Collision Detection

Almost all game engines make use of some form of collision handling. Detecting and resolving collisions between entities in a game accurately and efficiently is very important, otherwise you wouldn’t be able to tell if a bullet shot an enemy, or you would be able to walk through walls (which, unless you’re a ghost, isn’t that suitable for most games).

Collision systems are usually made up of two components: collision *detection* and collision *resolution*. In this tutorial, we’ll be looking at both of these and by the end of this session you’ll be able to knock some ships around by pushing them out of your way.

# Chase Camera Sample

There is a modified version of last week’s lab available on Moodle. Download and extract the package from there.

**Compile and run the sample** and you’ll notice a few small changes from last week. The ship now has a green wireframe sphere around it and there are a number of other ships dotted around the world encased in white wireframe spheres. You can move the ship around as normal and you’ll be able to fly straight through the other spaceships.

# Making the Ships Collision Friendly

If you take a look at the code, you’ll notice that the *Ship* class now inherits from a class called *Collidable*. The *Collidable* class contains a protected *BoundingSphere* member as well as two virtual functions: *CollisionTest()* and *OnCollision()*.

XNA provides us with a number of bounding volume types, including *BoundingSphere*, *BoundingBox* and *BoundingFrustum*. Each of these provides an *Intersects()* function that can check whether two bounding volumes are intersecting and return either true or false.

Since both the *CollisionTest()* and the *OnCollision()* functions are virtual, we can override them to do any custom collision checking code. Let’s add some into the *Ship* class.

**Open *“Ship.cs”* and add the following code to the Ship class (make sure this is inside the curly bracket for this class, so just after the definition of *MouseTurn* is a good place):**

public override bool CollisionTest(Collidable obj)

{

if (obj != null)

{

return BoundingSphere.Intersects(obj.BoundingSphere);

}

return false;

}

public override void OnCollision(Collidable obj)

{

}

This simply calls the intersection test function for the two bounding spheres and returns the result of the test. At the moment the *CollisionTest* method is not being called. We’ll fix that next.

# Managing Collisions

Now that we have a basic collision test written, we need some way of handling all of the collisions in a game world. One method of doing this would be during each entity’s *Update()* function. However the disadvantages of this are that: collision updates would end up being tightly coupled to entity updates; and the positions of the objects would depend on the order that the objects were updated. Instead, we can create a *CollisionManager* that maintains a list of all entities that can collide, checks collisions between them, and then resolves them.

**Add a new class to the project (Add > Class) and call the file *“CollisionManager.cs”* and give it the following implementation:**

public class CollisionManager

{

private List<Collidable> m\_Collidables = new List<Collidable>();

private HashSet<Collision> m\_Collisions = new HashSet<Collision>(new CollisionComparer());

public void AddCollidable(Collidable c)

{

m\_Collidables.Add(c);

}

public void Update()

{

UpdateCollisions();

ResolveCollisions();

}

private void UpdateCollisions()

{

}

private void ResolveCollisions()

{

}

}

This is a simple class that has two containers. The first is a list of all “collidables”, i.e. classes that inherit from *Collidable*. The second is a little more interesting; a *HashSet* to contain all collisions in a single update. A *HashSet* is used because it does not contain any duplicate entries, which is particularly important to make sure we don’t add the same collision twice (otherwise a collision could be resolved twice). The *HashSet* also makes use of a *CollisionComparer* class (provided in *“Collision.cs”*). This is a small class that overrides two methods (*Equals* and *GetHashCode*) so that the *HashSet* knows how to check whether two entries are equal.

We also make use of the *Collision* class here. Let’s take a look at it now so **open up *“Collision.cs”***.

public class Collision

{

public Collidable A;

public Collidable B;

public Collision(Collidable a, Collidable b)

{

A = a;

B = b;

}

}

As you can see, the class just contains two *Collidable* objects. We need to add some functionality in here to make it work nicely with the *HashSet* in the *CollisionManager,* and to allow it call the appropriate *OnCollision()* function.

