Physics - Collision Detection

Introduction

One of the most important aspects of a realistic physics system is the ability to detect and resolve the collisions that occur between the objects in the simulation - a snooker game isn't very fun if we can't knock balls around by hitting the cue ball with our cue!

The following two tutorials are going to show you the basics behind the process of *collision resolution*, which will greatly increase the realism of your physics simulation. In this tutorial, we're going to see how to use *collision detection* algorithms determine whether two objects are colliding in some way - that is, their shapes are touching or overlapping due to their movement over time. Detecting this, and determining how much overlap, and the positions in space the objects hit each other are the first steps towards creating realistic collision resolution. The next tutorial will deal with *collision response*. As objects intersecting each other isn't physically possible, we have to maintain the realism of the simulation by resolving those collisions by separating them out so that they aren't overlapping, and determining whether any kinetic energy should be transferred from one object to another - such as when we hit that cue ball, sending it flying off down the table.

Collision Volumes

We saw in the previous tutorial in raycasting, that there's a set of standard *collision volumes* used to represent most of the physics-enabled objects in our game simulations. Just as with rays, the process of detecting collisions between each combination of these shapes is a little bit different, so we will have to define a series of functions to handle each potential collision that can occur each simulation frame. To this end, in this tutorial we will investigate the most common collisions that we need to detect: sphere versus sphere, AABB versus AABB, and AABB versus sphere.

Collision Detection

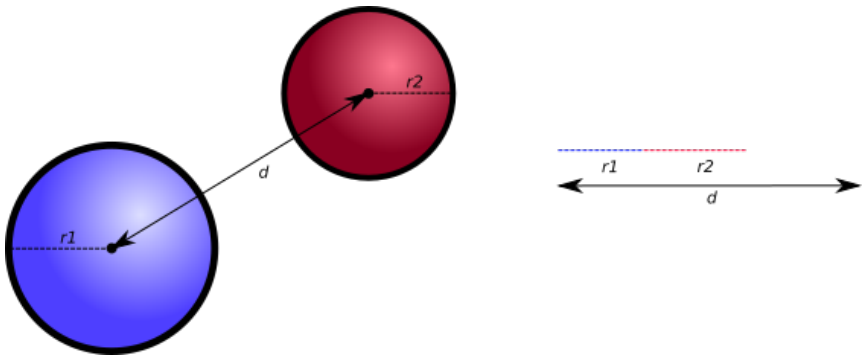
Of course, there are more complex collision volumes than simple boxes, and we'll see in a later tutorial how to enhance collision detection to support any convex shape. Don't discount the usefulness of simpler shapes like boxes and spheres, though! The calculations to determine collisions between them are generally quite fast, allowing lots of them to be calculated per frame. Also, sometimes we just simply don't need the additional fidelity that more complex collision detection would bring. Consider the following example of a character picking up a powerup by walking over it:

Does it really matter if we work out exactly which polygon of the powerup was touched by which polygon of the character's feet? Probably not, so we may as well just model this interaction as a simple box / box intersection test.

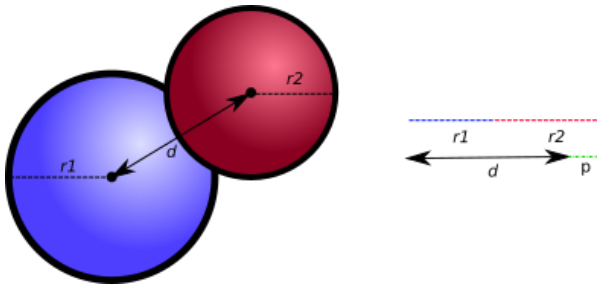
As well as knowing *if* two objects are colliding with each other, we also need to know details on exactly *how* they are colliding. We need to know roughly which bits of the objects are intersecting each other, so that they can be separated, and by roughly how much they are intersecting (sometimes known as the *penetration distance*), to calculate the forces that will separate them out again. To help with this, collision detection functions usually don't just return a true or false answer, but also some additional collision information - in our examples we'll be determining a collision point (or *contact* point), and an intersection distance and collision normal.

Sphere / Sphere Collisions

Determining whether two spheres have overlapped is fairly easy. If the distance between two spheres is *greater* than the sum of their radii, they **can't** be colliding:



From this, it is easy to then say that for any two bounding spheres whose distance d between them is *less* than the sum of their radii, **must** be colliding:

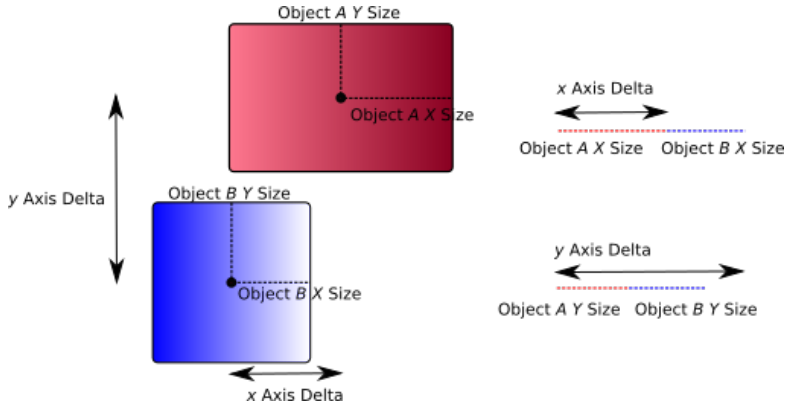


Determining the collision normal n is similarly easy - it's just the direction vector between the objects. It's good to establish a consistent collision normal direction, so we'll state that if objects A and B are colliding, the normal points *away* from A and *towards* B (the exact normal direction will depend on the collision shape and nature of the intersection, as we'll see with the other collision types). The intersection distance p is simply the difference between the sphere lengths and the sum of their radii. Lastly, we must consider a *collision point* - the point in space from which are assuming that any forces required to resolve the collision are to be applied from. For a number of reasons, we usually store *two* collision points for a pair of colliding objects, each relative to one of the object's origin. For a sphere, we can determine the collision point of each object by travelling either forwards or backwards along the collision normal by each sphere's radius.

