Collision Geometry

In modern 3D games, it's very common for human characters to be composed of 15,000+ polygons. When a game needs to determine whether or not two characters collide, it wouldn't be terribly efficient to check against all these triangles. For this reason, most games utilize simplified **collision geometry**, such as spheres and boxes, to test for collisions. This collision geometry isn't drawn to the screen, but only used to check for collisions efficiently. In this section we cover some of the most common collision geometries used in games.

It's also worth noting that it's not uncommon for games to have multiple levels of collision geometry per object. This way, a simple collision check (such as with spheres) can be performed first to check whether there's any possibility whatsoever that two objects collided. If the simple collision check says there might be a collision, we can then perform calculations with more complex collision geometry.

Bounding Sphere

The simplest type of collision geometry is the bounding sphere (or in 2D games, a bounding circle). A sphere can be defined with only two variables—a vector representing the center of the sphere, and a scalar for the radius of the sphere:

```
class BoundingSphere
   Vector3 center
   float radius
end
```

As illustrated in Figure 7.2, certain objects, such as an asteroid, fit well with bounding spheres. But other types of objects, including humanoid characters, end up with a lot of empty space between the edge of the character and the sphere. This means that for many types of objects, bounding spheres can have a large number of **false positives**. This happens when two game objects have intersecting collision geometry, yet the game objects themselves are not actually intersecting. False positives can be really frustrating for the player, because it will be apparent if the two game objects aren't actually intersecting when a collision is detected.

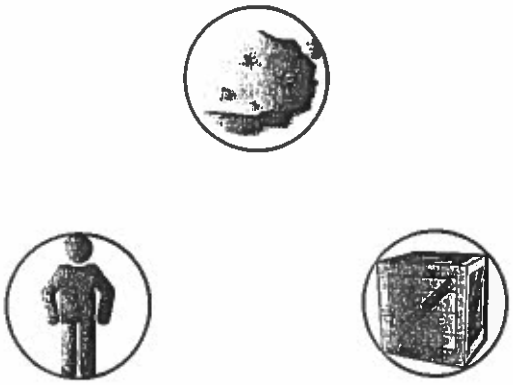

Figure 7.2 Bounding spheres for different objects.

Because of the relative inaccuracy of bounding spheres for most game objects, it usually is not appropriate to use bounding spheres as the only collision geometry. But the big advantage of bounding spheres is that instantaneous collision detection between two spheres is very simple, so this representation may be a great candidate for a first-line collision check.

Axis-Aligned Bounding Box

For a 2D game, an **axis-aligned bounding box** (or AABB) is a rectangle where every edge is parallel to either the x-axis or y-axis. Similarly, in 3D an AABB is a rectangular prism where every side of the prism is parallel to one of the coordinate axis planes. In both 2D and 3D, an AABB can be represented by two points: a minimum and a maximum point. In 2D, the minimum corresponds to the bottom-left point whereas the maximum corresponds to the top-right point.

```
class AABB2D
   Vector2 min
   Vector2 max
end
```

Because an AABB must keep its sides parallel to the coordinate axes, if an object is rotated the AABB may have to stretch, as demonstrated in Figure 7.3. But for 3D games, humanoid characters will typically only rotate about the up axis, and with these types of rotations the bounding box for the character may not change at all. Because of this, it's very common to use AABBs as a basic form of collision representation for humanoid characters, especially because collision detection between AABBs, as with spheres, is not particularly expensive.

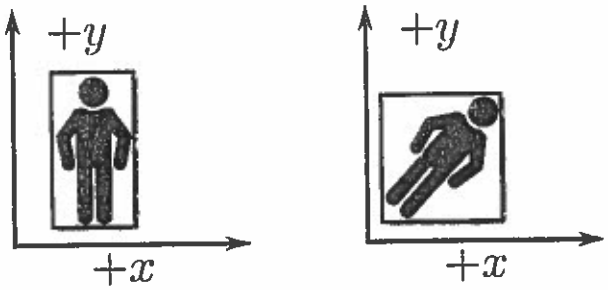

Figure 7.3 Axis-aligned bounding boxes for different orientations of a character.

Collision Detection

Now that we've covered some of the most common types of collision geometries used in games, we can take a look at actually testing for intersections between said objects. Any game that needs to respond to two objects colliding with each other must use some type of collision detection algorithm. Although some might find the amount of math in this section a bit off-putting, know that it is absolutely something that is used in a large number of games. Therefore, it's very important to understand this math—it's not covered just for the sake of being covered. This section won't cover every permutation of every geometry colliding with every other geometry, but it does go over the most common ones you might use.

