1. Create a new project or scene. Add to the project a script named PlayerInput (or modify the existing one) and attach it to the Main Camera.

2. Add the following code to the `Update` method in the PlayerInput script:

Click here to view code image

```
float mxVal = Input.GetAxis("Mouse X");
float myVal = Input.GetAxis("Mouse Y");
if(mxVal != 0)
    print("Mouse X movement selected: " + mxVal);
if(myVal != 0)
    print("Mouse Y movement selected: " + myVal);
```

3. Save the script and run the scene. Read through the Console to see the output when you move the mouse around.

Accessing Local Components

As you have seen numerous times in the Inspector view, objects are composed of various components. There is always a transform component, and there may optionally be any number of other components, such as a Renderer, Light, and Camera. Scripts are also components, and together these components give a game object its behavior.

Using `GetComponent`

You can interact with components at runtime through scripts. The first thing you must do is get a *reference* to the component you want to work with. You should do this in `Start()` and save the result in a variable. This way, you won't have to waste time repeating this relatively slow operation.

The `GetComponent<Type>()` method has a slightly different syntax than you've seen to this point, using chevrons to specify the type you are looking for (for example, Light, Camera, script name).

`GetComponent()` returns the first component of the specified type that is attached to the same game object as your script. As mentioned earlier in this hour, you should then assign this component to a local variable so you can access it later. Here's how you do this:

Click here to view code image

```
Light lightComponent; // A variable to store the light component.
Start ()
{
    lightComponent = GetComponent<Light> ();
    lightComponent.type = LightType.Directional;
}
```

Once you have a reference to a component, you can then easily modify its properties through code. You do so by typing the name of the variable storing the reference followed by a dot followed by whatever property you want to modify. In the example above, you change the `type` property of the light component to be `Directional`.

Accessing the Transform

The component you'll most commonly work with is the transform component. By editing it, you can make objects move around the screen. Remember that an object's transform is made up of its translation (or position), its rotation, and its scale. Although you can modify those directly, it is easier to use some built-in options called the `Translate()` method, the `Rotate()` method, and the `localScale` variable, as shown here:

Click here to view code image

```
// Moves the object along the positive x axis.
// The '0f' means 0 is a float (floating point number). It is the way Un
transform.Translate(0.05f, 0f, 0f); // Rotates the object along the z ax
transform.Rotate(0f, 0f, 1f);
// Scales the object to double its size in all directions
transform.localScale = new Vector3(2f, 2f, 2f);
```

NOTE

Finding the Transform

Because every game object has a transform, there's no need to do an explicit find operation; you can access the transform directly as above. This is the only component that works this way; the rest must be accessed using a `GetComponent` method.

Because `Translate()` and `Rotate()` are methods, if the preceding code were put in `Update()`, the object would continually move along the positive x axis while being rotated along the z axis.

▼ TRY IT YOURSELF

Transforming an Object

Follow these steps to see the previous code in action by applying it to an object in a scene:

1. Create a new project or scene. Add a cube to the scene and position it at (0, -1, 0).
2. Create a new script and name it **CubeScript**. Place the script on the cube. In Visual Studio, enter the following code to the `Update` method:

Click here to view code image

```
transform.Translate(.05f, 0f, 0f);
transform.Rotate(0f, 0f, 1f);
transform.localScale = new Vector3(1.5f, 1.5f, 1.5f);
```

3. Save the script and run the scene. You may need to move to Scene view to see the full motion. Notice that the effects of the `Translate()` and `Rotate()` methods are cumulative, and the variable `localScale` is not; `localScale` does not keep growing.

Accessing Other Objects

Many times, you want a script to be able to find and manipulate other objects and their components. Doing so is simply a matter of finding the object you want and calling on the appropriate component. There are a few basic ways to find objects that aren’t local to the script or to the object the script is attached to.

Finding Other Objects

The first and easiest way to find other objects to work with is to use the editor. By creating a public variable on the class level of type `GameObject`, you can simply drag the object you want onto the script component in the Inspector view. The code to set this up looks like this:

Click here to view code image

```
// This is here for reference
public class SomeClassScript : MonoBehaviour
{
    // This is the game object you want to access
    public GameObject objectYouWant;
    // This is here for reference
    void Start() {}
}
```

After you have attached the script to a game object, you see a property in the Inspector called Object You Want (see Figure 8.3). Just drag any game object you want onto this property to have access to it in the script.

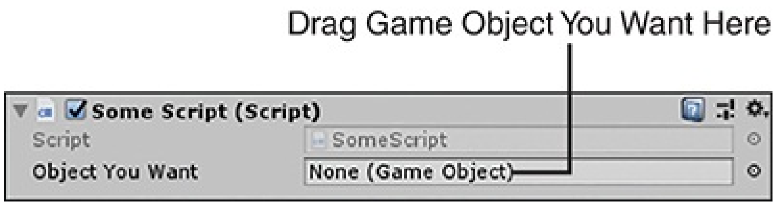

FIGURE 8.3
The new Object You Want property in the Inspector.