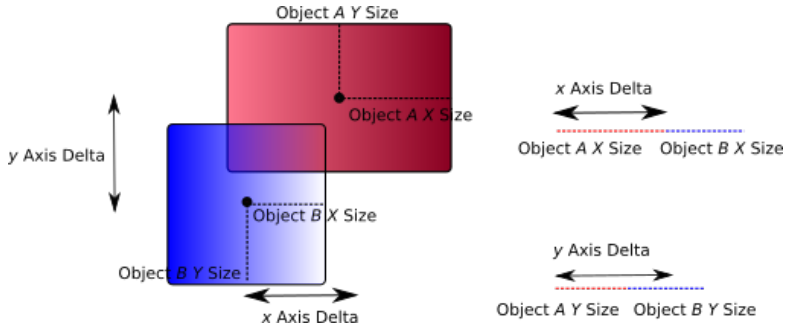
AABB / AABB Collisions

Calculating collisions between axis aligned bounding boxes is a little trickier than with bounding spheres. To determine if they are colliding, we treat them fairly similar to spheres - we calculate the sum of their bounding box dimensions on each axis, and then determine whether the absolute difference in position on each axis is less than the bounding box sum. If all 3 axes are less than their

bounding box sum, then the boxes must be colliding. Here's a few examples to see this in action:

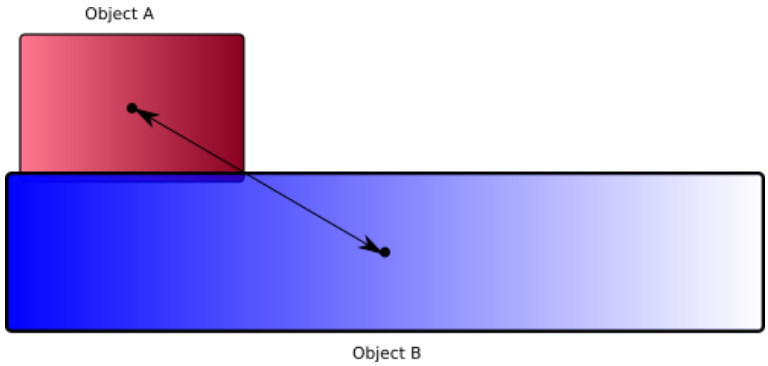


In this first case, we can see that the sum of the x axis box sizes is less than the difference in position on the x axis of the two objects, indicating that maybe there is a collision, here. However, the y axis position difference is greater than the box y axis sizes, so in actuality the boxes are not colliding. Now for case where the objects *are* colliding:



Now the sum of the AABB sizes on each axis is greater than the relative position between the two objects, indicating that the objects must be in contact. While these examples have only shown 2 dimensions, the same rule stands for 3D, too: the difference in position in *all* axes must be less than the sum of the box sizes in that axis, or the volumes are not colliding.

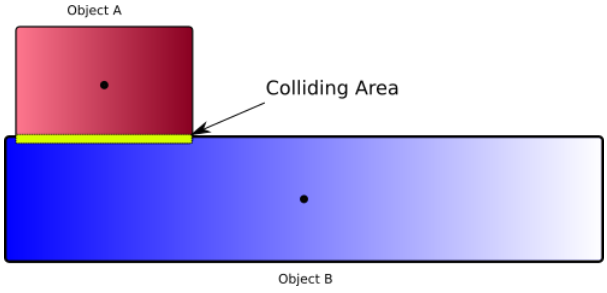
While this test tells us if the boxes are colliding or not, it doesn't really give us the other pieces of the puzzle: the collision normal, the intersection distance, or the intersection points. To get these, we need to do a bit more work. You might initially think that the direction vector between the boxes would be fine for a collision normal, as with bounding spheres, but consider the following example:



Box a should really be pushed straight up slightly to resolve its collision with box b, but the direction vector between their origins actually points towards the left, so box a would be pushed slightly to the left, where it will collide again in the next frame if the simulation has gravity pulling it down back towards b - it would end up in a cycle of constantly being shifted leftwards, gaining energy from seemingly nowhere.

To determine a collision normal for two AABBs, we need to think about the penetration distance on each axis individually, and find which axis overlaps the *least*. Why pick the axis with the least intersection between the objects? Look again at the diagram above - we can visually determine that the two objects are intersecting on the y axis (so object A should really be pushed up, and object B pushed down to 'fix' this collision), but their volumes are actually intersecting by far more on the x axis (object A is in fact completely overlapped by object B on this axis). As we go through each axis to see whether the objects intersect on that axis, we can keep track of by how much - if the shapes are determined to overlap, the axis with the least penetration then gives us the collision normal, and the penetration on that axis provides us with our total penetration distance.

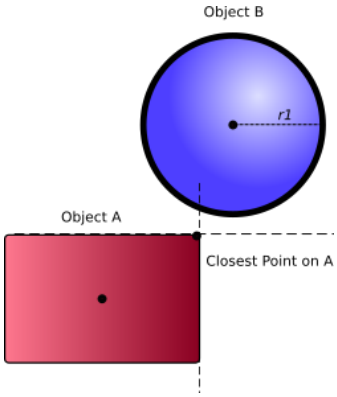
The final thing we need is to define our collision points, so we know where to apply the forces on our objects to move them apart. It can be seen in the example image above that there's not any single point in space for some AABB collisions, unlike with sphere/sphere collisions. A later tutorial will cover exactly how to handle cases where there is an *area* of collision. The case of an AABB collision does give us some more insight into why we calculate two collision points in our collision detection algorithms, rather than just a single point in the world. Consider the example of the two boxes colliding with each other:



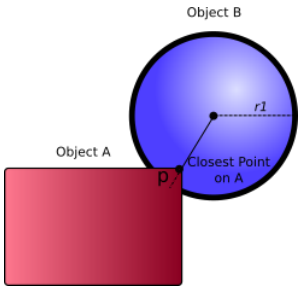
We can see that, assuming we have the y axis as the collision normal, that it would be natural to assume we could pick any point within the shown overlapping area. However, later in the tutorial series we'll be looking at how to add *torque* to objects, and make them spin. If we apply a force at any point along the collision area, then our two AABBs should really start to rotate around. This is *bad* for AABBs, as due to their axis-aligned property, any rotation of the object wouldn't change the volume taken up by the collider. Therefore, for AABBs, we will assume that the collision point of each object is at its local origin - when we look at collision resolution, we'll see how this prevents our AABBs from doing the wrong thing, with the collision normal and penetration distance still allowing us to separate out our objects to maintain the consistency of the simulation.

