HOUR 9

Collision

What You'll Learn in This Hour:

- The basics of rigidbodies
- How to use colliders
- How to script with triggers
- How to raycast

In this hour, you'll learn to work with the most prevalent physics concept in video games: collision. Collision, simply put, involves knowing when the border of one object has come into contact with another object. You'll begin by learning what rigidbodies are and what they can do for you. After that, you'll experiment with Unity's powerful built-in physics engines—Box2D and PhysX. From there, you'll learn some more subtle uses of collision with triggers. You'll end the hour by learning to use a raycast to detect collisions.

Rigidbodies

For objects to take advantage of Unity's built-in physics engines, they must include a component called a *rigidbody*. Adding a rigidbody component makes an object behave like a real-world physical entity. To add a rigidbody component, simply select the object that you want (make sure you choose the version without 2D on the end) and click **Component > Physics > Rigidbody**. The new rigidbody component is added to the object in the Inspector (see Figure 9.1).

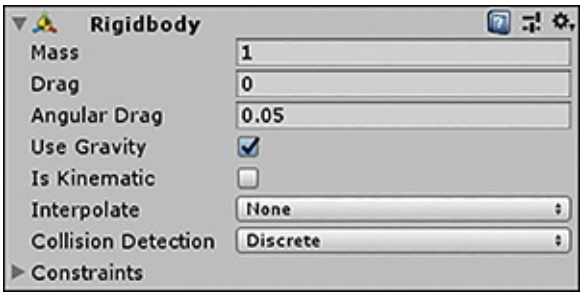

FIGURE 9.1
The rigidbody component.

The rigidbody component has several properties that you have not seen yet. Table 9.1 describes them.

TABLE 9.1 Rigidbody Properties

Property	Description
Mass	Specifies the mass of the object, in arbitrary units. Use 1 unit = 1 kg unless you have a good reason to deviate. Higher mass requires more force to move.
Drag	Indicates how much air resistance is applied to the object when moving. Higher drag means more force is required to move an object and stops a moving object more quickly. A drag of 0 applies no air resistance.
Angular Drag	Much like drag, indicates air resistance applied when spinning.
Use Gravity	Determines whether Unity's gravity calculations are applied to this object. Gravity affects an object more or less depending on its drag.
Is Kinematic	Allows you to control the movement of a rigidbody yourself. If an object is kinematic, it will not be affected by forces.
Interpolate	Determines how and whether motion for an object is smoothed. By default, this property is set to None. With Interpolate, smoothing is based on the previous frame, whereas with Extrapolate, it is based on the next assumed frame. It is recommended to turn this on only if you notice a lagging or stutter, and you want your physical objects to have smoother motion.

Collision Detection	Determines how collision is calculated. Discrete is the default setting, and it is how all objects test against each other. The Continuous setting can help if you are having trouble detecting collisions with very fast objects. Be aware, though, that Continuous can have a large impact on performance. The Continuous Dynamic setting uses discrete detection against other discrete objects and continuous detection against other continuous objects.
Constraints	Specifies movement limitations that a rigidbody enforces on an object. By default, these limitations are turned off. Freezing a position axis prevents the object from moving along that axis, and freezing a rotation axis prevents an object from rotating about that axis.

▼ TRY IT YOURSELF

Using Rigidbodies

Follow these steps to see a rigidbody in action:

1. Create a new project or scene. Add a cube to the scene and place it at (0, 1, –5).
2. Run the scene. Notice how the cube floats in front of the camera.
3. Add a rigidbody to the object (by selecting **Components > Physics > Rigidbody**).
4. Run the scene. Notice that the object now falls due to gravity.
5. Experiment with the drag and constraints properties and note their effects.

Enabling Collision

Now that you have objects moving around, it is time to start getting them to crash into each other. For objects to detect collision, they need a component called a collider. A *collider* is a perimeter that is projected around an object that can detect when other objects enter it.