**Add the following code to the class:**

public bool Equals(Collision other)

{

if (other == null) return false;

if ((this.A.Equals(other.A) && this.B.Equals(other.B)))

{

return true;

}

return false;

}

public void Resolve()

{

this.A.OnCollision(this.B);

}

Now that we’ve got the basic classes created, we can start checking for collisions. The *CollisionManager* will handle this for us in its *UpdateCollisions()* method. It will iterate through every *Collidable* object stored in its list, check for collisions, and add any *Collision* objects to resolve later.

**In *CollisionManager* add the following code to the *UpdateCollisions()* method:**

private void UpdateCollisions()

{

if (m\_Collisions.Count > 0)

{

m\_Collisions.Clear();

}

// Iterate through collidable objects and test for collisions between each one

for (int i = 0; i < m\_Collidables.Count; i++)

{

for (int j = 0; j < m\_Collidables.Count; j++)

{

Collidable collidable1 = m\_Collidables[i];

Collidable collidable2 = m\_Collidables[j];

// Make sure we're not checking an object with itself

if (!collidable1.Equals(collidable2))

{

// If the two objects are colliding then add them to the set

if (collidable1.CollisionTest(collidable2))

{

m\_Collisions.Add(new Collision(collidable1, collidable2));

}

}

}

}

}

Before testing if this works, we need to add a *CollisionManager* object to the main class, add some collidable objects and update it.

**Open up *“ChaseCameraGame.cs”* and in the class add a *CollisionManager* member. Initialize it in the constructor and add a call to *CollisionManager.Update()* near the top of the *ChaseCameraGame.Update()* method.**

*ChaseCameraGame* also contains the empty method *InitializeCollidableObjects()*.

**Add code to *InitializeCollidableObjects()* that will add the *ship* member and each enemy in the *enemies* list to the *CollisionManager*, using the *AddCollidable* method.**

To test our collision detection method now, **set a breakpoint in *CollisionManager.UpdateCollisions()* at the following line:**

m\_Collisions.Add(new Collision(collidable1, collidable2));

**Compile and run the sample.** You should now be able to move the ship around. Try manoeuvring towards an enemy and as soon as you make contact the debugger should hit the breakpoint.

Great! We’ve now got working collision detection. The next step is to try and do something with this information.

# Resolving Collisions

The other main component for collision systems is *collision resolution* (“What happens to two objects when they collide?”).

Let’s finish the *CollisionManager* class now by adding code to the *ResolveCollisions()* function we made earlier. In fact, this is a simple addition to make. Remember that the *Collision* class has a method called *Resolve()* that calls the appropriate *OnCollision()* function, so all we need to do is iterate through all of the collisions in the *HashSet* and call *Resolve()* in each one. **Try to add this in to *ResolveCollisions()* yourself.**

We’ll now add some code to stop the ships from flying through each other and make it look like that they are bumping into one another.

**In the *Ship* class add the following code to the *OnCollision()* method:**

public override void OnCollision(Collidable obj)

{

// Cast the object as a ship

Ship otherShip = obj as Ship;

if (otherShip != null)

{

// The collision normal is the direction in which the collision occurred

// We want the ships to react in this direction

Vector3 collisionNormal = Vector3.Normalize(otherShip.BoundingSphere.Center - BoundingSphere.Center);

// The distance between the two ships can be calculated using the centers of the two bounding spheres

float distance = Vector3.Distance(otherShip.BoundingSphere.Center, BoundingSphere.Center);

// The penetration depth determines how much the two spheres have intersected

float penetrationDepth = (otherShip.BoundingSphere.Radius + BoundingSphere.Radius) - distance;

// Negate the collision normal as we want to act in the opposite direction

// of the collision and multiply by how much the spheres have intersected

AddPosition(-collisionNormal \* penetrationDepth);