Box / Sphere Collisions

Testing collisions between boxes and spheres is much the same as with box intersection tests, in that we find the closest point on the box to the sphere, using the **clamp** operation, by limiting the sphere's position to be between the range (box position - box size, bos position + size) on each axis. If that point is greater than the radius away from the sphere's centre, then the objects cannot be colliding! The diagram below should make this clearer:

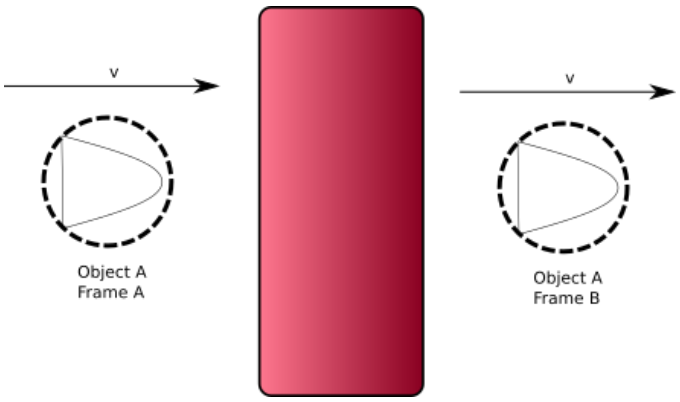


This lets us know about whether the shapes are colliding, but what about our collision point information? The collision normal is the direction vector between the sphere's origin and the closest point on the sphere. The sphere's collision point is then at a point r units along this collision normal, while for the AABB, we would need to pick a point directly along an axis, just as with AABB collisions. The axis to pick would be based on the collision normal - we'd pick whichever axis in the normal had the greatest magnitude. Finally, the penetration distance can be calculated by calculating the distance from the sphere's position to the closest point, and subtracting the sphere's radius:



Collision Detection Considerations

In our simulations, we update our object positions via a series of discrete steps per frame; calculating intersecting objects and separating them out as we go. However, there's a potential problem with this. Consider the following example of a bullet, moving at a velocity of 10m/s, with a bounding sphere for its collision volume:

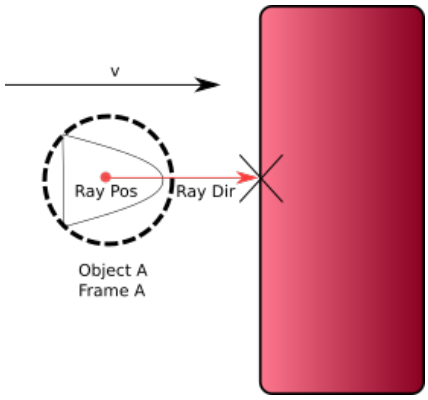


At frame A, the bullet was in front of the wall, and at frame B the bullet was *behind* the wall - This effect is known as *tunneling*, and is caused due to the bullet traveling so fast there was never a frame where it was intersecting the wall, so no collisions could ever be calculated! For certain types of scenario, this is a pretty serious impairment to the accuracy of our physics simulation, so let's investigate some ways to solve it (or avoid it altogether!).

Raycasting

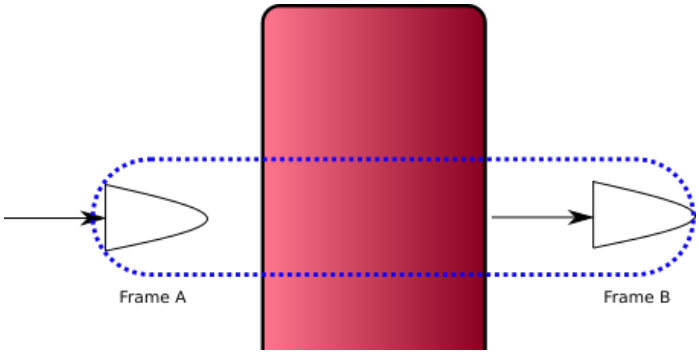
The easiest way to avoid this problem specifically for many physics-based interactions in our games is to not treat them as physics objects at all! Many games such as first person shooters and RTS games don't model bullets as rigid bodies, but simply determine which direction the bullet will go, and then perform a raycast to see what will be hit along this direction. Weapons that perform a raycast for their collision detection rather than rigid body dynamics are often known as *hitscan* weapons - in real life bullets travel in an arc as gravity and wind resistance affects them, but games are usually more about fun than realistic simulation, so an instantaneous, straight line projectile is often fine.

We can extend this concept a little further, too. For something we know will be traveling in a straight line fast (maybe the player picked up a rocket launcher instead of a machine gun?), we could instead raycast from the object, in the direction it is traveling - if the closest object is at a distance of less than the object's velocity away, we can assume it will hit it, and mark the object for collision next frame at point p:

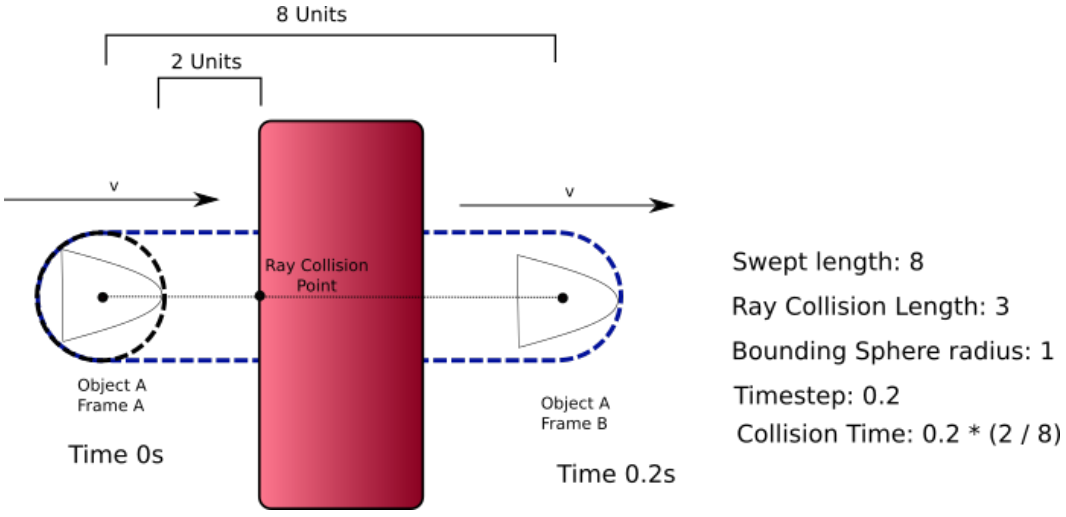


Swept Bounding Volumes

Just performing a raycast is often not enough though - the ray is infinitely thin, but we're trying to determine the collisions between shapes with volume. We must therefor consider how to make changes to the actual collision volume itself when trying to detect high-speed collisions. Consider the bullet going through the wall example from earlier. In both frames A and B, the collision volume of the bullet was not large enough to intersect with that of the wall, but in reality we know that they *must* have collided. To solve this, we could 'stretch' the bounding volume of an object as it moves, so that it encapsulates both its point at time A, and at time B, like so:

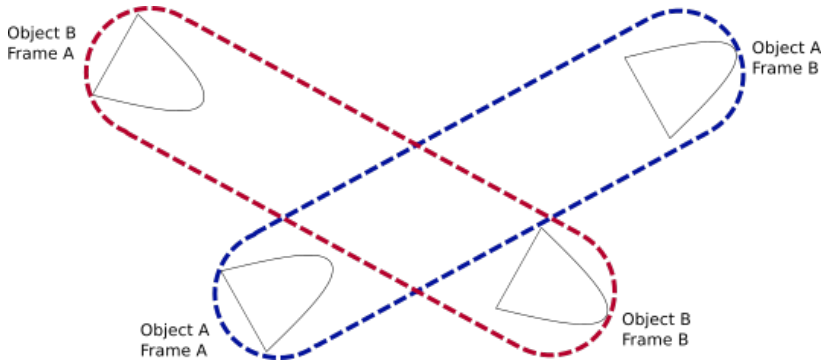


This is known as a *swept* bounding volume. If we then detect a collision with the swept volume, then at some point between frame time A and B, a collision occurred. We can determine approximately *when* using the collision point on to the collided object (in this case, the wall), using a raycast:

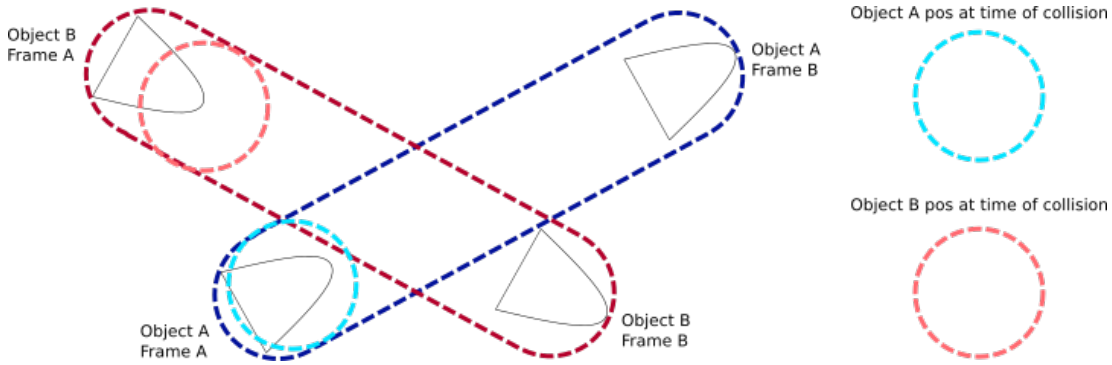


We know the object's position both before and after the collision, so we can form a ray which starts at the 'before' point, and aims in the direction of the 'after' point. We also know the *distance* between those positions, and also the distance between the raycast origin and the intersection point,

the ratio between those two, multiplied by *dt*, will let us know how far in to the physics update the collision occurred. Why would we need the approximate collision time? Wouldn't just moving our bullet to the collision point, and then resolving the collision as normal work? For a collision against a static object it would be fine, but consider this example:

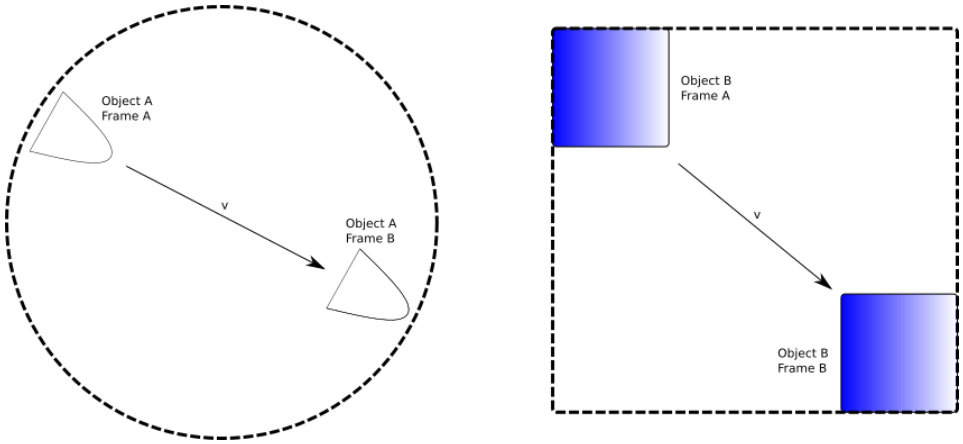


Now we have *two* high speed objects! If we sweep both of their volumes, it appears that there would be a collision, but did they really? To solve this, we need to determine when between frame A and frame B that object A collided with the swept volume of B, and from that time, determine where object B was, and then perform an additional collision detection test at these new positions, using the object's 'true' collision volumes (in our bullet example, we're assuming this is a bounding sphere).

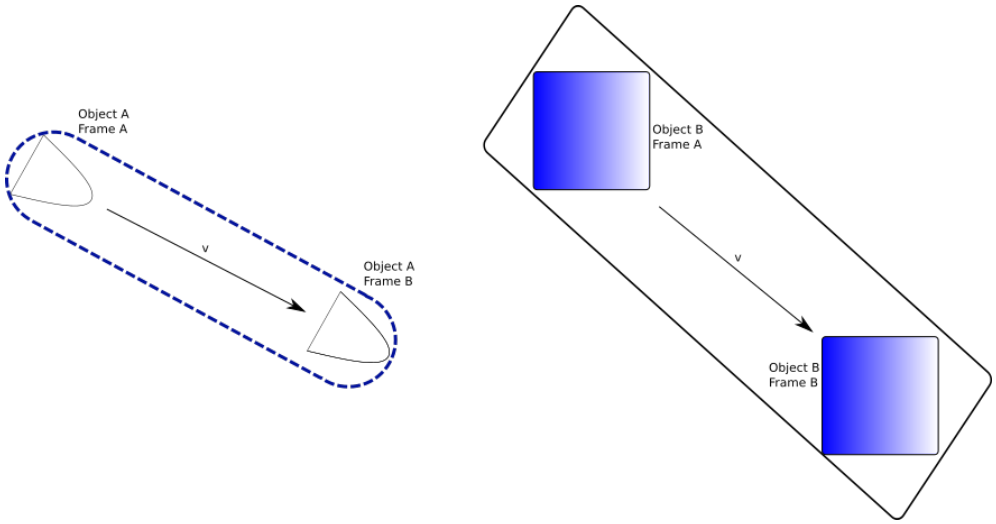


This should be enough to determine with reasonable accuracy whether or not two shapes collide over a time period - remember, a physics systme will run often multiple times per frame, and we aim to run at perhaps 60FPS, meaning that every iteration of our physics system may only be 2 or 3 milliseconds in length, making most swept volumes quite small.

