CHAPTER 5

Navigation and Functionality

So far, you have explored the terrain in the Scene view with zoom, pan, orbit, and flythrough navigation. Every environment artist's dream is to be able to experience his or her creation firsthand. And while "holodeck" technology is not quite here yet, being able to travel through your environment in first-person mode for the first time is, for many of us, one of the best parts of making real-time games. To make your environment into an actual game environment, you will need to be able to travel around inside the world, responding to terrain topography and large objects during runtime. Only by doing this can you tell if scale, layout, color scheme, and all the rest of the design have come together to create the mood you intend. Even though what you're creating here is just a temporary test scene, the effect is the basis for every entertainment author's goal: suspension of disbelief.

Navigation

Fortunately, Unity provides a prefab for a first-person shooter-type controller. It consists of a Capsule Collider to provide physical interaction with the scene, a camera to provide the viewpoint, and scripts to allow the user to control them through keyboard and mouse.

Anyone who has ever played 3D games, even of the same genre, is well aware that there are many different means of controlling the character or the first person. Moreover, if you get a group of 20 people together in the same room and ask them what the best controls are, you would most likely get a different answer from every person, depending on the game he is currently playing or his all-time favorite game.

You will be tweaking a bit of the First Person Controller as you develop the game, but it is extremely useful to be able to jump right in and travel around the scene right away.

1. Open the Project as you left it in the previous chapter, to the TerrainTest scene.
2. Locate or create a clearing on level ground, as shown in Figure 5-1.

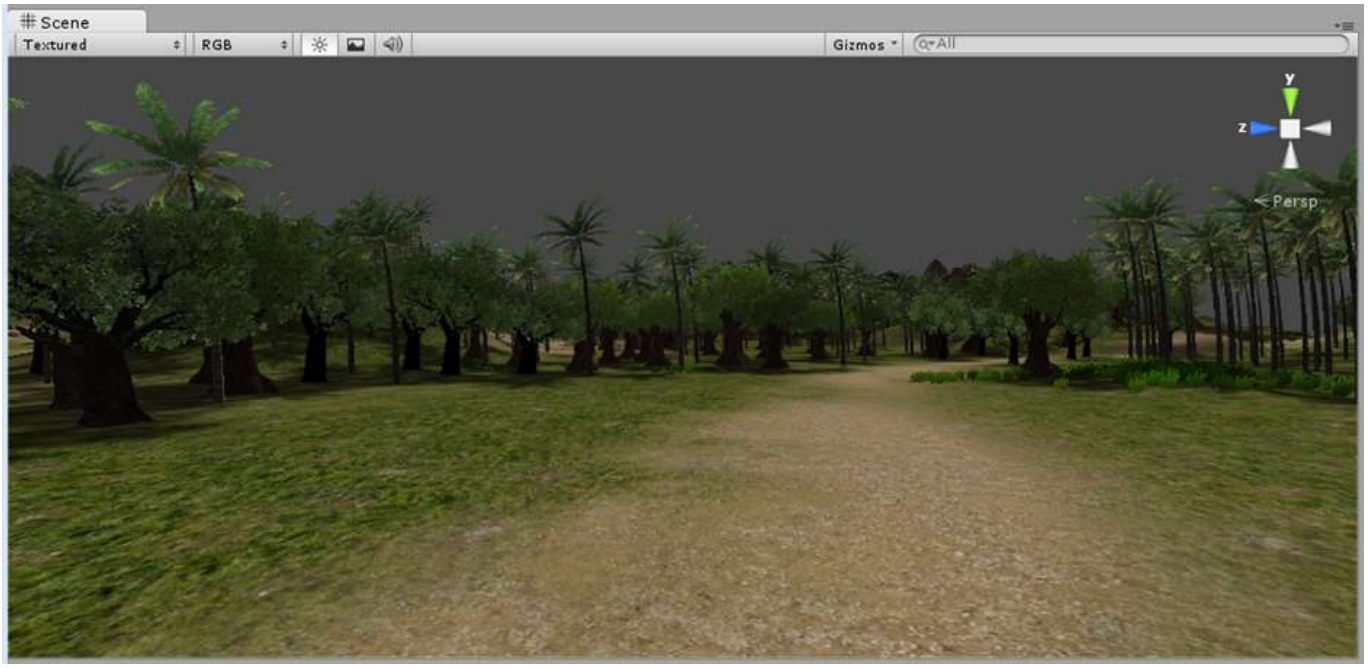

Figure 5-1. *The clearing*

3. Toggle the Overlays button off in the Scene view to hide the fog and sky and turn the fog off in the Render Settings for now.
4. From the Character Controllers folder in the Standard Assets directory, drag the First Person Controller into the Scene view.

Because the First Person Controller is a prefab, its text appears blue in the Hierarchy view, instead of the usual black (see Figure 5-2).

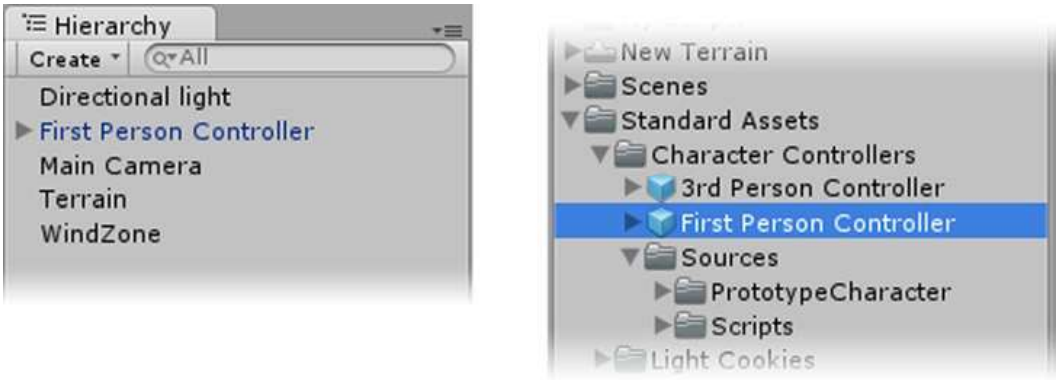

Figure 5-2. *The new First Person Controller in the Hierarchy view (left) and the single-column Project view (right)*

■ **Tip** You can add an object into the scene in a few different ways. Previously, we focused the view and added the object to the Hierarchy so it appears at the focus spot. This time, we added the object directly into the scene. If you know the object's position will need to be changed, you can simply drag it directly into the Scene view and get on with the repositioning.

5. From the GameObject menu, choose Align with View, to move it near the existing view.
6. Use the Scene Gizmo to select the Top view (Y), then click the SceneGizmo label to change it from a top perspective view to an orthographic (no perspective) view (see Figure 5-3).

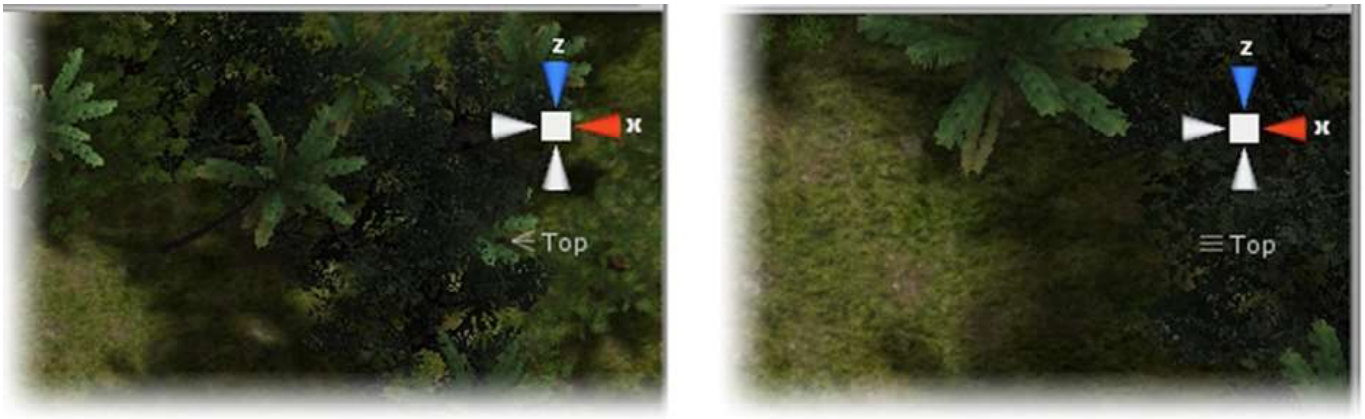

Figure 5-3. *Top view, perspective (left) and orthographic (right)*

The icon next to the label goes from wedge-shaped (perspective) to three short parallel lines (orthographic).

7. Select the First Person Controller in the Hierarchy view.
8. Zoom out to see the area better.
9. Select the Move tool.
10. Using the gray rectangle from the Transform Gizmo, adjust the position of the First Person Controller, if necessary, to center it in the clearing.

■ **Tip** The First Person Controller contains its own camera, which overrides the existing Main Camera object, so you will now see its view in the Game view.

Because the default height was set to 50, the prefab may be beneath the terrain surface, if you modified it in that area.

11. Use the F key to focus the view on the First Person Controller prefab.
12. Click the label again to put the Scene view back to a perspective view, then use Alt+Left mouse to orbit the view back down.
13. Move the First Person Controller up or down in the viewport by dragging its Y arrow up until you can see the ground in the Game view.

14. Use F to focus the First Person Controller prefab again.

15. Adjust its height until the capsule is slightly above ground level (see Figure 5-4).

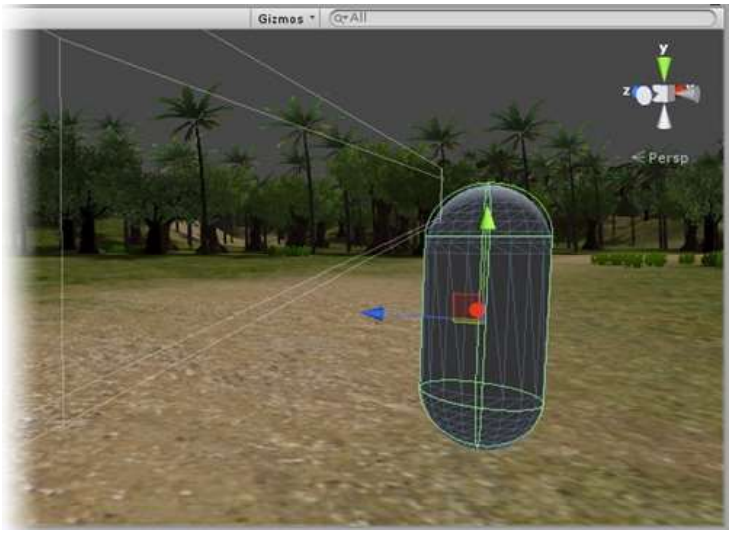

Figure 5-4. *The First Person Controller in the Scene view*

Before you head off to experience your scene in first-person mode, let's take a quick look at what the prefab contains.

1. Select the First Person Controller in the Hierarchy view and click the arrow to the left of the name to open the group (or GameObject), as shown in Figure 5-5.

Figure 5-5. *The contents of the First Person Controller*

It has two children—something called Graphics and Main Camera.

Graphics, as you probably guessed, contains the mesh for the capsule. If you check and uncheck Mesh Renderer in the Inspector (see Figure 5-6), you can see that it affects whether or not the capsule is drawn in the Scene view. Note that neither Cast Shadows nor Receive Shadows is checked. If you are using Unity Pro, you will be able to see what happens if they are checked.

