamicsWorld = ((JBulletPhysicsEngine)physicsEngine).getDynamicsWorld();
    dispatcher = dynamicsWorld.getDispatcher();
    int manifoldCount = dispatcher.getNumManifolds();

    for (int i=0; i<manifoldCount; i++)
    {   manifold = dispatcher.getManifoldByIndexInternal(i);
        object1 = (com.bulletphysics.dynamics.RigidBody)manifold.getBody0();
        object2 = (com.bulletphysics.dynamics.RigidBody)manifold.getBody1();
        JBulletPhysicsObject obj1 =
                        JBulletPhysicsObject.getJBulletPhysicsObject(object1);
        JBulletPhysicsObject obj2 =
                        JBulletPhysicsObject.getJBulletPhysicsObject(object2);
        for (int j = 0; j < manifold.getNumContacts(); j++)
        {   contactPoint = manifold.getContactPoint(j);
            if (contactPoint.getDistance() < 0.0f)
            {   System.out.println("collision: " + obj1 + " and " + obj2);
                break;
}   }   }   }
```