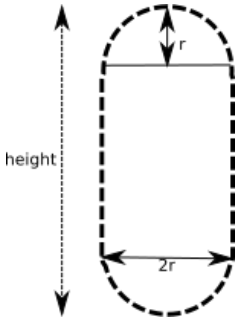
A final note on swept bounding volumes - the method you use to stretch the volume matters a great deal to the accuracy of the simulation. You might first think that for AABBs and spheres, that just stretching them so that they now encompass the end point is enough, but take a look at these examples:



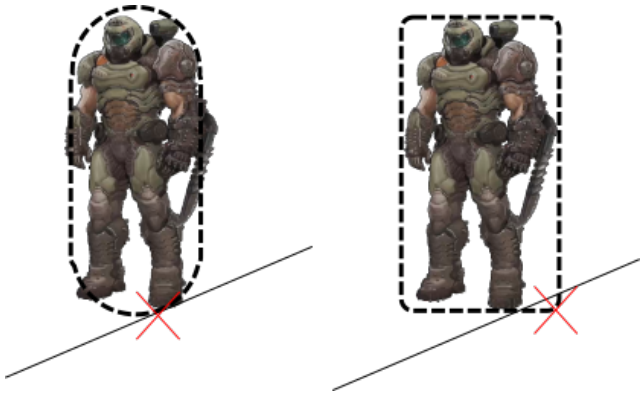
That's a pretty big sphere! While it's certainly expanded to cover the end point, it's also stretched out to take up space that the object will never fill during the course of its movement between frame A and frame B. This could generate false collisions, or at at the very least, cause additional collision tests that take up your frame time computing. Instead, we must be smarter with our swept volumes - an AABB becomes a swept OOBB, and a sphere becomes a swept *capsule*:



A capsule can be thought of as a cylinder with half a sphere on each end, and is usually defined as using a height, and a radius, like so:



Capsules are useful beyond just sweeping volumes. They are often used as the basic collision model of a player character, as the rounded edge provides a more accurate fit against the character's head versus a box, and the rounded bottom can be used as a more accurate representation of where the character may intersecting with ramps and steps:



Detecting collisions in a simulation

We now know the formulas for determining whether various simple shapes are intersecting with each other, and what data we'll later use to resolve this collisions. But we currently don't really know how to determine *when* to *use* these collision detection methods. As it's possible for objects to move around the world, it's therefore possible that any object *could* be colliding with any other object in the world at any given moment in time. Therefore, in code, to detect our collisions and resolve them, we can do something like this:

```
for every object x in game world {
   for every object y in game world {
      if(IsColliding(x,y)) {
         ResolveCollision(x,y)
      }
   }
}
```

Seems simple enough, but there's a problem - it's slow! There's a nested set of **for** loops, each of which has to go over every object in the world; to put it another way, we must do N^2 operations to detect all our collisions. That quickly builds up to *a lot* of collision detection tests! Imagine you were making a racing game where 100 cars competed to travel across the country; even ignoring any collisions with other cars that may be on the road, that's 10,000 collision detection tests that must be performed every frame.

In the above code example, we'll end up testing whether object x is colliding with object y, AND if object y is colliding with object x - as far as our physics interactions go, that's the same test, so really we should avoid repeating it. Even more stupidly, in cases where $x == y$, we'd be testing whether an object is colliding with *itself*! We can adjust our **for** loop to something a bit more sensible, like so:

```
for int x = 0; x < lastObject; ++x {
   for int y = x+1; y < lastObject; ++y) {
      if(IsColliding(x,y)) {
         ResolveCollision(x,y)
      }
   }
}
```

In this case, our inner **for** loop will skip any test combination that's already been done (looking at the **ints** created by the loops, they will now never generate object pair (6,5) or (6,6), but will still form (5,6), for example). The loop itself is still characterised by $O(N^2)$ as the number of potential collisions still increases, but we save some time, and save some potential headaches that might occur if collision resolution was pushing objects multiple times in a frame, and thus potentially adding too much energy into our physics system.

We'll see in a later tutorial that we can use properties about our knowledge of the world to skip collision testing between objects that can't possibly be colliding, but for now we'll just deal with the 'slow' way.

Tutorial Code

To show off collision detection and response, we're going to once again modify the test scene we've been building up. As welll as being able to click on objects to push them around, we'll also be able to receive some feedback on when those objects have collided. By pressing the gravity button, we'll also be able to see the objects be pulled down and bounce onto the floor, rather than just go through it.

CollisionDetection Namespace Changes

To detect collisions between our shapes, we need some new functions! So far we've been introduced to AABBs and spheres, so we'll need at least 3 new functions (to handle AABB vs AABB collisions, sphere vs sphere collisions, and AABB vs sphere collisions). As well as these, we're going to further split up AABB tests, so that we get the true/false collision answer from one function, but the actual collision data from another - this is something that will help us out in a later tutorial. Finally we need a function which takes in a pair of **GameObjects**, and calls the correct intersection function to apply. We also need a struct, to hold the collision information, so that we can resolve the collision correctly in the next tutorial.

These methods to perform all of these operations have been declared within the **CollisionDetection** namespace, but as in the raycasting tutorial, they're currently empty - we'll be filling in each to see how the theory outlined earlier translates to a practical implementation. In these methods, we're going to be filling in a struct we haven't so far come across, **CollisionInfo**, which looks like this:

```
1    struct ContactPoint {
2        Vector3  localA; //where did the collision occur...
3        Vector3  localB; //in the frame of each object!
4        Vector3  normal;
5        float    penetration;
6    };
7    struct CollisionInfo {
8        GameObject* a;
9        GameObject* b;
10
11       ContactPoint point;
12
13       void AddContactPoint(const Vector3& localA,
14           const Vector3& localB, const Vector3& normal, float p){
15           point.localA      = localA;
16           point.localB      = localB;
17           point.normal      = normal;
18           point.penetration = p;
19       }
20   };
```

CollisionDetection namespace additions

It might at first seem strange that we are splitting up the concept of an individual contact point (with its associated positions, normal, and penetration amount), from the overall concept of a collision between a pair of objects, but this is fairly common in physics engines for allowing the creation of what is known as a collision manifold, that represents a full intersecting surface, rather than just a single point.