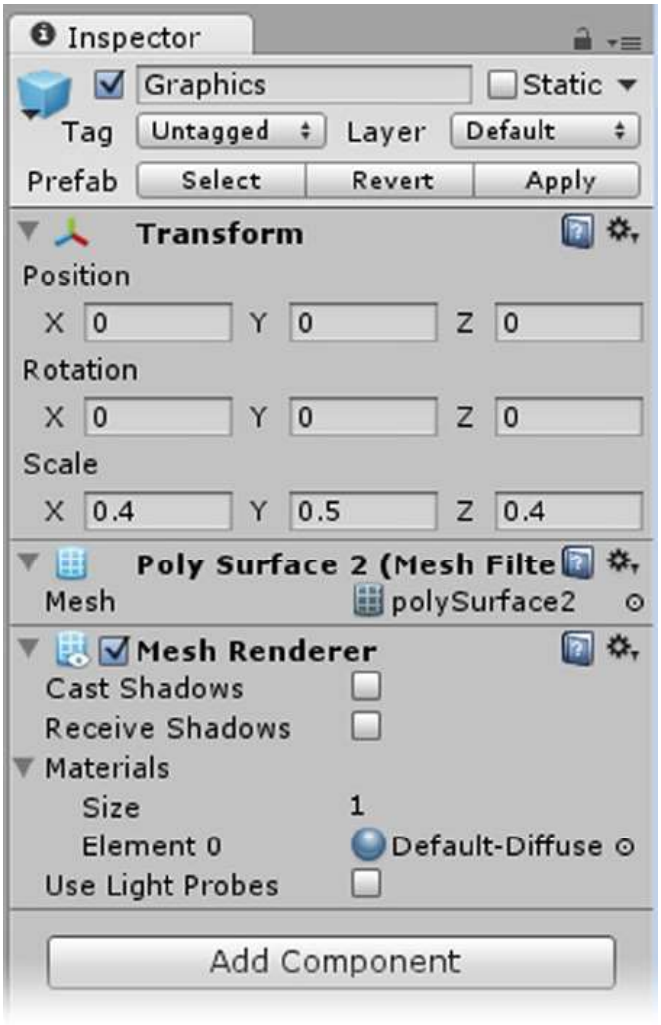

Figure 5-6. *The Graphics object in the Inspector*

Because the camera object is inside the capsule and you are looking out through the back side of the mesh's faces, you will not see it in the Game view. Remember that faces are only drawn on the side that their normals are pointing—in this case, the outside of the capsule.

2. Click Play.

You may notice that the Console status line is telling you there are two Audio Listeners in the scene. (See Figure 5-7.)

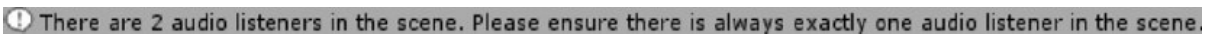

Figure 5-7. *The status line warning*

3. Stop the scene by clicking the Play button again.

As you may surmise, this has to do with enabling audio, but you should have only one per scene. Because every scene must have at least one camera, the default camera comes supplied with that component, as does the Main Camera present in the First Person Controller.

Eventually, you'll have to add and animate other cameras to show the player when and where some particularly important event happens, but for now, just remember to delete the default Main Camera whenever you add the First Person Controller prefab.

4. Select the original scene Main Camera, right-click, and delete it.

The remaining Main Camera object will be referenced by name by various other objects throughout this project, as well as by any other games that use the scripts you'll be creating. Do not rename this object!

5. Click Play.
6. Move around the scene, using W to go forward, S to go backward, A to strafe left, and D to strafe right.
7. Move the mouse around to look up and down or to turn left or right.
8. Use the spacebar to jump.
9. Try running into one of the banyan trees.
10. The collider prevents you from going through it.

If you experimented with tiling on the terrain textures, you may want to select the Terrain object and change the tiling on some of them. Remember, a smaller number increases the tiling, making the image smaller on the terrain.

If you added large rocks to the scene as Detail Meshes, you will notice they do not have colliders; you can move right through them.

11. Select the First Person Controller in the Inspector.
12. You will find three script components and one other component.
13. Take a look at the variables exposed to the Character Motor script (see Figure 5-8).

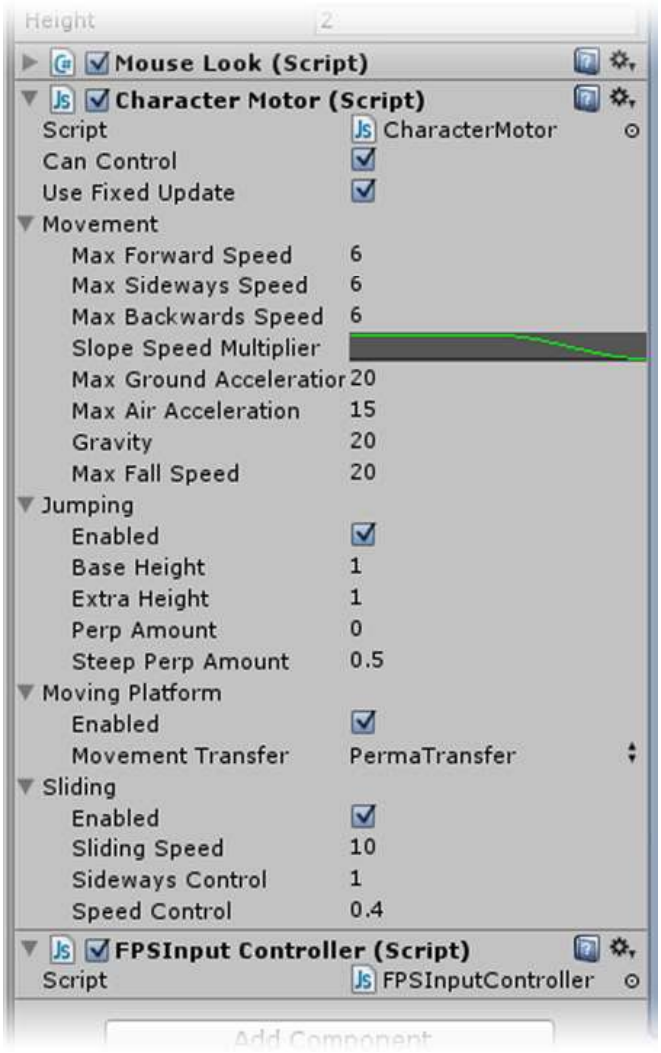

Figure 5-8. *The Character Motor script and its parameters*

As long as you are in Play mode, you can make changes to speed, jump speed, and gravity that will not be permanent.

■ **Tip** To remove focus from the Game view, click outside of it before trying to access other parts of the editor. This will stop the view from spinning wildly.

Opening the script in the editor is somewhat overwhelming. Fortunately, you can just accept that it does lots of nice things to help you travel around the scene and leave it at that. Movement and Jumping are pretty self-explanatory, even if a few of the parameters are not. Moving Platform is an option in the First Person Controller system that you will experiment with a bit later. Sliding is whether the character sticks or slides back down steep slopes when jumping.

The First Person Controller is basically a first-person shooter-type control, with some attributes that will be useful and some that will not. The quickest way to find out what controls what is to turn off scripts and see what happens—or in this case, no longer happens.

1. Select the First Person Controller.
2. Disable the MouseLook script.
3. Test the restricted functionality.

   You'll find that you can still go forward and sideways, but you can't turn left or right; you can only strafe with the left and right arrows or A and D keys.

You can still look up and down, however. A bit of investigation shows that the Main Camera also has a MouseLook script, but that its rotation Axes is set to work on the X axis rather than the Y axis, such as the one on the main group or GameObject (see Figure 5-9). This means that to turn to look, the whole object is turned. When the object goes forward, it will always go in the direction it is facing. When it has to look up or down, only the camera is rotated, leaving the forward direction correct.

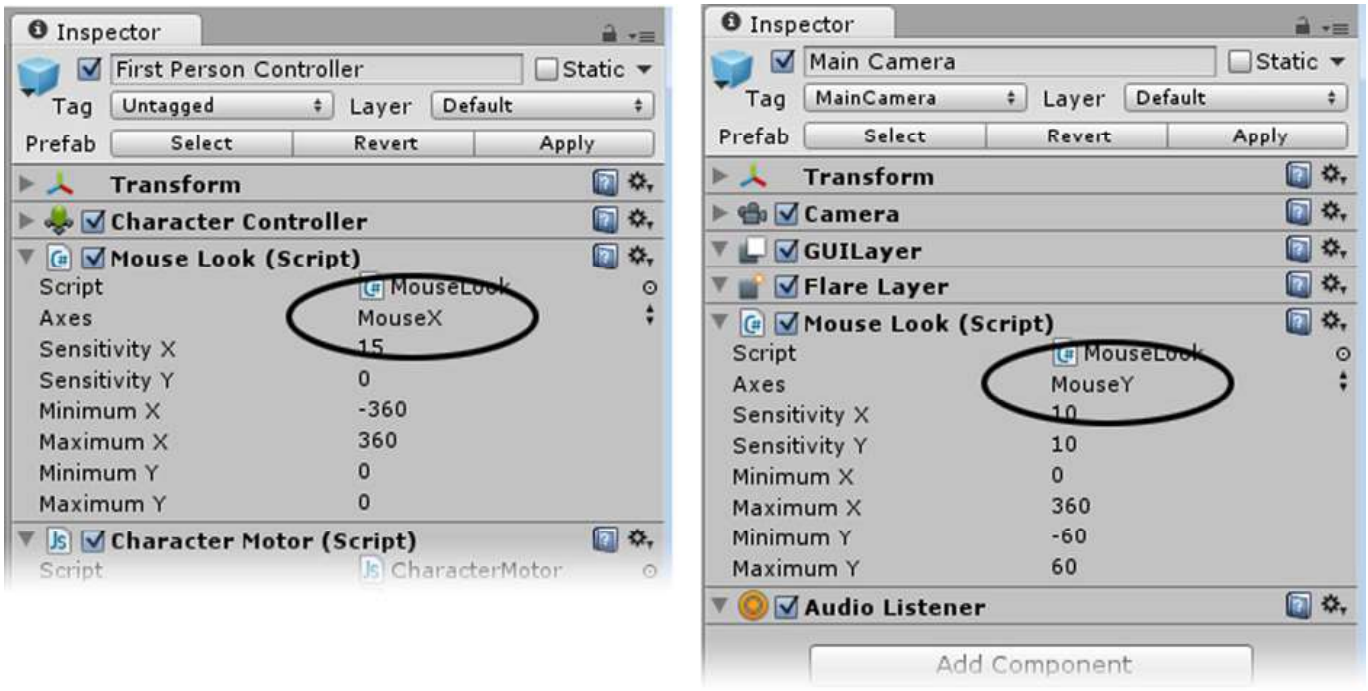

Figure 5-9. *The MouseLook script on the First Person Controller (left) and the MouseLook on the Main Camera (right)*

4. Disable (uncheck) the MouseLook script on the Main Camera object.
5. Test the restricted functionality.

Now you can only go forward, backward, or sideways. On the positive side, you now have full use of the cursor for your point-and-click adventure.

6. Stop Play mode.

Arrow Navigation and Input

Typically, people who are not avid shooter-type game players are more comfortable using the arrow keys to travel and turn. Ideally, you want to cater to both types of users as transparently as possible. To do so, you have to do a bit of tinkering with the Input system.

Unity's Input system lets you assign different input types to a named functionality. This enables two important things: it lets you reference the name of a behavior rather than specifically calling each key or mouse control the behavior can use, and it lets users change or re-map the keys according to their preferred configuration. Because you will be making the two control types work together, you will not allow the user to re-map the keys. You can, however, still make use of the naming benefits when you set up the functionality for the game.

Let's see how the re-mapping works. Jump has only a positive key assigned: the spacebar, or space. It's easy to temporarily change this input assignment.

1. From Edit ➤ Project Settings, choose Input.
2. Click the down arrow next to Axes to see the assignments.

There are currently 17 presets.

3. In the Jump preset, select the Positive Button's entry field (see Figure 5-10).

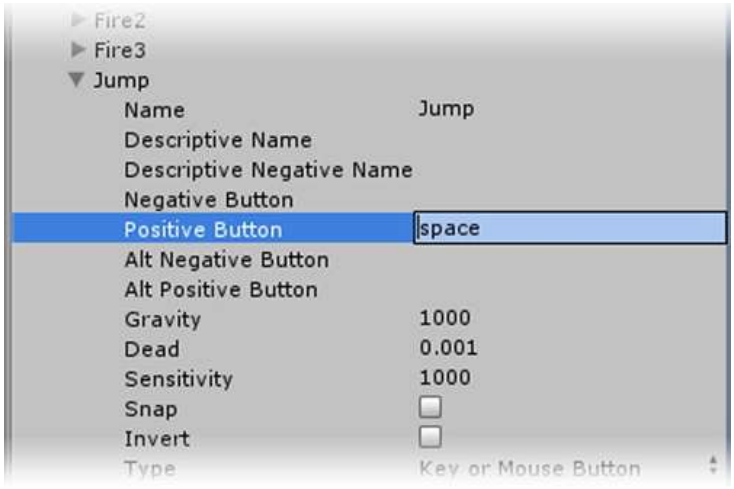

Figure 5-10. *Re-mapping the Jump key*

4. Change space to a different key (in lowercase).
5. Click Play and test.

The jump function now works from the newly assigned key. This means you could eventually let the user re-map or assign the keys without having to change any of your scripts.