Sphere versus Sphere Intersection

Two spheres intersect if the distance between their center points is less than the sum of their radii, as shown in Figure 7.5. However, computing a distance requires the use of a square root, so to avoid the square root, it is common instead to check the distance *squared* against the sum of the radii *squared*.

The algorithm for this is only a couple of lines of code, as shown in Listing 7.1. It's also extremely efficient, which is what makes using spheres a very popular basic collision detection option.

Listing 7.1 Sphere versus Sphere Intersection

```
function SphereIntersection(BoundingSphere a, BoundingSphere b)
   // Construct a vector between centers, and get length squared
   Vector3 centerVector = b.center - a.center
   // Recall that the length squared of v is the same as v dot v
```

```
    float distSquared = DotProduct(centerVector, centerVector)

    // Is distSquared < sum of radii squared?
    if distSquared < ((a.radius + b.radius) * (a.radius + b.radius))
        return true
    else
        return false
    end
end
```

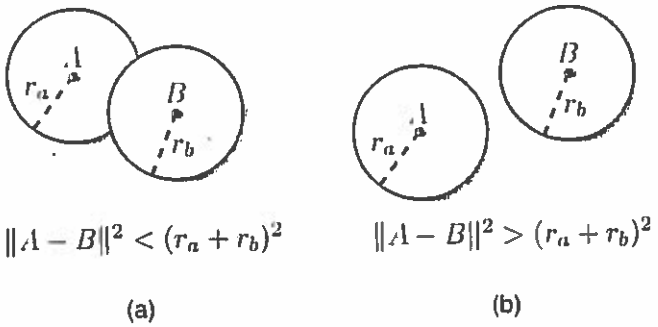

Figure 7.5 Two spheres intersect (a) and do not intersect (b).

AABB versus AABB Intersection

As with spheres, AABB intersection is not very expensive, even for 3D games. It's easier to visualize the algorithm with 2D AABBs, though, so that's what is covered here.

When checking for intersection between two 2D AABBs, rather than trying to test the cases where the two AABBs do intersect, it's easier to test the four cases where two AABBs definitely *cannot* intersect. These four cases are illustrated in Figure 7.6.

If any of the four cases are true, it means the AABBs do not intersect, and therefore the intersection function should return false. This means that the return statement should be the logical negation of the four comparison statements, as in Listing 7.2.

Listing 7.2 AABB versus AABB Intersection

```
function AABBIntersection(AABB2D a, AABB2D b)
    bool test = (a.max.x < b.min.x) || (b.max.x < a.min.x) ||
                (a.max.y < b.min.y) || (b.max.y < a.min.y)

    return !test
end
```

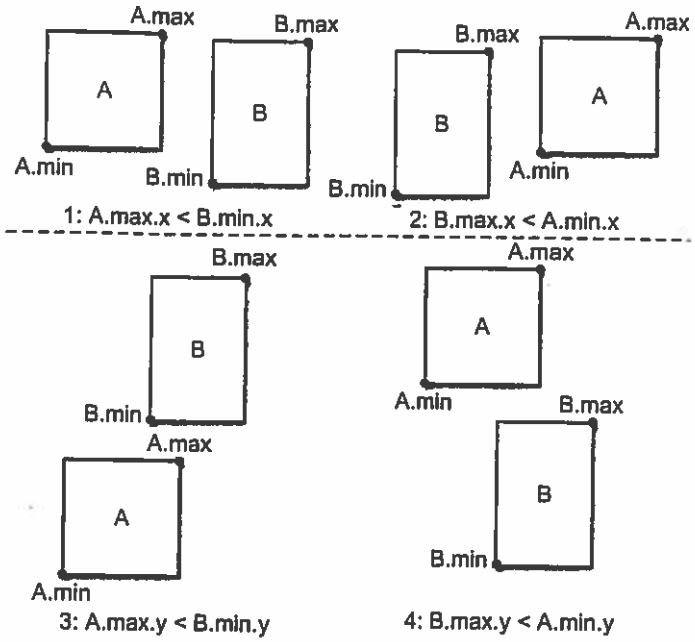

Figure 7.6 The four cases where two 2D AABBs definitely *cannot* intersect.

It's worth mentioning that AABB-AABB intersection is actually a very special application of an algorithm known as the separating axis theorem. In the general form, this theorem can actually be used on any type of convex polygon, though the details of it are beyond the scope of this book.

Collision Response

We can use any of the preceding algorithms to determine whether or not two objects did, in fact, collide. But once they do collide, what should the game do? This is the **response** to the collision. In some instances that response might be very simple: One or both objects may die or be otherwise removed from the world. A slightly more complex response might be something like a missile reducing the health of an enemy on collision.