Colli

Colliders

Geometric objects such as spheres, capsules, and cubes already have collider components on them when they are created. You can add a collider to an object that doesn't have one by selecting **Component > Physics** and then choosing the collider shape you want from the menu. Figure 9.2 shows the various collider shapes you can choose from.

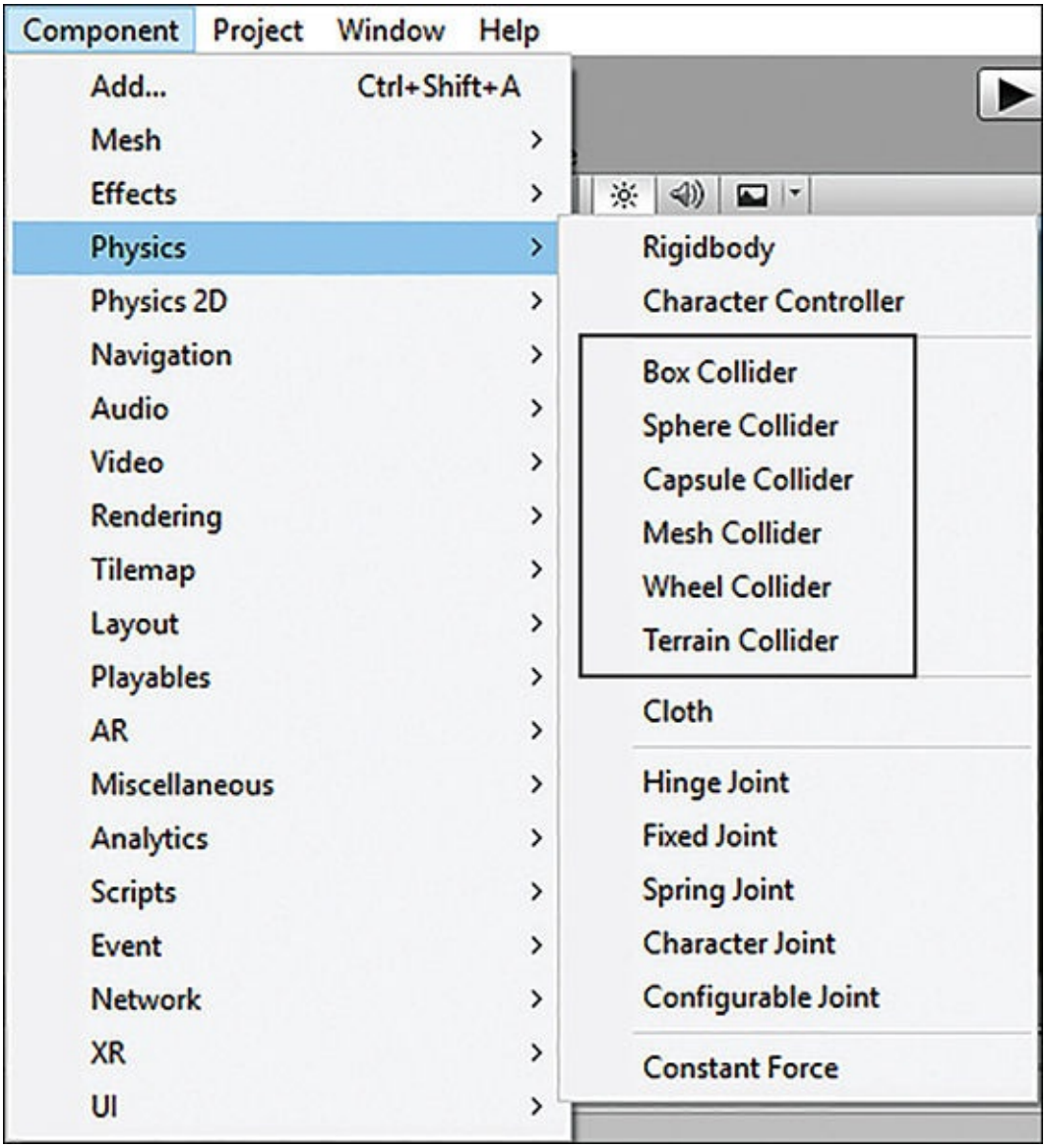

FIGURE 9.2
The available colliders.