6. Stop Play mode and change the Positive Button back to space.

■ **Tip** See Appendix B for the names of the keyboard keys to use in scripts.

Let's get started altering the default First Person Controller navigation.

The first thing to do is turn off strafing for the left and right arrows, leaving that capability strictly to the A and D keys.

7. Open the first Horizontal and Vertical sets (there are duplicates of both further down the list), as shown in Figure 5-11.

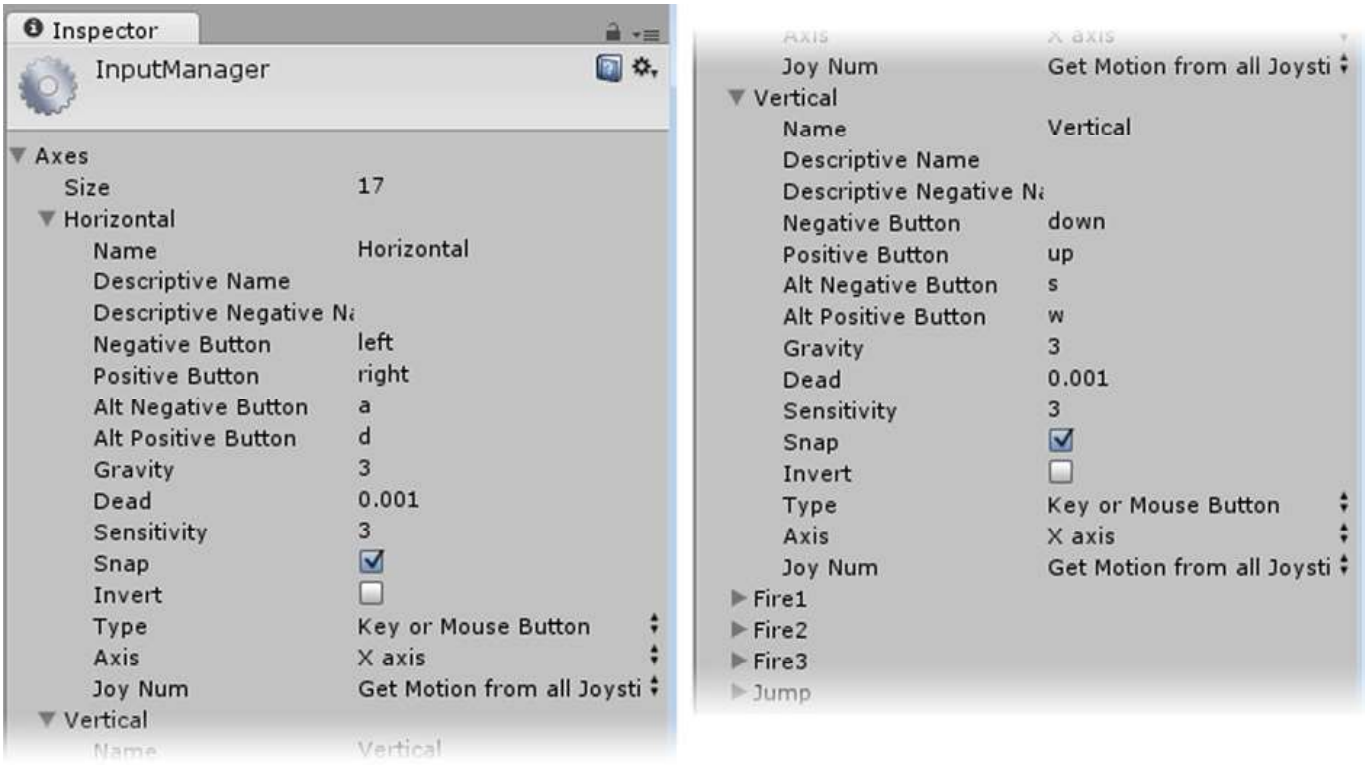

Figure 5-11. *The Input Manager, Horizontal, and Vertical opened in the Inspector*

A good portion of the parameter assignments are handled internally, so you can safely ignore them. Of interest to us are the button assignments: Negative, Positive, Alt Negative, and Alt Positive. Horizontal and Vertical are misleading, if you're used to a 3D world. Vertical is not up and down, as you'd assume, but forward and backward. Horizontal is a strafe (sideways move) left or right. At first glance, these don't make much sense in your 3D world, but if you picture early 2D games played on screen, you can see how the left and right arrows would move the cursor left and right on the horizontal X axis and the up and down arrows would move it up and down on the vertical Y axis. Tip the whole thing down to make the ground plane, and you can see why some 3D packages use Z as the up axis and why Unity calls its axes Vertical and Horizontal (see Figure 5-12).

Figure 5-12. *The origin of Unity navigation terminology. Left: The overhead view of a world where Vertical (on the monitor) moves you forward through the scene. Right: The directions tilted down to use in a 3D world*

The Vertical preset is perfect for our use, employing either the W and S keys or the up and down arrows to move forward or backward. The Horizontal preset needs a bit of modifying to disable the left and right arrow keys.

8. In Horizontal, delete the left and right entries for both the Negative Button and the Positive Button (see Figure 5-13).

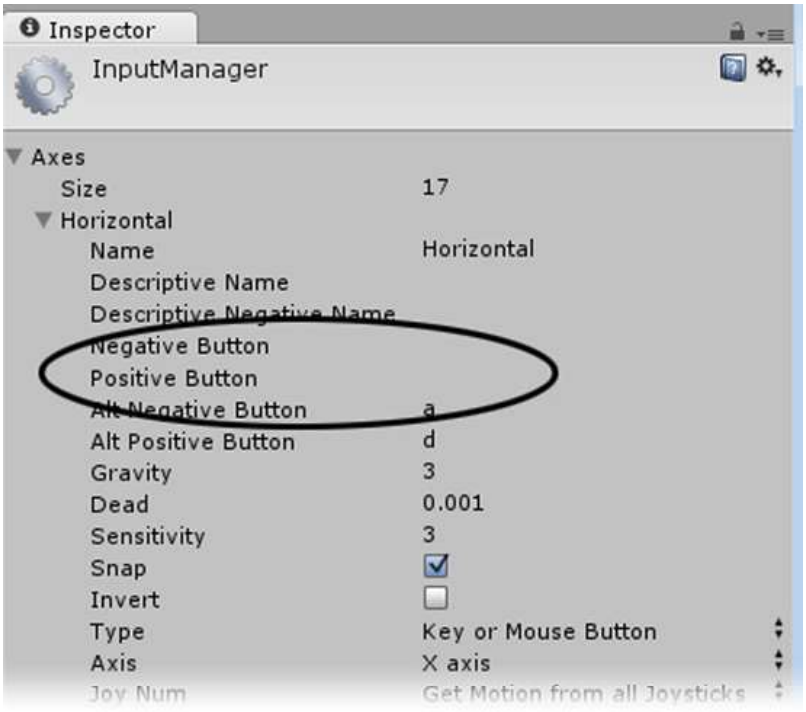

Figure 5-13. *Left and right removed from the Horizontal preset*

9. Click Play and test the left and right arrows.

The arrows no longer strafe.

10. Stop Play mode.

11. At the top of the Inspector, change the Size to **18**.

A duplicate of the last preset, Jump, is created.

12. Open the new preset and change its Name to **Turn**.

13. Change its Negative Button to **left** and its Positive Button to **right**.

14. Change its Sensitivity to **3**.

15. Because you just changed the game settings, now is a good time to save the project.

Now you have to see if the engine will recognize input from the new virtual key. The most logical place to test your new input preset is in the FPS Input Controller script. This is the script that processes the input from the player.

16. Select the FPS Input Controller script in the Project view ➤Standard Assets ➤ Character Controllers ➤ Sources ➤ Scripts. Use Ctrl+D (Cmd+D on the Mac) to duplicate the script.

17. Rename it to **FPAdventurerInputController**.

18. Double-click it to open it in the editor.

At line 4, you'll see a new function called `Awake`. It is a system function and is one of the very first to be evaluated when a game is started. Objects are created first, then scripts. Once the scripts exist, `Awake` is the first function to be called. You will make use of it in several of the later chapters.

Lines 36 and 37 are an oddity; they begin with an `@` and don't end with a semicolon. These lines are not JavaScript, and their functions differ.

- The first, `@script RequireComponent`, checks to make sure a certain component is attached to the GameObject. If that component is missing, the `RequireComponent` attribute will add it, if it is missing when the script is first added to the object.
- The second, `@script AddComponentMenu`, adds this script to the Component menu, so you can easily access it when you want to add it to a new GameObject.

Fortunately, by definition, the whole purpose of `@script` lines is to do things for you, so you don't have to worry about them.

A familiar item in the script should be the `Update` function at line 9. Most of the script's contents, however, seem to be dealing with more vector math than you probably care to analyze. Traditionally, Unity books started with an in-depth analysis of the navigation scripts. Since the newer character controller scripts are three or four times larger and more complicated than their predecessors and this book is written more for artists, feel free to accept its functionality as is and be grateful for it.

■ **Important Concept** You don't necessarily have to understand something to use it. People who get bogged down trying to understand every little detail rarely finish their first game. Don't expect the game to be perfect; you can always revisit it and make improvements as your understanding grows. In this project, we will be constantly improving and refining the scripts and functionality.

There should be a few more features that strike you as being vaguely familiar. Since you were just tweaking the Input settings, `Input.GetAxis("Horizontal")` and `Input.GetAxis("Vertical")` should stand out. While you shouldn't interfere with either of these, they should provide a clue as to how to check for input from your new preset, Turn.

Let's start with something simple.

1. Under the `var directionVector = new Vector3` line, add the following:

```
if (Input.GetAxis("Turn")) print  ("turning");
```

■ **Tip** You can also use `Debug.Log` instead of print. This has the advantage of giving you more information, which may make more sense as you become more familiar with scripting.

---

2. Save the new script via the Editor's Save button or through the File menu.
3. Select the First Person Controller in the Hierarchy view.
4. In the Inspector, with the First Person Controller selected, click the browse icon next to the FPS Input Controller script and select our new version of it, `FPAdventurerController`.

■ **Tip** Replacing a script this way preserves the current values exposed in the Inspector, as long as the variable names have stayed the same.

5. Because you are changing the First Person Controller prefab in the Hierarchy, you may receive a warning about losing the prefab connection. If so, press Continue and click Play.
6. Click in the Game view and then try the arrow keys.

As soon as you press either the left or right key, the "turning" message appears in the status line. You haven't yet told the First Person Controller to turn, but at least you know it reads your code.

You can get a bit more information from `Input.GetAxis`. It returns a value between –1 and 1 as you press the two arrow keys.

7. Change that line to the following:

```
if (Input.GetAxis("Turn")) print ("turning " + Input.GetAxis("Turn"));
```

8. Save the script.
9. Click in the Game view and test the keys again.

The left arrow returns a value less than 0, and the right arrow returns a value greater than 0. This tells you which direction to turn. You will add the turn functionality next, but instead of adding it to the `Update` function, which is frame-dependent, you will use another system function, the `FixedUpdate` function. The `FixedUpdate` function is called at fixed intervals rather than every frame. While this function is primarily used for physics operations, it is handy for other functionality that needs to be consistent across different systems and throughout the game. This way, the turn speed should be about the same on an old machine as on the latest cutting-edge gaming system.

10. Delete the **`if (Input.GetAxis("Turn"))`** line you added to the script.

You are ready for the real functionality now. (Caution: scary math follows).