The first function we'll implement is *ObjectIntersection*; It's currently a defined function that does nothing except return **false**, so we'll need to fill it out with some actual logic. The purpose of this function is to take in two of our **GameObjects**, and determine which of the collision detection functions to use. There's a variety of ways in which this could be implemented (virtual functions, double dispatch, function pointers, and so on), but to make it obvious what is going on, we'll take a direct approach and use **if** statements. We'll start the method by getting pointers to the collision volumes

of our objects (lines 3 and 4), and if one shape doesn't have a volume, return false straight away, as they can't possibly collide with anything (line 7). If the objects can collide, we fill in the start of our **CollisionInfo** struct with the two objects, and get the object's transforms (lines 13 and 14 - we don't strictly need to do this, but it cuts down on repeated calls to the same methods).

```
1  bool CollisionDetection::ObjectIntersection(
2        GameObject* a, GameObject* b, CollisionInfo& collisionInfo) {
3     const CollisionVolume* volA = a->GetBoundingVolume();
4     const CollisionVolume* volB = b->GetBoundingVolume();
5
6     if (!volA || !volB) {
7        return false;
8     }
9
10    collisionInfo.a = a;
11    collisionInfo.b = b;
12
13    const Transform& transformA = a->GetConstTransform();
14    const Transform& transformB = b->GetConstTransform();
```

CollisionDetection::ObjectIntersection Function

Next, we need to send our objects, and the *collisionInfo* struct, off to the correct function to detect the possible collision. To do this, we're going to do a little bit of bitwise operations on the object's types. This is an **enum** (*VolumeType*, defined in *BoundingVolume.h*). As each entry in the *VolumeType* enum is a power of two (so only one bit is set), if we do a bitwise OR of both object's types, we get a bitmask of the type of collision. If after the bitwise OR the type still equates exactly to either AABB (value of 1) or Sphere (value of 2), then it means that *both* shapes must be the same type, and so their ordering doesn't really matter. On line 15 we perform the bitwise OR (with the | symbol), and then use **if**s to either return the result of an AABB intersection, or a Sphere intersection on the objects. In each of these cases, we need to cast the collision volume to the correct type:

```
15    VolumeType pairType=(VolumeType)((int)volA->type|(int)volB->type);
16
17    if (pairType == VolumeType::AABB) {
18       return AABBIntersection((AABBVolume&)*volA, transformA,
19             (AABBVolume&)*volB, transformB, collisionInfo);
20    }
21
22    if (pairType == VolumeType::Sphere) {
23       return SphereIntersection((SphereVolume&)*volA, transformA,
24             (SphereVolume&)*volB, transformB, collisionInfo);
25    }
```

CollisionDetection::ObjectIntersection Function

But what to do if the objects being tested **aren't** of the same type? We could do some bit shifting when we create the *pairType* variable to differentiate between cases where object *b* is a sphere versus when *a* is a sphere and test against that, but for now we'll just handle the two cases we could get so far:

```
26    if (volA->type == VolumeType::AABB &&
27        volB->type == VolumeType::Sphere) {
28       return AABBSphereIntersection((AABBVolume&)*volA, transformA,
29             (SphereVolume&)*volB, transformB, collisionInfo);
30    }
31    if (volA->type == VolumeType::Sphere &&
32        volB->type == VolumeType::AABB) {
33       collisionInfo.a = b;
34       collisionInfo.b = a;
35       return AABBSphereIntersection((AABBVolume&)*volB, transformB,
36             (SphereVolume&)*volA, transformA, collisionInfo);
37    }
38
39    return false;
40 }
```

CollisionDetection::ObjectIntersection Function

Note that we might need to swap the objects of our *collisionInfo* variable around, to match up to the order of the parameters the function expects - *AABBSphereIntersection* always takes in the AABB volume object first, so we flip them in the *collsionInfo* struct to reflect this (lines 33 and 34).

Implementing Sphere-Sphere collision detection

With the selection of a collision detection function completed, we can start implementing the actual detection functions. The easiest to follow is sphere-sphere collisions, so we'll start there. Find the currently empty *SphereIntersection* function within the *CollisionDetection* namespace, and add in the following code:

```
1  bool CollisionDetection::SphereIntersection(
2     const SphereVolume& volumeA, const Transform& worldTransformA,
3     const SphereVolume& volumeB, const Transform& worldTransformB,
4     CollisionInfo& collisionInfo) {
5
6     float radii    =  volumeA.GetRadius() + volumeB.GetRadius();
7     Vector3 delta  =  worldTransformB.GetPosition() -
8                       worldTransformA.GetPosition();
9
10    float deltaLength = Vector::Length(delta);
11
12    if (deltaLength < radii) {
13       float penetration    = (radii - deltaLength);
14       Vector3 normal       = Vector::Normalise(delta);
15       Vector3 localA       =  normal * volumeA.GetRadius();
16       Vector3 localB       = -normal * volumeB.GetRadius();
17
18       collisionInfo.AddContactPoint(localA,localB,normal,penetration);
19       return true;//we're colliding!
20    }
21    return false;
22 }
```

CollisionDetection::SphereIntersection Function

This matches up pretty well to the theory outlined earlier. We get the sum of the two object's radii (line 6), then work out the distance between the two object's positions (line 7 and 10). If the distance between them is less than the sum of their radii, they must be intersecting (line 12), and so we can work out the penetration distance (the sum of the radii, minus the distance between the objects), the collision normal (the direction vector between the two objects), and the collision points - in this

case we're travelling along the collision normal by object A's radius, or backwards along the collision normal by object B's radius, to work out the collision points relative to the respective object's centres. If the distance is *larger* than the sum of the radii, we can't be colliding, and so return **false** (line 21).