Once a collider is added to an object, the collider object appears in the Inspector. Table 9.2 describes the collider properties. In addition to these, if a collider is a

sphere or capsule, you might have an additional property, such as Radius. These geometric properties behave exactly as you would expect them to.

TABLE 9.2 Collider Properties

Property	Description
Edit Collider	Allows you to graphically adjust the collider size (and in some cases shape) in the Scene view.
Is Trigger	Determines whether the collider is a physical collider or a trigger collider. Triggers are covered in greater detail later in this hour.
Material	Allows you to apply physics materials to objects to change the way in which they behave. You can make an object behave like wood, metal, or rubber, for instance. Physics materials are covered later in this hour.
Center	Indicates the center point of the collider relative to the containing object.
Size	Specifies the size of the collider.

TIP

Mix-and-Match Colliders

Using different-shaped colliders on objects can have some interesting effects. For instance, making the collider on a cube much bigger than the cube makes the cube look like it is floating above a surface. Likewise, using a smaller collider allows an object to sink into a surface. Furthermore, putting a sphere collider on a cube allows the cube to roll around like a ball. Have fun experimenting with the various ways to make colliders for objects.

TRY IT YOURSELF

Experimenting with Colliders

It's time to try out some of the different colliders and see how they interact. Be sure to save this exercise because you'll use it again later in the hour. Follow these steps:

1. Create a new project or scene. Add two cubes to it.

2. Place one cube at (0, 1, –5) and put a rigidbody on it. Place the other cube at (0, –1, –5) and scale it to (4, .1, 4) with a rotation of (0, 0, 15). Put a rigidbody on the second cube as well but uncheck the **Use Gravity** property.

3. Run the scene and notice how the top cube falls onto the other cube. They then fall away from the screen.

4. Now, on the bottom cube, under the constraints of the rigidbody component, freeze all three axes for both position and rotation.

5. Run the scene and notice how the top cube now falls and stops on the bottom cube.

6. Remove the box collider from the top cube (by right-clicking the Box Collider component and selecting **Remove Component**). Add a sphere collider to the top cube (by selecting **Component > Physics > Sphere Collider**). Give the bottom cube a rotation of (0, 0, 350).

7. Run the scene. Notice how the box rolls off of the ramp like a sphere, even though it is a cube.

8. Experiment with various colliders. Another fun experiment is to change the constraints on the bottom cube. Try freezing the y axis position and unfreezing everything else. Try out different ways of making the boxes collide.

TIP

Complex Colliders

You may have noticed a collider called the mesh collider. This collider is specifically not discussed in this book because it is quite difficult to make and very easy to get wrong. Basically, a *mesh collider* is a collider that uses a mesh to define its shape. In actuality, the other colliders (box, sphere, and so on) are mesh colliders; they are just simple built-in ones. While mesh colliders can provide for very accurate collision detection, they can also hurt performance. A good habit to get into (at least at this stage in your learning) is to compose a complex object with several basic colliders. If you have a humanoid model, for example, try spheres for the head and hands and capsules for the torso, arms, and legs. You will save on performance and still have some very sharp collision detection.

Physics Materials

Physics materials can be applied to colliders to give objects varied physical properties. For instance, you can use a rubber material to make an object bouncy or an ice material to make it slippery. You can even make your own physics materials to emulate specific materials of your choosing.

There are some premade physics materials in the Characters standard assets pack. To import them, select **Assets > Import Package > Characters**. In the Import screen, click the **None** button to deselect all and then scroll down. Check the box next to Physics Materials near the bottom and click **Import**. This adds an **Assets\Standard Assets\PhysicsMaterials** folder in your project that contains physics materials for Bouncy, Ice, Metal, Rubber, and Wood. To create a new physics material, right-click the Assets folder in the Project view and select **Create > Physic Material** or **Create > Physics Material 2D** (if you're working with 2D physics, covered in Hour 12, "2D Game Tools").