11. Add the following function above the `// Require a character controller` line at the bottom of the script:

```
//this is for the arrow turn functionality
function FixedUpdate () {

   if (Input.GetAxis("Turn")) {    // the left or right arrow key is being pressed
      // the rotation = direction * speed * sensitivity
      var rotation : float = ( Input.GetAxis("Turn") ) * rotationSpeed * rotationSensitivity ;
      // add the rotation to the current orientation amount
      rotation = rotation + transform.eulerAngles.y ;
      // convert degrees to quaternion for the up axis, Y
      transform.localRotation = Quaternion.AngleAxis ( rotation, Vector3.up ) ;

   }
}
```

12. Near the top of the script, just beneath the motor variable declaration, add the two variables needed to go along with the arrow turn code:

```
//add these for arrow turn
var rotationSpeed : float = 20.0;
internal var rotationSensitivity = 0.1 ; // This makes rotationSpeed more managable.
```

13. Save the script.

■ **Tip** When you are having trouble making a bit of code work, you can often get the help you need on the Unity forums. If you have more complex issues and not enough time to work them out on your own, you might consider trading art assets for code. One of the great things about the Unity forums is the Collaboration section. To get the rotation functionality nailed down quickly for a few key features of our game, I enlisted the help of the team at Binary Sonata Studios. Thanks guys!

---

14. Click in the Game window and test the keys again.

The left and right arrow keys now allow you to turn left and right in the scene.

15. Stop Play mode.

16. Save the scene and save the project.

Rotation math can be very complicated. Euler angles allow us to change rotations on X, Y, and Z axes independently, but they are subject to gimbal lock, which is when the camera is watching a target and must flip to continue. In Figure 5-14, the character on the left can watch the sphere as it goes around him by rotating around his vertical axis with no problems. The figure on the right rotates his head on a horizontal axis, but when the sphere goes overhead, he must turn 180 degrees on his vertical axis to continue watching it once it has passed overhead. With Euler angles, if a rotation involves multiple axes, it needs to be handled one axis at a time. Deciding which axis to solve first is often problematic.

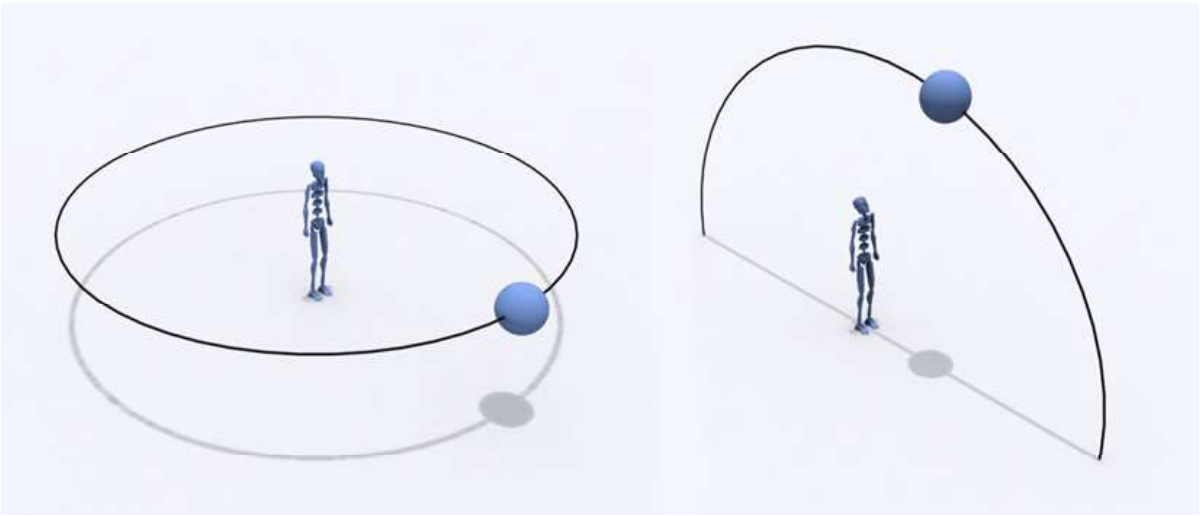

Figure 5-14. *Depiction of gimbal lock*

Quaternions (complex numbers used to represent rotations) are not subject to gimbal lock but are more confusing to use without a good background in vector math (which is akin to voodoo for many of us). Quaternions use vectors to point in a given direction, so they can always rotate the objects on all axes simultaneously to get to the desired new vector with efficiency and stability.

On top of all of this, objects have both local (object space) and global (world space) rotation possibilities. To further complicate matters, transforms can be permanently changed, as in the editor, or temporarily changed, as in game animations.

When you see `transform.rotation` in a Unity script, it is referring to quaternions. When you see `transform.eulerAngles`, it is dealing with degrees. The numbers we see in the Inspector's Transform component for rotation show us `localEulerAngles`. Internally, Unity uses quaternions, but in scripting, you will see both used.

Bottom line: feel free to accept the navigation code as is (with no shame), so that you can continue with the game.

If you are still curious about rotation, try looking up "Quaternions and spatial rotation" in Wikipedia. While you are there, check out "gimbal" to see the origin of the term and a nice example of one in action.

Tweaking the Mouse Look

Before we change the existing `MouseLook` script, we need to define the required functionality. At the top of the functionality list is preventing it from working all the time. There's nothing worse than having the view spinning wildly as you try to position the cursor on an object in the scene for picking. The first thing the script will need is a conditional to tell it when it can be used. We will start with the basics and add to them.

1. Stop Play mode.
2. Select the `MouseLook` script in the Project view ➤Standard Assets ➤ Character Controllers ➤ Sources ➤ Scripts.
3. Duplicate the script using Ctrl+D (Cmd+D on the Mac).
4. Rename it **`MouseLookRestricted`**.

Note the `.cs` extension. The `MouseLook` script looks a bit different as it is written in C# instead of JavaScript. The syntax is close enough in many cases for making minor adjustments. All of the other scripts you will create or deal with in this project are JavaScript.

5. Replace the `MouseLook` script for both the First Person Controller and the Main Camera with the `new MouseLookRestricted` in the Inspector, through the browser.
6. Double-check to make sure the First Person Controller is using the Mouse X axis and the Main Camera is using the Mouse Y axis (Figure 5-15).

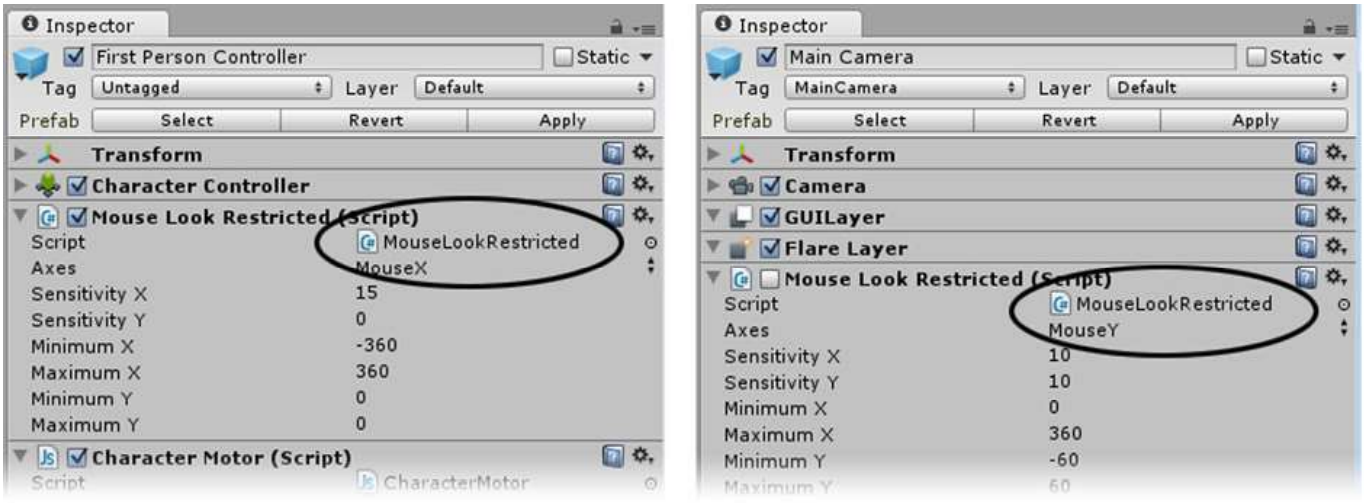

Figure 5-15. *The correct axes for the First Person Controller and Main Camera*

7. Turn on the `MouseLookRestricted script` on the Main Camera.
8. Open the `MouseLookRestricted` script in the script editor.
9. Inside the `Update` function (`void Update ()`), just beneath the open curly bracket, add the condition so that it looks like this:

```
void Update ()
   {
   // only do mouse look if right mouse button is down
   if (Input.GetMouseButton ( 1 ) ) {
```

You can use curly brackets inline, as with the line containing the `if` conditional, or on a different line altogether, such as with the void `Update ()` line. Feel free to use whichever makes the script easier to read. The important point to remember is that, just as with parentheses, if you have an open one, you must have a closed one to match.

■ **Tip** In the MonoDevelop script editor, you can find the closing curly bracket that goes with an opening bracket by clicking just before or just after the opening bracket. The corresponding closing bracket will be highlighted a pale gray.

In this case, we are putting the entire contents of the `Update` function within the conditional, so the closing curly bracket will go right above the `Update` function's closing curly bracket.

10. Add the closing curly bracket one line above the curly bracket that closes the `Update` function (the only thing left below that in the script is a short `Start` function).

The final four lines of the `Update` function are as follows:

```
   transform.localEulerAngles = new Vector3(-rotationY, transform.localEulerAngles.y, 0);
  }
  } // add this to close the conditional
}
```

11. Select the original contents, right-click, and select Indent Select from the right-click menu.

Indenting text inside the conditional or other types of code blocks will make reading easier and help ensure parentheses and curly brackets match.

12. Save the script.

You should get an error on the status line, as shown in Figure 5-16.

Figure 5-16. *One of many possible errors reported in the Console*

The status line tells us that the namespace `global::'` already contains a definition. Or if you are using a Mac, you may see "The class defined in script file named 'MouseLookRestricted' does not match the file name!" A quick look through the script shows that the script is referred to by name somewhere around lines 17 and 18.

13. Change those lines to reflect the new name.

```
[AddComponentMenu("Camera-Control/Mouse Look Restricted")]
public class MouseLookRestricted : MonoBehaviour {
```

14. Save the script.

The Console should clear itself of error messages.
Line 17 is interesting in that it shows how scripts can add to the Component menu for easy access.

15. From the Component menu ➤ Camera-Control, observe the new entry, Mouse Look Restricted, shown in Figure 5-17.

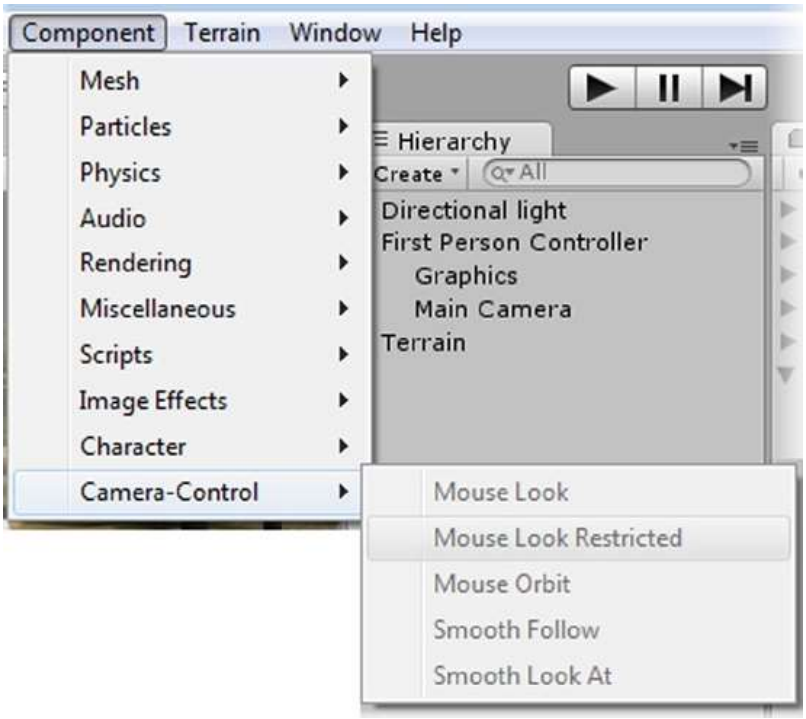

Figure 5-17. *The new menu item*

16. Stop Play mode.
17. Click Play and test.

■ **Tip** If you do a search in the Scripting Reference for `Input.GetMouseButton`, you will see that it "Returns whether the given mouse button is held down. /button/ values are 0 for left button, 1 for right button, 2 for the middle button."