But what if the two objects need to bounce off of each other, such as two asteroids colliding in *Asteroids*? A tempting solution might be to simply negate the velocity of the asteroids upon collision. But a couple major issues crop up with this approach. One issue is the asteroids getting stuck. Suppose there are two asteroids that intersect on a particular frame, which results in their velocities being negated. But what if they're travelling so slow that they still intersect on the next frame? Well, their velocities will get negated again and again, frame after frame, and they will be stuck in place. This is shown in Figure 7.13(a).

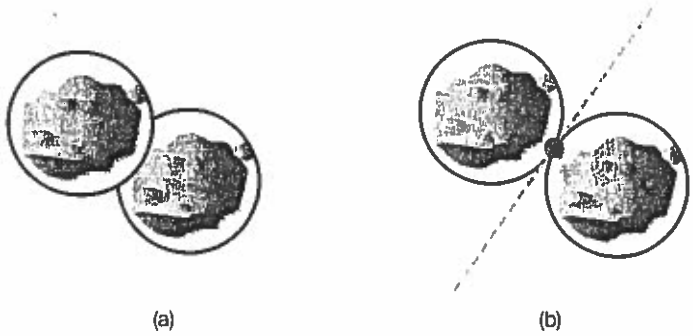

Figure 7.13 Two asteroids get stuck (a), and the plane tangential to the point of intersection (b).

To solve this first issue, we need to find the precise point in time that the asteroids intersect, even if it occurs between frames. Because the asteroids use bounding spheres, we can use swept sphere intersection to find the exact time of intersection. Once we find the time of intersection, we must roll back the position of the two spheres to that point in time. We can then apply whatever change to the velocity we want (in our naïve case, negate them) and then complete the rest of the time step with the new velocity.

However, there's still a big problem with our collision response: Negating the velocities is not the correct behavior. To see why this is the case, imagine you have two asteroids travelling in the same direction, one in front of the other. The front asteroid is travelling slower than the back asteroid, so eventually the back asteroid will catch up. This means when the two asteroids intersect, they will both suddenly start travelling in the opposite direction, which definitely is not correct.

So rather than negating the velocity, we actually want to reflect the velocity vector based on the normal of the plane of intersection. This uses the vector reflection concept discussed in Chapter 3. If an asteroid were colliding with a wall, it would be easy enough to figure out the plane of intersection because we can make a plane out of the wall. However, in the case of two spheres touching at precisely one point, we need to find the plane that's tangential to the point of intersection, as in Figure 7.13(b).

To construct this tangential plane, we first need to determine the point of intersection. This can actually be done using linear interpolation. If we have two spheres that are touching at one point, the point ends up being somewhere on the line segment between the center of the two spheres. Where it is on that line segment depends on the radii of the spheres. If we have two instances of `BoundingSphere` `A` and `B` that are intersecting at precisely one point, we can calculate the point of intersection using linear interpolation:

```
Vector3 pointOfIntersection = Lerp(A.position, B.position,
                                   A.radius / (A.radius + B.radius));
```

It turns out the normal of this tangential plane is simply the normalized vector from the center of one sphere to the center of the other. With both a point on the plane and a normal, we can then create the plane that's tangential at the point of intersection. Although if our collision response is merely a reflection of the velocity, we only really need to compute the normal of the plane.

With this reflected velocity, our asteroid collision will look much better, though it still will seem weird because the asteroids will reflect and continue at the same speed as before. In reality, when two objects collide there is a **coefficient of restitution**, or a ratio between the speed after and before a collision:

$$C_R = \frac{\text{Relative speed after collision}}{\text{Relative speed before collision}}$$

In an **elastic** collision ($C_R > 1$), the relative speed afterward is actually higher than the relative speed beforehand. On the other hand, an **inelastic** collision ($C_R < 1$) is one where the relative speed becomes lower. In the case of asteroids, we likely want an inelastic collision, unless they are magical asteroids.

A further consideration is angular dynamics, which are briefly covered at the end of the chapter. But hopefully this section gave you some idea on how to implement a more believable collision response.

Optimizing Collisions

All of the collision detection algorithms we covered allow for testing between pairs of objects to see if they collide. An optimization question that might come up is what can be done if there is a large number of objects to test against? Suppose there are 10,000 objects in the world, and we want to check whether the player collides with any of them. A naïve solution would perform 10,000 collision checks: one between the player and each of the objects. That's not terribly efficient, especially given that the vast majority of the objects are going to be far away.