Implementing AABB-Sphere collision detection

Sphere-sphere is pretty easy, but going through it allowed us to see how the *collisionInfo* **struct** should be filled in. Now we can make things a little more advanced, and look at AABB-sphere collision detection, using the currently empty *AABBSphereIntersection* function. As described earlier, the core of AABB-sphere intersection is to find the closest point on the box to the sphere's location. We can do this by simply clamping the relative position of the sphere, to limit it to the box size - for any axis where the position is greater than the box size, we 'lock' it to the box size. We can do that in code by getting the box dimensions (line 5) and the relative position of the sphere to the box (line 7), and then forming a new position in space, *closestPointOnBox* (line 10). To clamp a value to a range, we can use the *Clamp* function inside the **Maths** namespace, which takes in a value *a*, along with a minimum and maximum, and returns a modified *a* that is limited to the range passed in (for example *Clamp*(10,3,12) will try to limit the value 10 to be between 3 and 12, and so will return 10, as it lies within the range, but *Clamp*(20,3,12) will return 12, as 20 is beyond the maximum value allowed).

```
1  bool CollisionDetection::AABBSphereIntersection(
2      const AABBVolume& volumeA  , const Transform& worldTransformA,
3      const SphereVolume& volumeB, const Transform& worldTransformB,
4      CollisionInfo& collisionInfo) {
5      Vector3 boxSize = volumeA.GetHalfDimensions();
6
7      Vector3 delta  =  worldTransformB.GetPosition() -
8                        worldTransformA.GetPosition();
9
10     Vector3 closestPointOnBox = Vector::Clamp(delta,-boxSize,boxSize);
```

CollisionDetection::AABBSphereIntersection Function

Once we have the sphere's closest point on the box, we can determine how far away the sphere is from this point, by subtracting this point from the sphere's relative position (line 11). If this point lies at a distance less than the sphere's radius (line 14), then it must be colliding, and we can fill in our *collisionInfo* struct. The collision point for our sphere (line 19) will then lie *radius* units backwards along the collision normal (remember, our normals are pointing towards object B), whereas our AABB will as usual be assumed to be colliding at its relative position (line 18). The penetration distance will be the sphere's radius minus the distance from the sphere to the closest point on the box (line 16).

```
11     Vector3 localPoint    = delta - closestPointOnBox;
12     float distance        = Vector::Length(localPoint);
13
14     if (distance < volumeB.GetRadius()) {//yes, we're colliding!
15        Vector3  collisionNormal = Vector::Normalise(localPoint);
16        float    penetration     = (volumeB.GetRadius() - distance);
17
18        Vector3 localA = Vector3();
19        Vector3 localB = -collisionNormal * volumeB.GetRadius();
20
21        collisionInfo.AddContactPoint(localA, localB,
22              collisionNormal, penetration);
23        return true;
24     }
25     return false;
26 }
```

CollisionDetection::AABBSphereIntersection Function

Implementing AABB-AABB collision detection

The AABB-AABB collision detection has been split up into two functions - one to tell us whether the objects are colliding, and one to fill in the collision information. The easier to understand is the testing function, so we'll start off by implementing it inside the *AABBTest* function:

```
1  bool CollisionDetection::AABBTest(
2     const Vector3& posA, const Vector3& posB,
3     const Vector3& halfSizeA, const Vector3& halfSizeB) {
4     Vector3 delta     = posB - posA;
5     Vector3 totalSize = halfSizeA + halfSizeB;
6
7     if (abs(delta.x) < totalSize.x &&
8         abs(delta.y) < totalSize.y &&
9         abs(delta.z) < totalSize.z) {
10       return true;
11    }
12    return false;
13 }
```

CollisionDetection::AABBTest Function

It's logic follows the earlier description of AABB intersection - we're looking to see if the distance between the object's on each axis is less than the sum of the box sizes on that axis, and only if that is true for all axes should we consider the objects intersecting. On line 4 we work out the distance on each axis, and line 5 we calculate the sum of the box sizes, and then on line 7 we calculate whether the distances are less than the box sizes. We need to use the abs function to get the absolute value of the relative position, as the box A might be to the left of box B, or to the right - it doesn't matter, all we care about is whether the difference is position is greater than the box dimensions.

After the *AABBTest* function, you should be able to also find an empty *AABBIntersection* function, too. We'll start adding some code to it to make it actually do something:

```
1  bool CollisionDetection::AABBIntersection(
2     const AABBVolume& volumeA, const Transform& worldTransformA,
3     const AABBVolume& volumeB, const Transform& worldTransformB,
4     CollisionInfo& collisionInfo) {
5
6     Vector3 boxAPos = worldTransformA.GetPosition();
7     Vector3 boxBPos = worldTransformB.GetPosition();
8
9     Vector3 boxASize  = volumeA.GetHalfDimensions();
10    Vector3 boxBSize  = volumeB.GetHalfDimensions();
11
12    bool overlap = AABBTest(boxAPos, boxBPos, boxASize, boxBSize);
```

CollisionDetection::AABBIntersection Function

It's a pretty simple start - we extract the positions and sizes of the AABBs, and feed them into the *AABBTest* function to determine whether they are overlapping. If they are overlapping, things get a bit more interesting, as the next bit of code to add to this new function demonstrates:

```
13    if (overlap) {
14       static const Vector3 faces[6] =
15       {
16          Vector3(-1,  0,  0), Vector3( 1,  0,  0),
17          Vector3( 0, -1,  0), Vector3( 0,  1,  0),
18          Vector3( 0,  0, -1), Vector3( 0,  0,  1),
19       };
20
21       Vector3 maxA = boxAPos + boxASize;
22       Vector3 minA = boxAPos - boxASize;
23
24       Vector3 maxB = boxBPos + boxBSize;
25       Vector3 minB = boxBPos - boxBSize;
26
27       float distances[6] =
28       {
29          (maxB.x - minA.x),//distance of box 'b' to 'left' of 'a'.
30          (maxA.x - minB.x),//distance of box 'b' to 'right' of 'a'.
31          (maxB.y - minA.y),//distance of box 'b' to 'bottom' of 'a'.
32          (maxA.y - minB.y),//distance of box 'b' to 'top' of 'a'.
33          (maxB.z - minA.z),//distance of box 'b' to 'far' of 'a'.
34          (maxA.z - minB.z) //distance of box 'b' to 'near' of 'a'.
35       };
36       float penetration = FLT_MAX;
37       Vector3 bestAxis;
38
39       for (int i = 0; i < 6; i++)
40       {
41          if (distances[i] < penetration) {
42             penetration = distances[i];
43             bestAxis    = faces[i];
44          }
45       }
46       collisionInfo.AddContactPoint(Vector3(), Vector3(),
47          bestAxis, penetration);
48       return true;
49    }
50    return false;
51 }
```

CollisionDetection::AABBIntersection Function

As the theory outlined earlier, we need the axis of *minimum* penetration between the two objects. To determine this, and help determine the collision normal, there's quite a lot to go through. From a box's world position, we can determine the minimum and maximum position for each axis by either adding or subtracting the box's size. On lines 21 to 25 we calculate this, and use it to calculate the amount of overlap on each axis - the maximum extent of object *b*, minus the minimum extent of object a on each axis gives us the amount of overlap (to put it another way, we're seeing how far into box *a*, that box *b*'s maximum position extends). We then iterate through these penetration extents (line 39), and work out which is the smallest value, storing the penetration, and the axis (from the static *faces* array, which is arranged to match up to the *distances* array) every time we find a smaller value - this is why the *penetration* **float** starts with the maximum value a float can hold, rather than starting at 0.