You should now be able to mouse look when the right mouse button is held down . . . unless you don't have a right mouse button!

If you are developing on a Mac or just want your executable to run on either a Mac or a PC, you'll need to add another option for the conditional. This time, we'll select a key that must be held down to enable mouse look.

To do this, we will need to introduce the or operator in the condition that must be met. The *or* operator uses two pipe characters, || (the shift character of the \ key).

1. Change the `if` line to include a check for the left Shift key:

```
if (Input.GetMouseButton ( 1 ) || Input.GetKey ("left shift") ) {
```

Note that the key name is all lowercase characters. The Scripting Reference indicates that `Input.GetKey()` "Returns true while the user holds down the key identified by name." In contrast, `Input.GetKeyDown()` "Returns true during the frame the user starts pressing down the key identified by name." This function represents a single event and, therefore, is not the one you need to use.

2. Save the script and test.

Now, mouse-button-impaired users can also take advantage of mouse look.

■ **Tip** See Appendix B for the names of the keyboard keys for use in scripts or search for Input Manager in the Unity Manual.

If the user tries to use the right Shift key instead of left Shift, she will find that it doesn't work. You should probably add the right Shift to round out the choices.

3. Add the right Shift option:

```
if (Input.GetMouseButton (1) || Input.GetKey ("left shift") ||
    Input.GetKey ("right shift")){
```

4. Save the script and test.

By now you're probably getting some idea about controlling input. A nice touch for WASD users would be to allow automatic mouse look while navigating the scene. You know when the player is navigating the scene because of the keys mapped out for this. To cover all of the possibilities, the conditional would be something like the following:

```
if (Input.GetKey ("up") || Input.GetKey ("down") ||Input.GetKey  ("d") ||
   Input.GetKey ("s") || Input.GetKey ("w") || Input.GetKey ("a") || Input.GetKey
   ("left shift") || Input.GetKey ("right shift") || Input.GetMouseButton ( 1 ) ) {
```

Though this will certainly work as expected, it hardcodes the keys that must be used, some of which already have virtual counterparts in the Input Manager.

You have used `Input.GetAxis()` to get movement from the virtual buttons, but you will need something that returns a Boolean value (true or false) to check to see if our virtual buttons are being held down. That will be `Input.GetButton()`, which "Returns true while the virtual button identified by `buttonName` is held down."

Now you can simplify the condition by using the virtual buttons.

5. Change the `if` line to the following:

```
if (Input.GetMouseButton ( 1 ) || Input.GetKey ("left shift") ||
   Input.GetKey ("right shift") || Input.GetButton("Horizontal") ||
   Input.GetButton("Vertical") ) {
```

■ **Tip** Make sure there is a `||` between each of the conditions and the code is all on one line, or you will get several errors.

---

6. Save the script and test.

Now, when the player is navigating the scene, the mouse look works automatically, as with regular shooter controls, but as soon as the player stops, the cursor can be moved around the scene without the world spinning dizzily.

At this point, it may have occurred to you that if you could create a virtual button for any keys that need to be held down when the user is not navigating, you could simplify things even more and consider yourself quite clever in the process. Let's create a new virtual button called ML Enable (Mouse Look Enable).

1. Stop Play mode.

■ **Tip** As you experimented with the code in the `MouseLookRestricted` script, you were able to keep Unity in Play mode because saving the script re-compiled it, and the changes were picked up immediately in the `Update` function. If you had added changes to the `Start` function, you would have needed to restart Unity after each save so the new content could be used. `Awake` and `Start` are evaluated only at startup.

---

2. From Edit ➤ Project Settings, select Input.
3. Increase the array size to **19**.
4. Open the new Turn duplicate and name it **ML Enable**.
5. Add the shift keys and right mouse button into the Negative, Positive, and Alt Negative Button fields (see Figure 5-18).

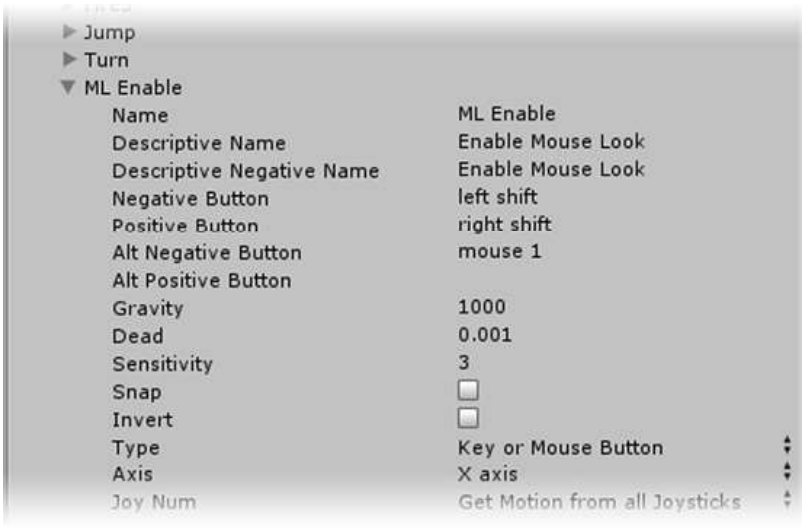

Figure 5-18. *The new ML Enable virtual button*

6. Set the Descriptive Name to **Enable Mouse Look**.

Because the settings were inherited from the previous entry, yours may be different if you tweaked any of them. Now you can simplify the `if` conditional in the `MouseLookRestricted` script. At the same time,

7. Change the line to the following:

```
if (Input.GetButton ("ML Enable") || Input.GetButton("Horizontal") ||
    Input.GetButton("Vertical") ) {
```

8. Save the script.
9. Click Play and test.
10. Save the project so you don't lose the new input assignment.

The functionality should remain the same.

As a final refinement, you can allow mouse look only on the X axis when the left or right arrows are pressed, so the player can mouse look up and down. The tricky part about this functionality is that the same script is used on the camera for X-axis rotation and on the First Person Controller for the Y-axis rotation. Because the left and right arrow keys rotate the first person on the Y axis, you must avoid multiple, possibly conflicting, input on the Y.

The script uses a variable named axes that lets the author choose the axis to rotate. Of the three choices, `RotationAxes.MouseXAndY`, `RotationAxes.MouseX`, and `RotationAxes.MouseY`, the only one that does not allow Y to rotate is `RotationAxes.MouseX`. The condition that must be met if the mouse look is allowed will be if either the Turn virtual button is pressed and the value of the axes variable is `RotationAxes.MouseX`. To ensure that the entire condition is evaluated together, you will wrap it in parentheses. The *and* in the conditional is written as a double ampersand, &&.

11. Finally, change the `if` line to the following:

```
if (Input. GetButton ("ML Enable") || Input.GetButton("Horizontal") ||
    Input.GetButton("Vertical") ||
   (Input.GetButton("Turn") && axes == RotationAxes.MouseY) ) {
```

12. Save the script.
13. Click Play and test the functionality.

The navigation for the game is now ready to use.

Now that you've altered a few scripts, you should start organizing your scripts for the game before they get out of hand.

1. Create a new folder in the Project view.
2. Name it **Adventure Scripts**.
3. From the Standard Assets ➤ Character Controllers ➤Sources ➤ Scripts folder, move the `FPAdventurerInputController.js` script into the new folder.

The `MouseLookRestricted` script is C# rather than JavaScript and needs to be left where it is. Unity compiles scripts according to where they are located, which can affect access to the scripts by other scripts. JavaScript is more forgiving, so you can move the other scripts where you please. All of the remaining scripts you'll work with or create are JavaScript.

4. Save the scene and save the project.

■ **Tip** You added three keys to be represented by your virtual ML Enable button, using three of the four button slots in the preset. If you need more than four keys or mouse buttons, you could add another preset using the same name.

At this point, you need to make an executive decision—a design decision. The other navigation inputs are handled only if the Collider is on the ground. If you want the player to be able to turn only while on the ground, you must add your code inside the `if (grounded)` conditional. If, however, you want the player to be able to "turn his head," or even "twist his body around," while in midair, the code should not be inside the conditional. Because the purpose of the game is entertainment, let's go with the latter.

Fun with Platforms

As a reward for getting through the navigation code, let's have a bit of fun using one of the features of the Character Motor script—the Platform functionality. Unless you are an experienced platform-jumper player, the thought of trying to navigate several fast-moving platforms may be intimidating. For an adventure game, however, you might want to make use of platforms that aren't challenging to use but add interest to navigating the environment. Many graphical adventure games of the pre-rendered era featured rendered sequences where the player was taken on wild rides through tunnels or on skyways. The book's game will feature a simple raft to allow the player access to a grotto behind a waterfall.

1. Select the First Person Controller.
2. Set its Y Rotation to **270** in the Inspector.

■ **Tip** This last step is optional, but will make the next couple of sections easier to follow. If you prefer to keep your First Person Controller looking in a different direction, just try to keep your scene looking similar to the screen shots.

---

3. Select the Main Camera from the First Person Controller object.
4. Use Align View to Selected to set the Scene view.
5. Create a Cube and name it **Platform**.
6. Change its Scale to **2** on the X, **0.3** on the Y, and **6** on the Z (see Figure 5-19).