}

}

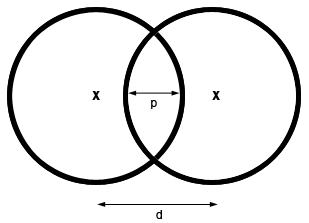
There are a number of things to take on board here. The first line makes use of a useful feature of C# - the *as* keyword. This is equivalent to C++’s *dynamic\_cast*, although a lot safer. If you were to do the following:

Ship otherShip = (Ship)obj;

Where *obj* is not a *Ship* or a derived class from *Ship*, then this would throw an exception. If the same thing happened with the *as* keyword *otherShip* would simply be set to *null*. No exceptions are thrown and so it makes it easy to check whether the value is expected just by comparing against *null* and without the need for a *try-catch­* statement.

Now we get to the actual collision response. The collision normal gives us the direction in which the collision occurred which is found by calculating the vector between the two bounding sphere centre points and normalizing it to get a unit vector.

The penetration depth can be shown in the following diagram:



*p* can be calculated by adding up the two radii of the bounding spheres and then subtracting the actual distance of the spheres (*d*).

*Note:* this is very similar to the code used to find out if two bounding spheres are in collision.

**Compile and run the sample** and you should see that if you move around and collide with the enemy ships, it will feel like an actual collision (no more flying through ghost ships), although currently the enemy ships aren’t going anywhere.

**In the *EnemyShip* class (not the *Ship* class) add the following code:**

public override void OnCollision(Collidable obj)

{

Ship playerShip = obj as Ship;

if (playerShip != null)

{

Vector3 collisionNormal = Vector3.Normalize(playerShip.BoundingSphere.Center - BoundingSphere.Center);

AddPosition(-collisionNormal \* 50.0f);

}

}

Run it again and you will find that you are able to bully some enemy ships around.

# Extra challenges:

## Task 1:

**In the *EnemyShip* class there is member called *CollisionColor*. See if you can get enemy ships to change the colour of their bounding sphere when they collide with the player. Hint: the appropriate function to draw the wireframe sphere is in the *ChaseCameraGame.Draw()* function called *DrawWireSphere()*.**

## Task 2:

**Remove EnemyShips from the game world after collisions. Hint: try adding a “flagForRemoval” member to the Collidable class, then remove all EnemyShips that have been flagged for removal in a new ResolveRemovals() method of the ChaseCameraGame. If you are using a *foreach* loop, you will need to build a new list of elements to delete and then remove these from the *enemies* and *collisionManager.m\_Collidables* lists in a separate loop (you cannot delete elements from a *foreach* loop inside the loop). You will also need to add a “RemoveCollidable” method to the CollisionManager to access the private *m\_Collidables* list.**

# Concluding Remarks:

In this tutorial we’ve seen how we can make use of XNA’s inbuilt bounding volume classes to detect collisions between objects, and also how to respond to those collisions within the game world. Creating a manager to handle the collisions for us provides a separate subsystem that is loosely coupled from everything else, which is vital to good game engine architecture.

While completing the above tasks you may have noticed that some of the functionality above could be made more general.

* Our Collidable class is not very flexible and we are likely to repeat the collision test code in every game object we create. It could be improved by containing a BoundingShape class, which different types of bounding object (including BoundingSphere) could inherit from.
* We may want to remove game objects even if they are not collidable (e.g. in response to an end of level event). It would be helpful to have a more general GameObject base class that encapsulates collision detection if it is needed. We could then store a list of GameObjects and remove them when flagged, rather than keeping separate lists for enemies and ships. This functionality could also be moved to a general “game engine” base-class that the ChaseCamera game inherits from.
* The code could be further improved by the use of events. If objects generate collision events, additional listeners (such as a sound subsystem) could register to respond to the events. This would create an even more loosely coupled architecture, because the event generating code does not need to know about the code that responds.