Another way to find a game object is by using one of the Find methods. As a rule of thumb, if you want a designer to be able to connect the object, or if it's optional, then connect via the Inspector. If disconnecting would break the game, then use a Find method. There are three major ways to find using a script: by name, by tag, and by type.

One option is to search by the object's name. The object's name is what it is called inside the Hierarchy view. If you were looking for an object named `Cube`, the code would look like this:

Click here to view code image

```
// This is here for reference
public class SomeClassScript : MonoBehaviour
{
    // This is the game object you want to access
    private GameObject target; //  Note this doesn't need to be public i
    // This is here for reference
    void Start()
    {
        target = GameObject.Find("Cube");
    }
}
```

The shortcoming of this method is that it just returns the first item it finds with the given name. If you have multiple `Cube` objects, you won't know which one you are getting.

TIP

Finding Efficiency

Be aware that using a `Find()` method is extremely slow as it searches every game object in the scene, in order, until it finds a match. In very large scenes, the amount of time this method takes can cause a noticeable framerate drop in your game. It is advisable to ***never*** use `Find()` if you can avoid it. That being said, there are many times when it is necessary. In these instances, however, it is very beneficial to call the method in `Start()` and save the results of the find in a variable for future use to minimize the impact.

Another way to find an object is by its tag. An object's tag is much like its layer (discussed earlier in this hour). The only difference is semantics. The *layer* is used for broad categories of interaction, whereas the *tag* is used for basic identification. You create tags using the Tag Manager (click **Edit > Project Settings > Tags & Layers**). Figure 8.4 shows how to add a new tag to the Tag Manager.

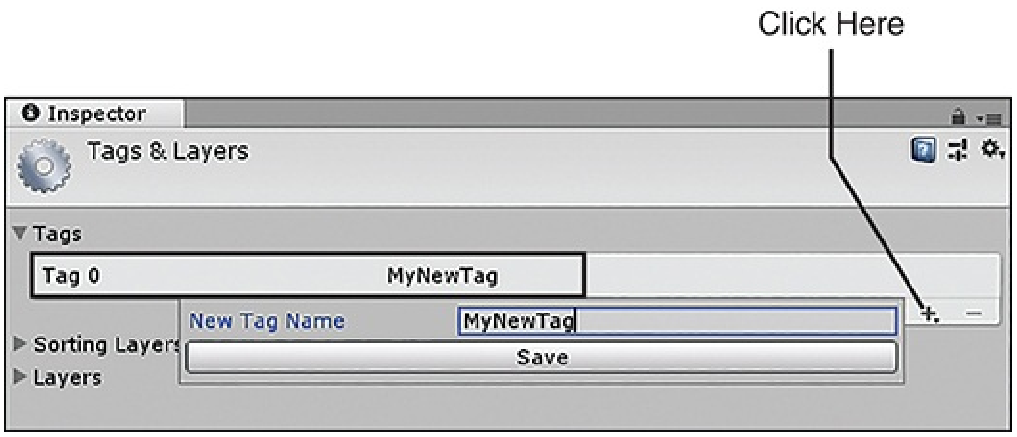

FIGURE 8.4
Adding a new tag.

Once a tag is created, simply apply it to an object by using the Tag drop-down list in the Inspector view (see Figure 8.5).

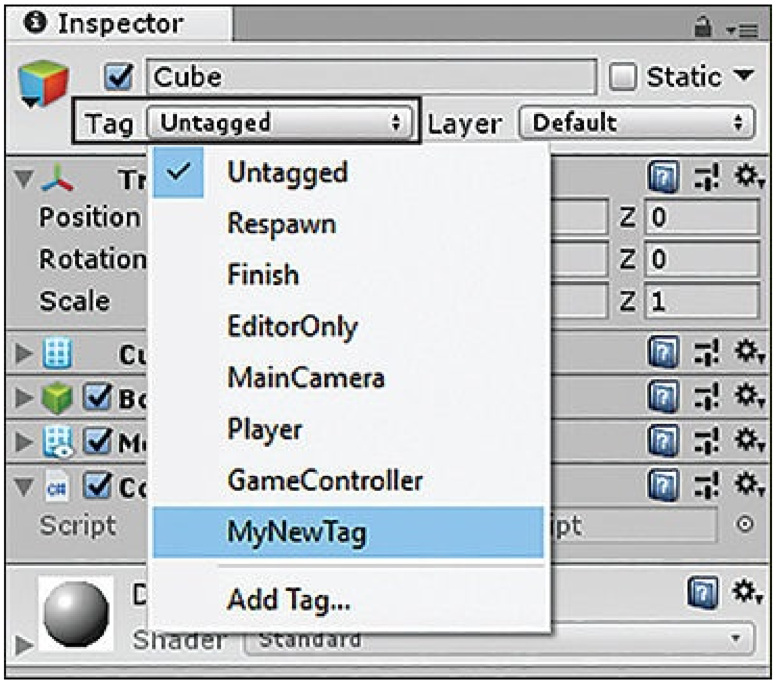

FIGURE 8.5
Selecting a tag.