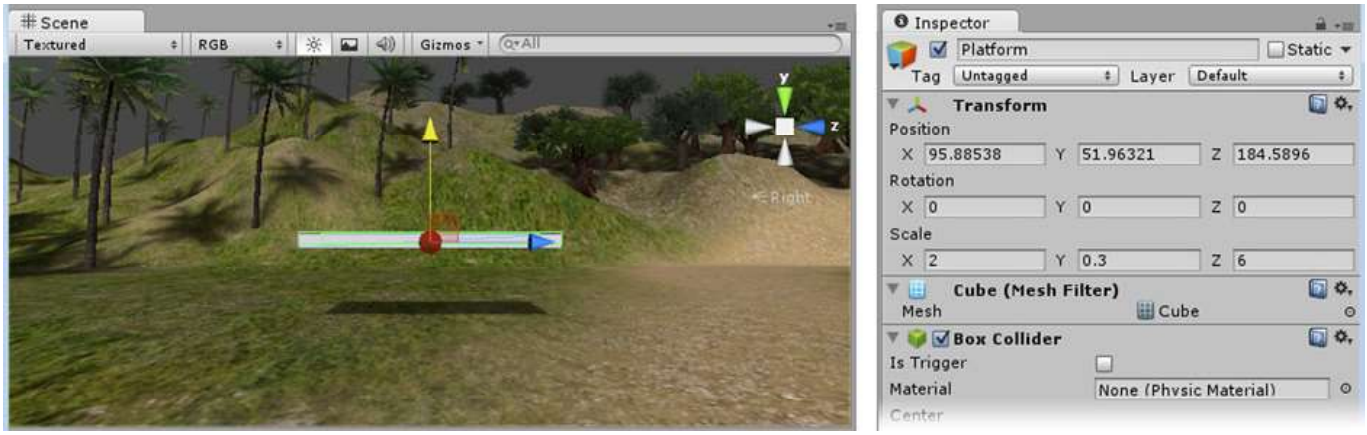

Figure 5-19. *The new Platform settings*

7. Move it down, so it just intersects with the ground.
8. Add a material, such as TestBark, to it (see Figure 5-20).

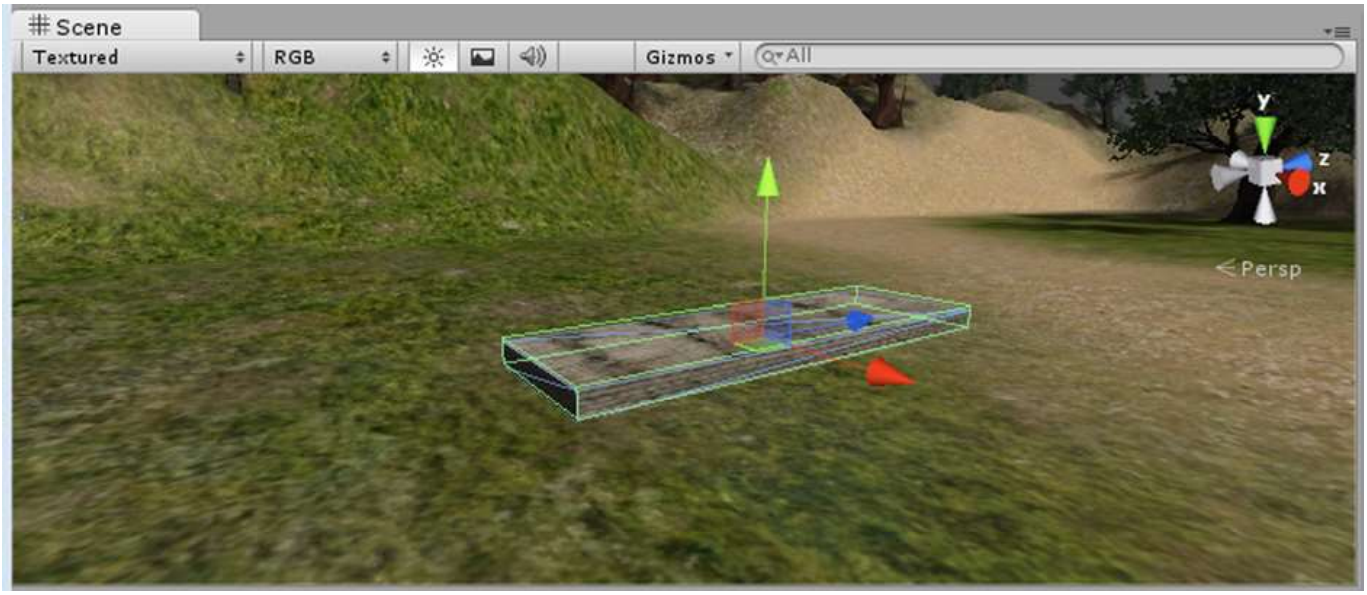

Figure 5-20. *The new platform*

Next, you will need to build a simple script to move the platform. You could feed numbers into it, but it will be easier to set up if you just create a couple of target objects to move between. I have borrowed the code from a script in an old Unity tutorial.

9. Select the Adventure Scripts folder.
10. Create a new JavaScript in it.
11. Name the script **PlatformMover**.
12. Open the script in the editor.

13. Replace the default functions with the following:

```
var targetA : GameObject;
var targetB : GameObject;

var speed : float = 0.1;

function FixedUpdate () {
        var weight = Mathf.Cos(Time.time * speed * 2 * Mathf.PI) * 0.5 + 0.5;
        transform.position = targetA.transform.position * weight +
                        targetB.transform.position * (1-weight);
}
```

■ **Tip** Just in case the math for that last bit of code made your head hurt, you should know it originated from the 2D Gameplay tutorial. While it is currently not available through the Asset Store, a search of the forums should help you find a link to it. As recommended at the start, you can find all manner of useful code in tutorials for all different genres. Don't be afraid to copy, paste, and tweak—you never know what will prove useful in your game!

---

14. Save the script.

Plan Ahead

In the early stages of prototyping your game or even just testing techniques for functionality, it is tempting to go with the simplest solution. When you are dealing with animating objects, however, it is well worth a couple of extra steps to create a GameObject to hold the mesh and then put the animation on it instead of the mesh. This gives you the freedom to swap out early proxy objects with the final versions or add more objects to the group without having to redo the animation or triggers.

1. Focus the scene on the Platform by double-clicking the Platform in the Hierarchy view.
2. Create an Empty GameObject; it should be created in the same place as the Platform.

If the GameObject is not in the same place (especially the Y axis), you can copy the x, y, and z coordinates from the Platform's Transform component at the top of the Inspector to the new Empty GameObject's x, y and z.

3. Name it **Platform Group**.
4. Drag the `PlatformMover script` from the Project view onto the Platform Group object.
5. Drag the Platform into the Platform Group.

This script uses two target objects for its transforms. The variables that will hold them are defined as type GameObject. After you create a couple of targets, you'll have to drag the target objects onto the variables' value fields in the Inspector (see Figure 5-21).

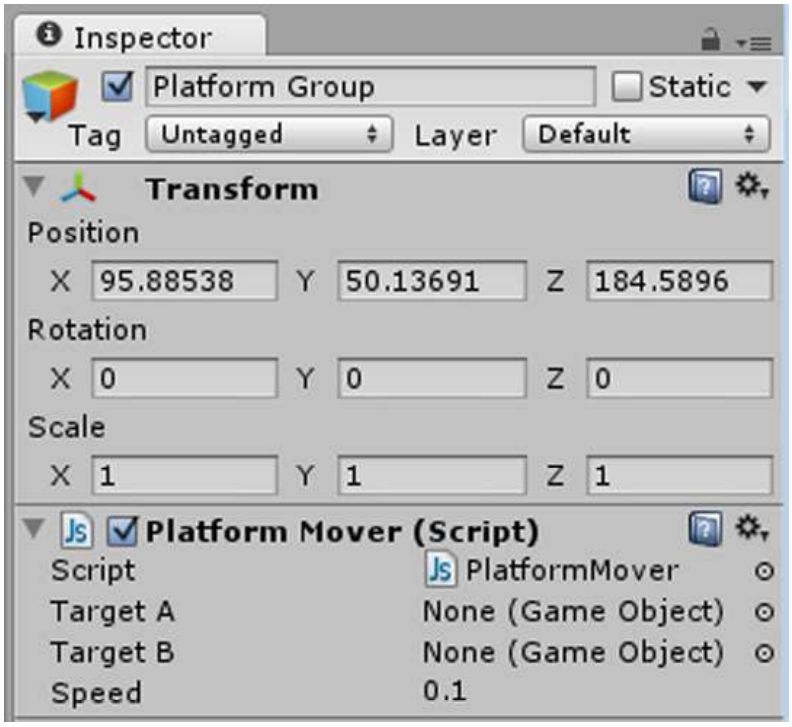

Figure 5-21. *The Target A and B variables in the Inspector, waiting for GameObjects*

The Speed variable allows you to adjust the speed with which the platform will move between the targets.

Because you want the speed to be consistent regardless of frame rate or computers, it uses a `FixedUpdate` function.

Inside the `FixedUpdate` function, a cosine function is used in conjunction with the speed and target positions to move the platform between the targets (see Figure 5-22)

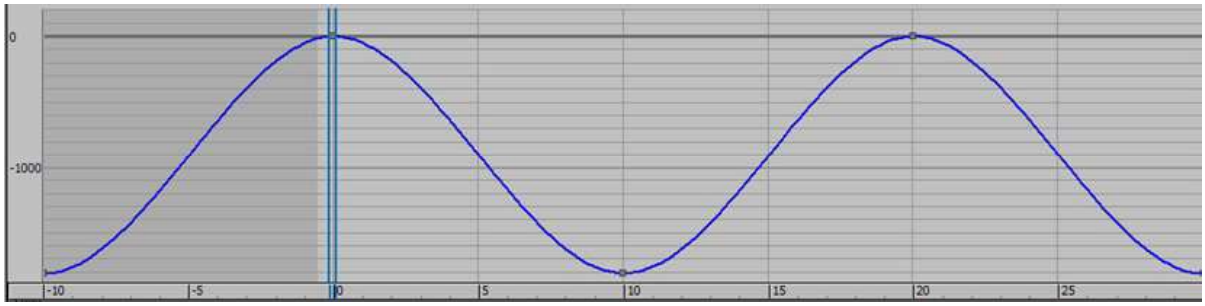

Figure 5-22. *A cosine function curve. With time on the horizontal and position on the vertical, you can see how the platform slows to a stop before reversing direction*

Theoretically, all you have to use for your target objects are a couple of empty GameObjects. The problem is that it would be difficult to get them positioned correctly without any geometry. As an alternative, you can clone the original platform, position it, and then remove its renderer and collider components.

6. Select the Platform object.
7. Use Ctrl+D (Cmd+D on the Mac) to make a duplicate of it.
8. Name it **PlatformMesh** and drag it out of the Platform Group.
9. Locate its Box Collider component in the Inspector.
10. Right-click the component label and choose Remove Component, as shown in Figure 5-23.

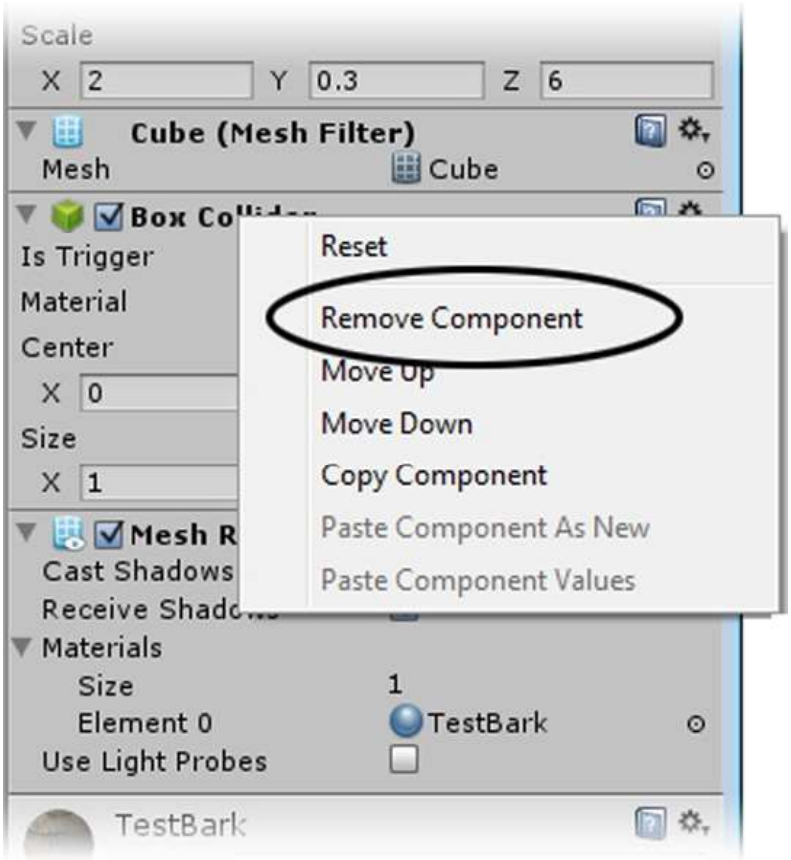

Figure 5-23. *Remove Component*

11. In the Mesh Renderer component, uncheck Cast Shadows and Receive Shadows.

At this point, you could clone the platform and finish the platform functionality, but in the spirit of getting in the habit of using an important Unity concept, let's create a prefab for extended use.

1. Create a new folder.
2. Name it **Adventure Prefabs**.
3. Select the new folder.
4. Right-click on top of it and create a new prefab.

5. Name the prefab **Platform Target**.
6. Drag the PlatformMesh object from the Hierarchy view onto the Platform Target prefab in the Project view.

The PlatformMesh is now an instance of the Platform Target prefab, and its white cube icon has turned blue.

7. Rename the PlatformMesh to **Target A**.

Because it is already in the same place as the original platform, you can leave it in place and use it as one of the target positions.

8. Drag the Platform Target prefab from the Project view into the Hierarchy view.
9. Name this one **Target B**.
10. In the Scene view, move Target B away from the platform in the Z direction (see Figure 5-24).

Figure 5-24. *The two target platforms (A is in place, over the original platform.)*

11. Select the Platform Group object.
12. Drag Target A and Target B from the Hierarchy view onto their target parameters in the Inspector, as shown in Figure 5-25.

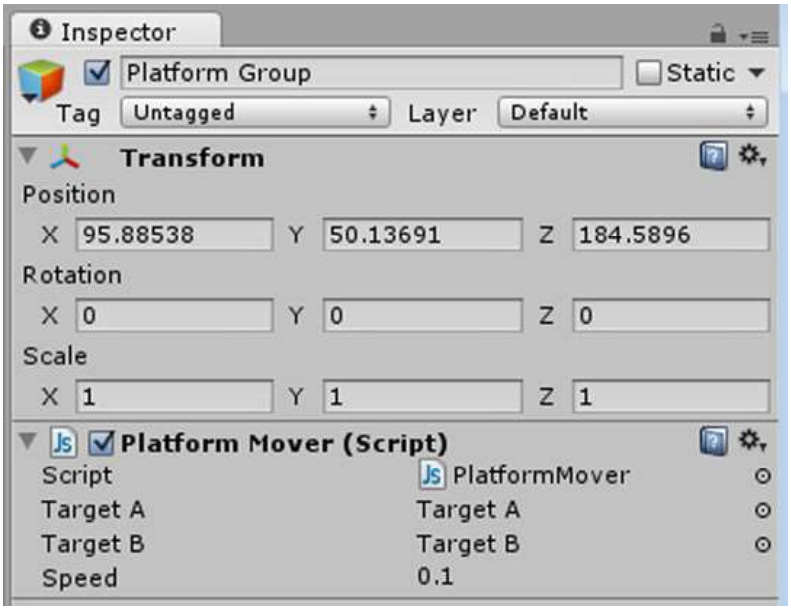

Figure 5-25. *The PlatformMover script with both targets loaded*

13. Click Play.

The PlatformGroup moves smoothly between the two targets.