So what games must do is try to partition the world in some way such that the player only needs to test against a subset of the objects. One such type of partitioning for a 2D game is called a **quadtree**, because the world is recursively split into quarters until each node in the tree only references one particular object, as shown in Figure 7.14.

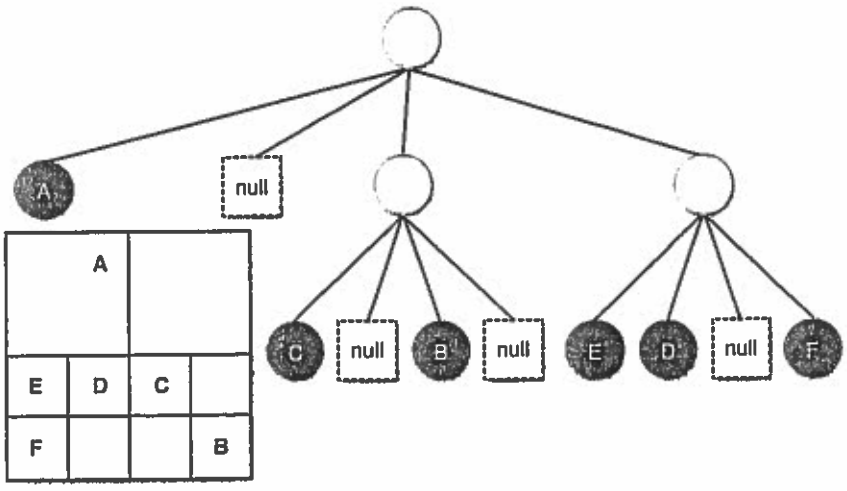

Figure 7.14 A quadtree, where letters represent objects in the world.

When it's time to do the collision check, the code would first check which of the original quarters intersect against the player, potentially eliminating three fourths of the objects instantly. Then a recursive algorithm could continue testing until it finds all of the nodes that potentially intersect with the player. Once there's only a handful of objects left, they could each be tested against the player using the normal collision checks.

A quadtree isn't the only partitioning method available; there are others, including binary spatial portioning (BSP) and octrees (which are the 3D version of quadtrees). Most of the algorithms partition the world spatially, but others partition it with some other grouping heuristic. Partitioning algorithms could easily be an entire chapter in a book, and it turns out there are multiple chapters on this topic in the aforementioned *Real-time Collision Detection*.

Physics-Based Movement

If a game requires objects to move around in the world, some form of physics will be used to simulate the movement. Newtonian physics (also known as classical mechanics) was formulated by Isaac Newton, among others, in the seventeenth century. The vast majority of games utilize Newtonian physics, because it is an excellent model so long as the objects are not moving close to the speed of light. Newtonian physics has several different components, but this section focuses on the most basic one: **linear mechanics**, which is movement without any rotational forces applied.

I want to forewarn you before going further into this topic that classical mechanics is an extremely comprehensive topic—that's why there is an entire college-level class built around it. Of course, I can't fit everything within the context of this book (and honestly, there are enough physics textbooks out there already). So I had to be very selective of which topics I covered—I wanted to talk about topics that had a high change of being included in a game written by someone who's in the target audience for this book.

Linear Mechanics Overview

The two cornerstones of linear mechanics are force and mass. **Force** is an influence that can cause an object to move. Force has a magnitude and a direction, and therefore it is represented by a vector. **Mass** is a scalar that represents the quantity of matter contained in an object. For mechanics, the primary relevant property is that the higher the mass, the more difficult it is to move the object.

If enough force is applied to an object, in theory it would eventually start accelerating. This idea is encapsulated by Newton's second law of motion:

$$F = m \cdot a$$

Here, F is force, m is mass, and a is the acceleration. Because force is equal to mass times acceleration, it's also true that acceleration is force divided by mass. Given a force, this equation is what a game will use in order to calculate the acceleration.

Typically, we will want to represent the acceleration as a function over time, $a(t)$. Now acceleration understandably has a relationship between velocity and position. That relationship is that the derivative of the position function ($r(t)$) is the velocity function ($v(t)$) and the derivative of the velocity function is the acceleration function. Here it is in symbolic form:

$$v(t) = \frac{dr}{dt}$$
$$a(t) = \frac{dv}{dt}$$

But this formulation is not terribly valuable for a game. In a game, we want to be able to apply a force to an object and from that force determine the acceleration over time. Once we have the acceleration, we then want to determine the velocity over time. Finally, with the velocity in

hand, we can then determine the position of the object. All of these are need to be applied over the current time step.