After a tag is added to an object, you can find it by using the `FindWithTag()` method:

Click here to view code image

```
// This is here for reference
public class SomeClassScript : MonoBehaviour
{
    // This is the game object you want to access
    private GameObject target;
    // This is here for reference
    void Start()
    {
        target = GameObject.FindWithTag("MyNewTag");
    }
}
```

There are additional `Find()` methods available, but the ones covered here should work for you in most situations.

Modifying Object Components

Once you have a reference to another object, working with the components of that object is almost exactly the same as using local components. The only difference is that now, instead of simply writing the component name, you need to write the object variable and a period in front of it, like so:

Click here to view code image

```
// This accesses the local component, not what you want
transform.Translate(0, 0, 0);
// This accesses the target object, what you want
targetObject.transform.Translate(0, 0, 0);
```

▼ TRY IT YOURSELF

Transforming a Target Object

Follow these steps to modify a target object by using scripts:

1. Create a new project or scene. Add a cube to the scene and position it at (0, -1, 0).

2. Create a new script and name it **TargetCubeScript**. Place the script on the Main Camera. In Visual Studio, enter the following code in TargetCubeScript:

Click here to view code image

```
// This is the game object you want to access
private GameObject target;
// This is here for reference
void Start()
{
    target = GameObject.Find("Cube");
} void Update()
{
    target.transform.Translate(.05f, 0f, 0f);
    target.transform.Rotate(0f, 0f, 1f);
    target.transform.localScale = new Vector3(1.5f, 1.5f, 1.5f);
}
```

3. Save the script and run the scene. Notice that the cube moves around even though the script is applied to the Main Camera.

Summary

In this hour, you have explored more scripting in Unity. You have learned all about methods and looked at some ways to write your own. You have also worked with player inputs from the keyboard and mouse. You learned about modifying object components with code, and you finished the hour by learning how to find and interact with other game objects via scripts.

Q&A

Q. How do I know how many methods a task requires?

A. A method should perform a single, concise function. You don’t want to have too few methods with a lot of code because each method would then have to do more than one thing. You also don’t want to have too many small methods because that defeats the purpose of modularizing blocks of code. As long as each process has its own specific method, you have enough methods.

Q. Why didn’t we learn more about gamepads in this hour?

A. The problem with gamepads is that all are unique. In addition, different operating systems treat them differently. The reason they aren’t covered in detail in this hour is that they are too varied, and discussing some of them wouldn’t allow for a consistent reader experience. (In addition, not everyone has a gamepad.)

Q. Is every component editable by script?

A. Yes, at least all of the built-in ones are.

Workshop

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

Quiz

1. True or False: Methods can also be referred to as functions.
2. True or False: Not every method has a return type.
3. Why is it a bad thing to map player interactions to specific buttons?
4. In the Try It Yourself exercises in the sections on local and target components, the cube was translated along the positive x axis and rotated along the z axis. This caused the cube to move around in a big circle. Why?

Answers

1. True
2. False. Every method has a return type. If a method returns nothing, the type is `void`.
3. If you map player interactions to specific buttons, the players will have a much harder time remapping the controls to meet their preferences. If you map controls to generic axes, players can easily change which buttons map to those axes.
4. Transformations happen on the local coordinate system (refer to Hour 2, "Game Objects"). Therefore, the cube did move along the positive x axis. The direction that axis was facing relative to the camera, however, kept changing.

Exercise

It is a good idea to combine each hour's lessons together to see them interact in a more realistic way. In this exercise, you'll write scripts to allow the player directional control over a game object. You can find the solution to this exercise in the book assets for Hour 8 if needed.

1. Create a new project or scene. Add a cube to the scene and position it at (0, 0, -5).
2. Create a new folder called Scripts and create a new script called CubeControlScript. Attach the script to the cube.
3. Try to add the following functionality to the script. If you get lost, check the book assets for Hour 8 for help:
   - Whenever the player presses the left or right arrow key, move the cube along the x axis negatively or positively, respectively. Whenever the player presses the down or up arrow key, move the cube along the z axis negatively or positively, respectively.
   - When the player moves the mouse along the y axis, rotate the *camera* about the x axis. When the player moves the mouse along the x axis, rotate the *cube* about the y axis.
   - When the player presses the M key, have the cube double in size. When the player presses the M key again, have the cube go back to its original size. The M key should act as a toggle switching between the two scale sizes.