Having positioned the targets, we can now turn off their MeshRenderers, so they will not be rendered in the scene. One of the beauties of the prefab is that by turning off *its* Mesh Renderer, it will be turned off in both of its children.

1. Stop Play mode.
2. Select the PlatformTarget prefab in the Project view.
3. Uncheck the Mesh Renderer component (see Figure 5-26).

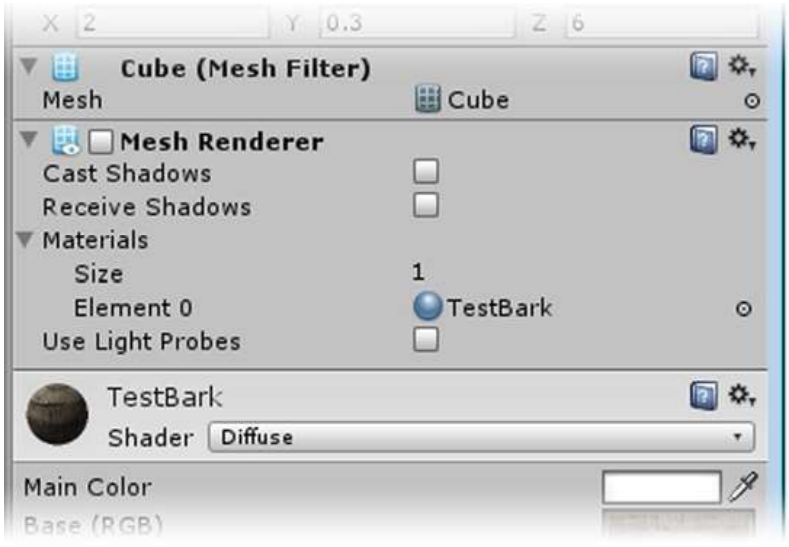

Figure 5-26. *Disabling the Mesh Renderer*

The two target platforms are no longer rendered in the scene.

4. Click Play.

5. Drive the First Person Controller onto the platform.

If your platform is low enough to drive onto, the First Person Controller gets taken for a ride. If not, use the spacebar to jump up onto your platform, or lower it so the First Person Controller can climb aboard (see Figure 5-27).

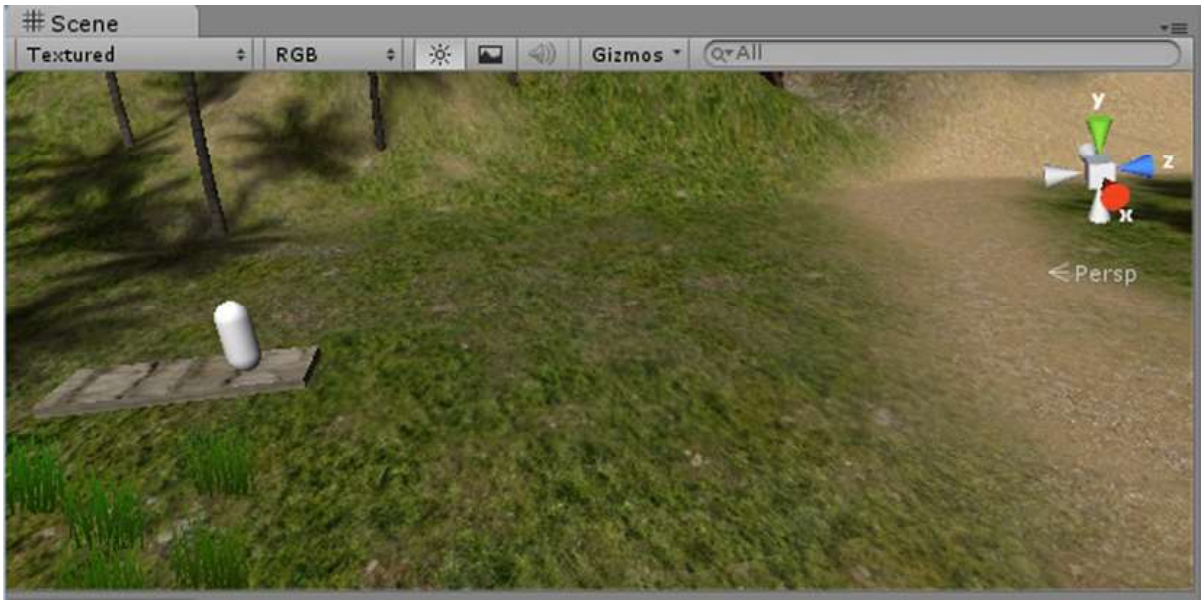

Figure 5-27. *Going for a ride*

The default platform functionality is only a starting point for serious platform jumpers. We don't need more than the basics for our game, so we just need to be aware of what it can and can't do. Let's investigate.

6. Drive off the platform and position the First Person Controller in the path of the platform.

Depending on your platform height, the platform may push the First Person Controller out of the way, or it may just go through it, occasionally picking it up and taking it for a ride. The important thing to note here is that the results are not consistent. You should investigate functionality fully, so you know its pros and cons before deciding to add it to your game and spending time on asset creation.

Let's see what happens if the platform is taller.

7. While still in Play mode, select the Platform object and change its Y Scale to **1**.

Half of the platform goes lower, but there's still have about half a meter aboveground.

8. Move into the path of the moving platform again.

This time, the platform passes through the First Person Controller every time, as shown in Figure 5-28.

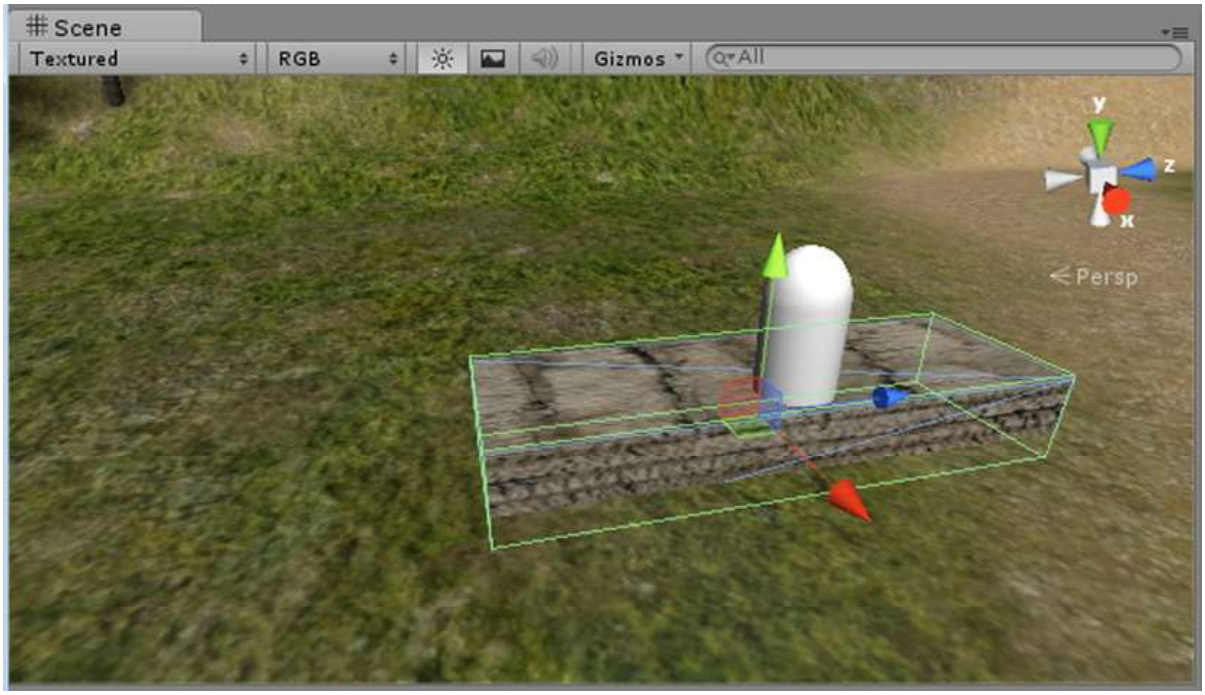

Figure 5-28. *The taller platform ignores the First Person Controller*

9. As the platform approaches again, press the spacebar to jump on to it as it gets close.

You should now be riding the platform again.

If you drive at the taller platform, the First Person Controller might be stopped, but unless it is in motion, collision is not detected. While there are things you can do to overcome this effect, they are generally costly and not within the scope of this adventure game.

If you do want to use a moving platform in your game, you'll need to set up a system to prevent the player from getting near the platform while it is in motion. This could be a toll gate or barricade that the player can only enter at certain times. It could require the player to insert a token into a receptacle to activate the platform. You'd also need a means of preventing the player from jumping off the platform, which, of course, you'd have to disable to allow him to get off and on at the correct times. All of this is possible and would indeed increase player enjoyment of the game, but you should always be aware of the work involved.

1. Stop Play mode.
2. Select the PlatformTarget object in the Project view and turn its Mesh Renderer back on.
3. Select the Target B object and move it up about 10 meters in the Scene view.
4. Turn off the PlatformTarget's Mesh Renderer.
5. Click Play.
6. Take a ride on the lifting platform (see Figure 5-29).

Figure 5-29. *View from the top*

The First Person Controller is carried up to the new vantage point.

7. While you are up near the top, drive off the platform.

The First Person Controller falls gracefully to the ground.

Clearly, you will need to design carefully to deal with the various aspects of the functionality. While you might be happy to let the player jump off a funicular while it is in motion, you'd want to prevent him from landing where it will be able to pass through him on the way back down. As this is a more complicated problem to solve, you can leave it for now and tackle walls instead.

8. Stop Play mode.
9. Save your scene and save the project.

Collision Walls

In traveling around your terrain, you've had firsthand experience with the topography of both the terrain and the moving platform. You've also had a run-in or two with the collider on the banyan tree. You need colliders on anything you don't want the player to be able to pass through. Unity can generate Mesh Colliders for imported meshes, such as buildings, and offers several primitive colliders as well for more efficient collision detection.

As with the platform, walls in Unity have some peculiarities that need to be investigated.

1. Select the First Person Controller.
2. Use Align View to Selected from the GameObject menu.
3. Create a cube and position it in front of the First Person Controller but closer than the platform.
4. Name it **Wall.**
5. Set its Scale parameters to **0.3** for X, **5** for Y, and **6** for Z.
6. Move the wall up in the Y direction, so only a small bit intersects the ground.

7. Rotate it about **5** degrees on the Y, so it does not face the viewport exactly and the First Person Controller will approach it at a slight angle.
8. Drag the TestBark material onto it.
9. Zoom back in the viewport until you can see quite a bit above it (see Figure 5-30).