A physics material has a set of properties that determine how it behaves on a physical level (see Figure 9.3). Table 9.3 describes the physics material properties. You can apply a physics material to an object by dragging it from the Project view onto an object with a collider.

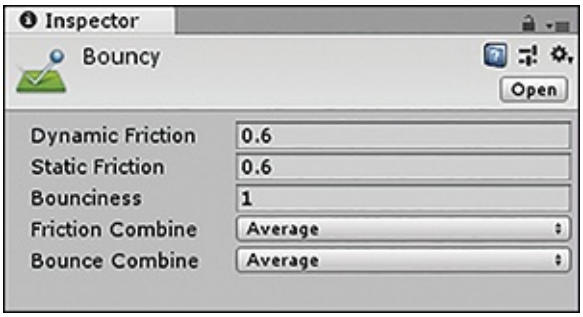

FIGURE 9.3
The properties of physics materials.

TABLE 9.3 Physics Material Properties

Property	Description
Dynamic Friction	Specifies the friction applied when an object is already moving. Lower numbers make an object more slippery.
Static Friction	Specifies the friction applied when an object is stationary. Lower numbers make an object more slippery.
Bounciness	Specifies the amount of energy retained from a collision. A value

Bouncines	Specifies the amount of energy retained from a collision. A value of 1 causes the object to bounce without any loss of energy; it will bounce forever. A value of 0 prevents the object from bouncing.
Friction Combine	Determines how the friction of two colliding objects is calculated. The friction can be averaged, the smallest or largest can be used, or they can be multiplied.
Bounce Combine	Determines how the bounce of two colliding objects is calculated. The bounce can be averaged, the smallest or largest can be used, or they can be multiplied.

The effects of a physics material can be as subtle or as distinct as you like. Try it out for yourself and see what kinds of interesting behaviors you can create.

Triggers

So far, you have seen physical colliders—colliders that react in a positional and rotational fashion using Unity's built-in physics engine. If you think back to Hour 6, "Game 1: *Amazing Racer*," however, you can probably remember using another type of collider. Remember how the game detected when the player entered the water hazards and finish zone? That was the trigger collider at work. A trigger detects collision just like normal colliders do, but it doesn't do anything specific about it. Instead, triggers call three specific methods that allow you, the programmer, to determine what the collision means:

Click here to view code image

```
void OnTriggerEnter(Collider other) // is called when an object enters t
void OnTriggerStay(Collider other) // is called while an object stays in
void OnTriggerExit(Collider other) // is called when an object exits the
```

Using these methods, you can define what happens whenever an object enters, stays in, or leaves the collider. For example, if you want to write a message to the Console whenever an object enters the perimeter of a cube, you can add a trigger to the cube. Then you attach a script to the cube with the following code:

Click here to view code image

```
void OnTriggerEnter(Collider other)
{
    print("Object has entered collider");
}
```

You might notice the one parameter to the trigger methods: the variable `other`

of type `Collider`. This is a reference to the object that entered the trigger. Using that variable, you can manipulate the object however you want. For instance, to modify the preceding code to write to the Console the name of the object that enters the trigger, you can use the following:

Click here to view code image

```
void OnTriggerEnter(Collider other)
{
    print(other.gameObject.name + " has entered the trigger");
}
```

You could even go so far as to destroy the object entering the trigger with some code like this:

Click here to view code image

```
void OnTriggerEnter(Collider other)
{
    Destroy(other.gameObject);
}
```

▼ TRY IT YOURSELF

Working with Triggers

In this exercise, you'll get a chance to build an interactive scene with a functioning trigger:

1. Create a new project or scene. Add a cube and a sphere to the scene.
2. Place the cube at (–1, 1, –5) and place the sphere at (1, 1, 5).
3. Create two scripts named TriggerScript and MovementScript. Place TriggerScript on the cube and MovementScript on the sphere.
4. On the cube's collider, check **Is Trigger**. Add a rigidbody to the sphere and uncheck **Use Gravity**.
5. Add the following code to the `Update()` method of MovementScript:

Click here to view code image

```
    float mX = Input.GetAxis("Mouse X")  10;
    float mY = Input.GetAxis("Mouse Y")  10;
    transform.Translate(mX, mY, 0);
```

6. Add the following code to TriggerScript:

Click here to view code image

```
void OnTriggerEnter (Collider other)
{
    print(other.gameObject.name + " has entered the cube");
}
void OnTriggerStay (Collider other)
{
   print(other.gameObject.name + " is still in the cube");
}
void OnTriggerExit (Collider other)
{
     print(other.gameObject.name + " has left the cube");
}
```

Be sure to place this code outside any methods but inside the class—that is, at the same level of indentation as the `Start()` and `Update()` methods.

7. Run the scene. Notice how the mouse moves the sphere. Collide the sphere with the cube and pay attention to the Console output. Notice how the two objects don't physically react, but they still interact. Can you work out which cell of Figure 9.4 this interaction falls into?

	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider		collision			trigger	trigger
Rigidbody Collider	collision	collision	collision	trigger	trigger	trigger
Kinematic Rigidbody Collider		collision		trigger	trigger	trigger
Static Trigger Collider		trigger	trigger		trigger	trigger
Rigidbody Trigger Collider	trigger	trigger	trigger	trigger	trigger	trigger
Kinematic Rigidbody Trigger Collider	trigger	trigger	trigger	trigger	trigger	trigger

FIGURE 9.4
Collider interaction matrix.

NOTE

Collisions Not Working

Collisions Not Working

Not all collision scenarios result in collisions actually happening. Refer to Figure 9.4 to look up whether the interaction between two objects will create a collision, a trigger method in code, or neither.

Static colliders are simply colliders on any game object that doesn't have a rigidbody component. They become rigidbody colliders when you add a rigidbody component, and these become kinematic rigidbody colliders when you check the Is Kinematic box. For each of these three colliders, you can also select Is Trigger if desired. These options lead to the six types of colliders shown in Figure 9.4.

Raycasting

Raycasting is the act of sending out an imaginary line, called a *ray*, and seeing what it hits. Imagine, for instance, looking through a telescope. Your line of sight is the ray, and whatever you can see at the other end is what your ray hits. Game developers use raycasting all the time for things like aiming, determining line of sight, gauging distance, and more. There are a few `Raycast` methods in Unity. The two most common uses are laid out here. The first `Raycast` method looks like this:

Click here to view code image

```
bool Raycast(Vector3 origin, Vector3 direction, float distance, LayerMas
```

Notice that this method takes quite a few parameters. Also notice that it uses a variable called `Vector3` (which you've used before). `Vector3` is a variable type that holds three floats inside it. Using it is a great way to specify x, y, and z coordinates without requiring three separate variables. The first parameter, `origin`, is the position at which the ray starts. The second, `direction`, is the direction in which the ray travels. The third parameter, `distance`, determines how far out the ray will go, and the final variable, `mask`, determines what layers will be hit. You can omit both the `distance` and the `mask` variables. If you do, the ray will travel an infinite (or very far, in this case) distance and hit all object layers.

As mentioned earlier, there are many things you can do with rays. If you want to determine whether something is in front of the camera, for example, you can attach a script with the following code:

Click here to view code image

```
void Update()
{
    // cast the ray from the camera's position in the forward direction
    if (Physics.Raycast(transform.position, transform.forward, 10))
        print("There is something in front of the camera!");
}
```

Another way you can use this method is to find the object that the ray collided with. This version of the method uses a special variable type called `RaycastHit`. Many versions of the `Raycast` method use `distance` (or don't) and `layer mask` (or don't). The most basic way to use this version of the method, though, looks something like this:

Click here to view code image

```
bool Raycast(Vector3 origin, Vector3 direction, out Raycast hit, float d
```

There is one new interesting thing about this version of the method. You might have noticed that it uses a new keyword that you have not seen before: `out`. This keyword means that when the method is done running, the variable `hit` will contain whatever object was hit. The method effectively sends the value back *out* when it is done. It works this way because the `Raycast()` method already returns a Boolean variable that indicates whether it hit something. Since a method cannot return two variables, you use `out` to get more information.