Once we've iterated through all the possible penetration amounts and found the best, we can then start to build the collision info using the calculated penetration, and the *bestAxis* variable for the collision normal. As we're checking for AABBs and we don't want them to twist later on, we set their relative collision points both to the origin (line 46) - you'll see why this is important in a later tutorial!

PhysicsSystem Class Changes

Nearly all of the complex code was added in the **CollisionDetection** namespace, so there's not much to do. First off, we need to create a way of determining whether collisions are occuring each frame, as outlined earlier. To do this, the **PhysicsSystem** class has a *BasicCollisionDetection* method, but once again it has been left empty. Add in the following code:

```
1  void PhysicsSystem::BasicCollisionDetection() {
2      std::vector<GameObject*>::const_iterator first;
3      std::vector<GameObject*>::const_iterator last;
4      gameWorld.GetObjectIterators(first, last);
5
6      for (auto i = first; i != last; ++i) {
7          if ((*i)->GetPhysicsObject() == nullptr) {
8              continue;
9          }
10         for (auto j = i+1; j != last; ++j) {
11             if ((*j)->GetPhysicsObject() == nullptr) {
12                 continue;
13             }
14             CollisionDetection::CollisionInfo info;
15             if (CollisionDetection::ObjectIntersection(*i, *j, info)) {
16                 std::cout << "Collision between " << (*i)->GetName()
17                           << " and " << (*j)->GetName() << std::endl;
18                 info.framesLeft = numCollisionFrames;
19                 allCollisions.insert(info);
20             }
21         }
22     }
23 }
```

PhysicsSystem::BasicCollisionDetection Method

It's not so different to how we did raycasting, except now instead of iterating over all of the objects once (via the iterators we get from the **GameWorld** class on lines 2-4), we need to go over the objects twice. The important thing to get right here is line 10 - we are starting the inner **for** loop at one element past the outer **for** loop, so that we don't resolve identical collisions multiple times in a frame. As we *might* be inserting objects into the game world that don't have physics objects, we also need to skip over these (a little bush or flower may have a graphical representation, but no physical representation, for instance). For now, we're going to just output that a collision has taken place into the console, but we'll soon be adding some code to actually resolve the collision to this method, too. The **if** statement on line 15 calls our new *VolumeIntersection* method, which will return **true** if a collision has taken place, and if so will fill in the *info* **struct** with the appropriate data for later use.

There's one last thing we do, which is to insert the successfully detected collision into an **STL::List**, which will let us keep track of objects that are colliding - later on we might need to use this information across multiple frames.

As an example of why we might want to keep a list of colliding objects, we're going to look at another currently empty function, *UpdateCollisionList*. This gets called by the *PhysicsSystem* every frame, and should have the following code in it:

```
void PhysicsSystem::UpdateCollisionList() {
    for (std::set<CollisionDetection::CollisionInfo>::iterator i =
        allCollisions.begin(); i != allCollisions.end(); ) {
        if ((*i).framesLeft == numCollisionFrames) {
            i->a->OnCollisionBegin(i->b);
            i->b->OnCollisionBegin(i->a);
        }
        (*i).framesLeft = (*i).framesLeft - 1;
        if ((*i).framesLeft < 0) {
            i->a->OnCollisionEnd(i->b);
            i->b->OnCollisionEnd(i->a);
            i = allCollisions.erase(i);
        }
        else {
            ++i;
        }
    }
}
```

PhysicsSystem::UpdateCollisionList Method

What is it doing? Games without any feedback from the physics system as to which collisions have occurred aren't very exciting - we need to know when a rocket has hit a player to reduce their health, or when they have dropped the companion cube into the furnace in *Portal*. So in this function, we're going to go through the list of collisions we filled up in the *BasicCollisionDetection* method, and if they're brand new to the list this frame, call a **virtual** function on the **GameObjects** contained within the **CollisionInfo** struct - by default this won't do anything, but it allows for subclasses of the **GameObject** to implement game specific logic by overriding the *OnCollisionBegin* method. Sometimes, we might even need to know when an object has *stopped* colliding with something (maybe the player has wandered into some lava, and should lose health until they leave), so in this case we can detect it and call another **virtual** function *OnCollisionEnd*. Otherwise, we reduce a counter on each collision pair, and if necessary, update the iterator or remove the pair from the list - later on we'll see why we need to keep objects around for multiple frames. Every time the same object pair collides, this counter gets reset (by reinserting it in the *BasicCollisionDetection* method), so only if a pair of objects hasn't been colliding with each other for a while will the *OnCollisionEnd* method be called.

That's everything for now! If we use the mouse pointer to push objects together like we did in previous tutorials, the objects will still overlap (we aren't *resolving* the collision, merely *detecting* it), but we should be able to see in the console that a collision is occurring, allowing us to test that our spheres and boxes return the correct collision state.

Conclusion

Collision detection is an important part of any physics engine, as without them we can't determine how objects should interact with each other - objects will fall through the floor due to gravity, and powerups will be impossible to pick up! Being able to efficiently detect that collisions are occurring between the shapes in our world is therefore incredibly important, and thus tutorial has shown enough to get us started in building up all of the collision detection we need. In the next tutorial, we'll take a look at how to resolve the collisions we're detecting, allowing our objects to bounce around the world, and come to rest on the ground.

Further Work

1) While the tutorial text has discussed capsules, and how they might be useful, we haven't yet seen any code to implement them. Try making a **CapsuleCollider** class that inherits from **CollisionVolume**, and which uses a new *VolumeType* **enum** to identify it. Remember that we need a height variable, and a radius to represent our shape - treating the ends as spheres and the middle like an

AABB is a good place to start, but you may also wish to think about projecting a collision volume's position onto a plane.

2) If you *do* add capsules, you might then want to think about adding the ability to *sweep* a bounding volume - just try it with spheres for now, and test it by firing spheres at high speed towards a very thin wall; you should be able to create a scenario where the spheres will sometimes bounce through, unless swept spheres are used.