So in other words, what we need is the opposite of the derivative, which you might recall is the integral. But it's not just any type of integration we're interested in. You are likely most familiar with symbolic integration, which looks something like this:

$$\int \cos(x)dx = \sin(x) + C$$

But in a game, symbolic integration is not going to be very useful, because it's not like we are going to have a symbolic equation to integrate in the first place. Rather, on an arbitrary frame we want to apply the acceleration over the time step to get the velocity and the position. This means using **numeric integration**, which is an *approximation* of the symbolic integral applied over a specific time step. If you ever were required to calculate the area under a curve using the midpoint or trapezoidal rule, you calculated a numeric integral. This section covers a few of the most common types of numeric integration used in games.

Issues with Variable Time Steps

Before we cover any of the numeric integration methods, it's important to discuss one big gotcha for any physics-based movement. Once you are using numeric integration, you more or less *cannot* use variable time steps. That's because the accuracy of numeric integration is wholly dependent on the size of the time step. The smaller the time step, the more accurate the approximation.

This means that if the time step changes from frame to frame, the accuracy of the approximation would also change from frame to frame. If the accuracy changes, the behavior will also change in very noticeable ways. Imagine you are playing a game where the character can jump, as in *Super Mario Bros.* You're playing the game, and Mario is jumping at a constant speed. Suddenly, the frame rate drops low and you notice that Mario can jump much further. The reason this happens is that the percent error in numeric integration gets higher with a lower frame rate, so the jump arc becomes more exaggerated. This is demonstrated with the jumping character in Figure 7.15. This means a player on a slower machine will be able to jump further than a player on fast machine.

For this reason, any game that uses physics to calculate the position should not use a variable time step for physics calculations. It's certainly possible to code a game where the physics is running at a different time step than the rest of the game, but that's a little bit more advanced. For now, you could instead utilize the frame rate limiting approach outlined in Chapter 1, "Game Programming Overview."

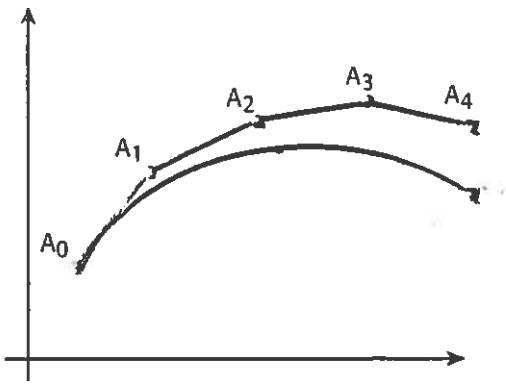

Figure 7.15 Different jump arcs caused by different sized time steps.

Calculating the Force

Numeric integration will allow us to go from acceleration to velocity, and then velocity to position. But in order to compute the acceleration, we need force and mass. There are multiple types of forces to consider. Some forces, such as gravity, are constantly applied to an object. Other forces may instead be **impulses**, or forces applied for a single frame.

For example, a jump might be caused first by applying an impulse force to begin the jump. But once the jump starts, it's the force of gravity that'll bring the character back down to the ground. Because multiple forces can be acting on an object at the same time, the most common approach in a game is to sum up all the force vectors affecting an object and divide this by the mass to determine the current acceleration:

```
acceleration = sumOfForces / mass
```

Euler and Semi-Implicit Euler Integration

The simplest type of numeric integration is **Euler integration**, which is named after the famed Swiss mathematician. In Euler integration, first the new position is calculated by taking the old position and adding to it the old velocity multiplied by the time step. Then the velocity is similarly calculated using the acceleration. A simple physics object that uses this in its update function is shown in Listing 7.7

Listing 7.7 Euler Integration in Physics Object

```
class PhysicsObject
    // List of all the force vectors active on this object
    List forces
    Vector2 acceleration, velocity, position
    float mass

    function Update(float deltaTime)
        Vector2 sumOfForces = sum of forces in forces
        acceleration = sumOfForces / mass

        // Euler Integration
        position += velocity * deltaTime
        velocity += acceleration * deltaTime
    end
end
```

Although Euler integration is very simple, it does not really exhibit very much accuracy. One big issue is that the position calculation is using the old velocity, not the new velocity after the time step. This results in a propagation of error that causes the approximation to diverge further and further as time continues.

A simple modification to basic Euler integration is to swap the order of the position and velocity calculations. What this means is that the position is now updating based on the new velocity, not the old velocity. This is **semi-implicit Euler integration**, and it ends up being reasonably more stable, to the point that respectable physics engines such as Box2D utilize it. However, if we want further accuracy we have to explore more complex numerical integration methods.