▼ TRY IT YOURSELF

Casting Some Rays

In this exercise, you will create an interactive "shooting" program. This program will send a ray from the camera and destroy whatever objects it comes into contact with. Follow these steps:

1. Create a new project or scene and add four spheres to it.
2. Place the spheres at (–1, 1, –5), (1, 1.5, –5), (–1, –2, 5), and (1.5, 0, 0).
3. Create a new script called RaycastScript and attach it to the Main Camera. Inside the `Update()` method for the script, add the following:

Click here to view code image

```
    float dirX = Input.GetAxis ("Mouse X");
    float dirY = Input.GetAxis ("Mouse Y");
    // opposite because we rotate about those axes
    transform.Rotate (dirY, dirX, 0);
```

```
transform.Rotate (diff, diffx, 0);
CheckForRaycastHit (); // this will be added in the next step
```

4. Add the method `CheckForRaycastHit()` to your script by adding the following code outside a method but inside the class:

Click here to view code image

```
void CheckForRaycastHit() {
    RaycastHit hit;
    if (Physics.Raycast (transform.position, transform.forward, ou
     print (hit.collider.gameObject.name + " destroyed!");
     Destroy (hit.collider.gameObject);
  }
}
```

5. Run your scene. Notice how moving the mouse moves the camera. Try to center the camera on each sphere. Notice how the sphere is destroyed and the message is written to the Console.

Summary

In this hour, you have learned about object interactions through collision. First, you learned about the basics of Unity's physics capabilities with rigidbodies. Then you worked with various types of colliders and collision. From there, you learned that collision is more than just stuff bouncing around, and you got hands-on with triggers. Finally, you learned to find objects by using raycasting.

Q&A

Q. Should all my objects have rigidbodies?

A. Rigidbodies are useful components that serve largely physical roles. That said, adding rigidbodies to every object can have strange side effects and may reduce performance. A good rule of thumb is to add components only when they are needed and not preemptively.

Q. There are several colliders we didn't talk about. Why not?

A. Most colliders either behave the same way as the ones covered here or are beyond the scope of this text (and are, therefore, omitted). Suffice it to say that this text still provides what you need to know to make some very fun games.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. What component is required on an object if you want it to exhibit physical traits such as falling?
2. True or False: An object can have only a single collider on it.
3. What sorts of things are raycasts useful for?

Answers

1. Rigidbody
2. False. An object can have many, and varied, colliders on it.
3. Determining what an object can see and finding objects along line of sight as well as finding distances between objects

Exercise

In this exercise, you'll create an interactive application that uses motion and triggers. The exercise requires you to creatively determine a solution (because one is not presented here). If you get stuck and need help, you can find the solution to this exercise, called Hour 9 Exercise, in the book assets for Hour 9.

1. Create a new project or scene. Add a cube to the scene and position it at (–1.5, 0, –5). Scale the cube to (.1, 2, 2) and rename it **LTrigger**.
2. Duplicate the cube (by right-clicking the cube in the Hierarchy view and selecting **Duplicate**). Name the new cube **RTrigger** and place it at (1.5, 0, –5.)
3. Add a sphere to your scene and place it at (0, 0, –5). Add a rigidbody to the sphere and uncheck **Use Gravity**.
4. Create a script named TriggerScript and place it on both LTrigger and RTrigger. Create a script called **MotionScript** and place it on the sphere.
5. Now comes the fun part. You need to create the following functionality in your application:
   - The player should be able to move the sphere with the arrow keys.

- When the sphere enters, exits, or stays in either of the triggers, the corresponding message should be written to the Console. The message should indicate which of the three it is and the name of the trigger that the sphere collided with (LTrigger or RTrigger).
- There is a hidden “gotcha” built into this exercise that you need to overcome in order to make this work.

Good luck!