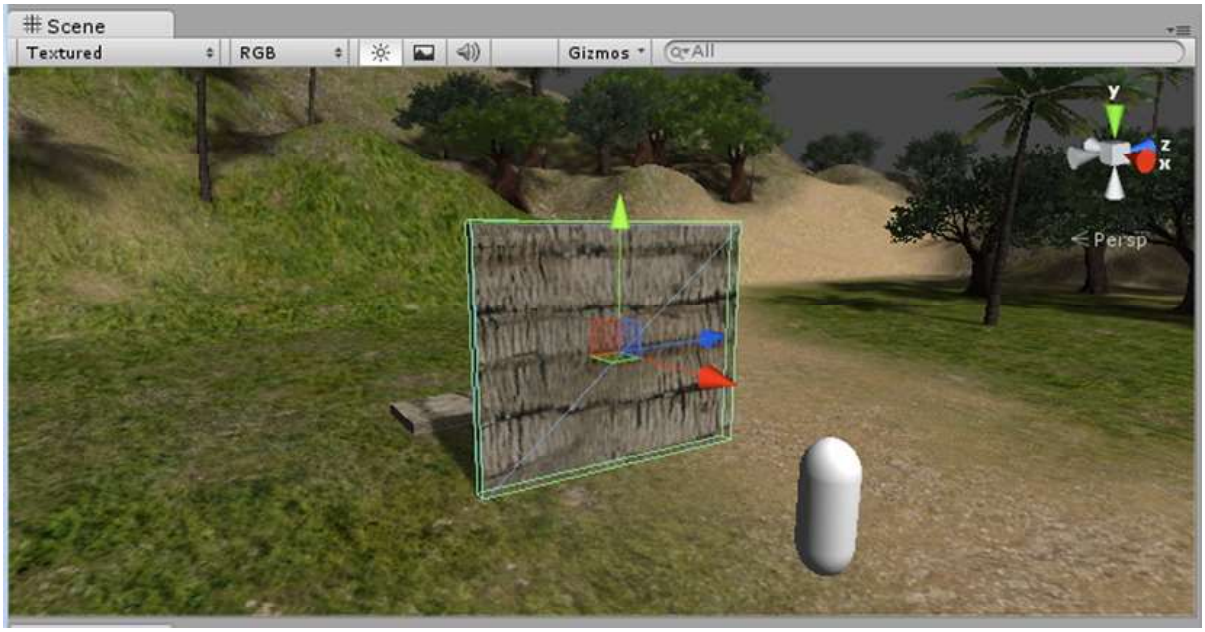

Figure 5-30. *The wall object and adjusted Scene view*

The green bounding box you see in the Scene view is the collider that was automatically generated when the cube was created. In the Inspector, you'll see that it is a simple Box Collider.

10. Click Play and drive into the wall.

The First Person Controller is stopped.

11. Use the spacebar to jump as you try to move forward.

It does not get you any farther.

12. Stop Play mode.
13. In the Inspector, rotate the wall to **20** degrees on its Z axis.
14. Click Play and drive at the wall again.

The First Person Controller may slide left or right along the wall as you continue to try to go forward, but it does not climb the wall.

15. Once again, use the spacebar to jump while trying to move forward.

This time, you can easily scale the wall and go over the top if you move at a bit of an angle.

While this may be quite entertaining, it means that your players can probably get to places you'd prefer they do not. You could turn off jumping altogether in the Motor script, but it does make navigation more interesting when allowed.

16. Stop Play mode.
17. Select the First Person Controller.
18. In the Inspector, in the Jumping section of the Character Motor component, set the Base Height to **0.5** and the Extra Height to **0**.
19. Click Play and attempt to climb or jump the wall again.

It is more difficult, but tenacious gamers may discover that moving sideways a bit as they jump and move forward will eventually allow them to reach the top.

■ **Tip** Always assume that if something can be done, it will be done, regardless of how pointless it seems to you as the author of the game.

So far, you know that a player will not be able to scale a fully perpendicular wall. Anything with even a small degree of slope, however, is subject to abuse. Because Unity does not differentiate between floor and wall collision surfaces, you will need a way to prevent unwanted climbing of some buildings or other artifacts. Rather than getting deep into the Character Motor code and worrying about up vectors and the like, you will take the artist's way out and simply block access with a nonrendering planar object.

1. Stop Play mode.
2. Again, in the Character Motor component, temporarily set the Base Height to **3** and the Extra Height back to **1**.
3. Create a Plane object and position it about three-quarters of the way up the wall.
4. Adjust its size and position to make it an effective barrier, as shown in Figure 5-31.

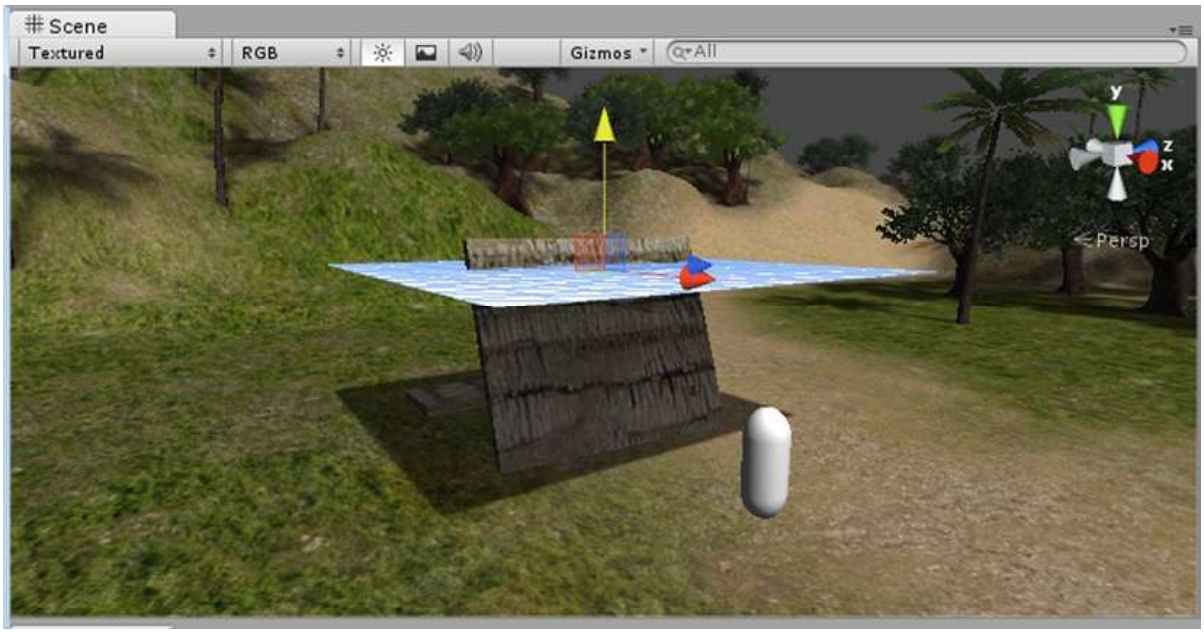

Figure 5-31. *The plane in position as a ceiling*

5. Click Play and test the new barricade.

The First Person Controller still gets through it.

6. Select the plane.
7. Rotate the Scene view until you can see underneath the plane (see Figure 5-32).

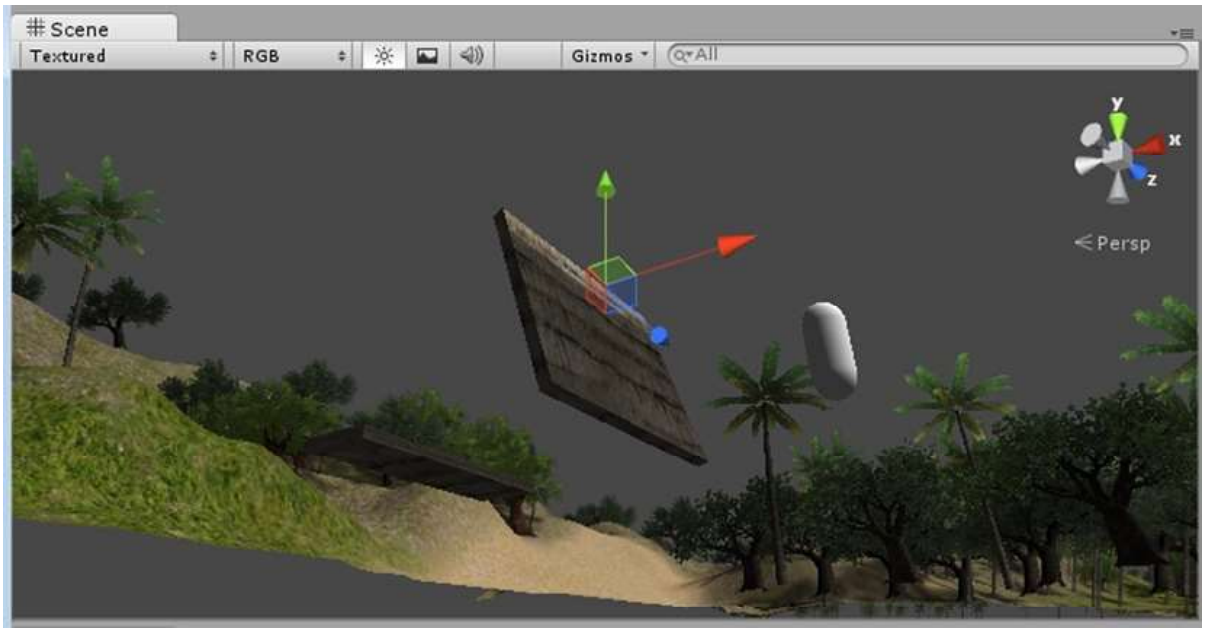

Figure 5-32. *The underside of the plane*

And nothing shows!

In the Inspector, you'll see that the plane has a Mesh Collider. Because there are no faces on the backside of the plane, no collision detection is taking place. The other collider types (Box, Sphere, and Capsule) use only the primitive shape of their collider and are not dependent on the object mesh for collision.

It would seem you could solve the problem by flipping the plane over.

8. Set the plane's X rotation to **180**, as shown in Figure 5-33.

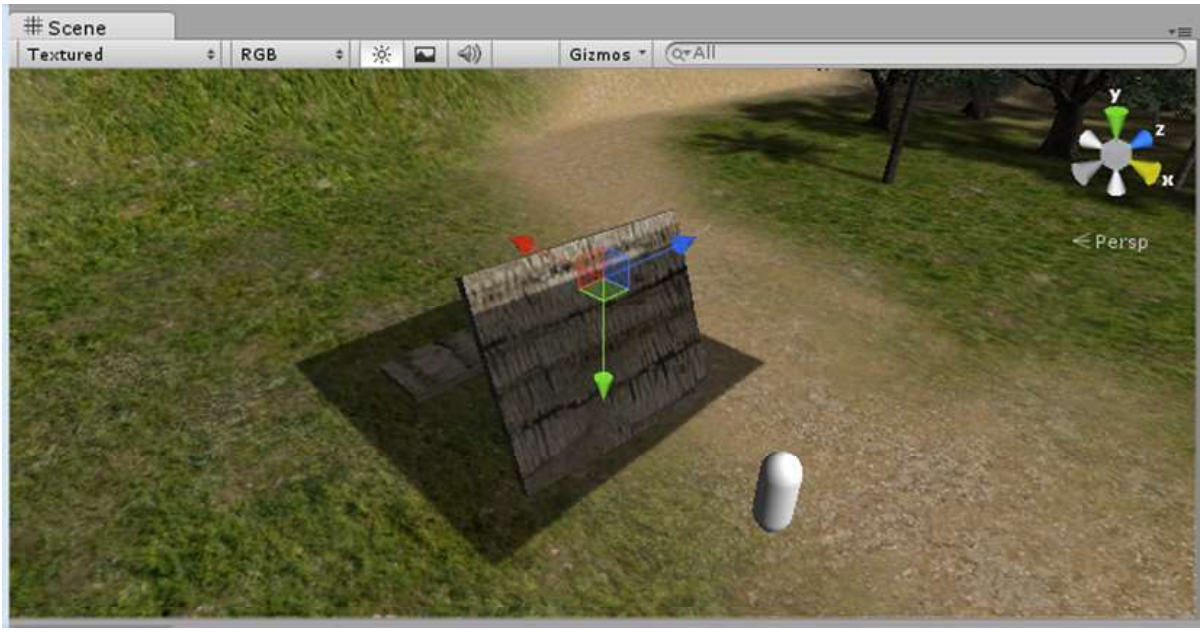

Figure 5-33. *The plane flipped upside down. Note the orientation of the Transform Gizmo and also, if you are using Unity Pro, that shadows are cast regardless of the plane's orientation*

9. Make sure the coordinate system is set to Local.
10. Click Play and try to jump over the wall again.

This time, the First Person Controller hits the ceiling and falls back down.

Since you can now see the plane from the underside, you need to either turn off its Mesh Renderer or remove that component altogether.

11. Stop Play mode.
12. Select the Plane.
13. Right-click over the Mesh Renderer component and Remove it.

The plane is no longer drawn in the Game view, but the Mesh Collider is drawn in the Scene view (see Figure 5-34).

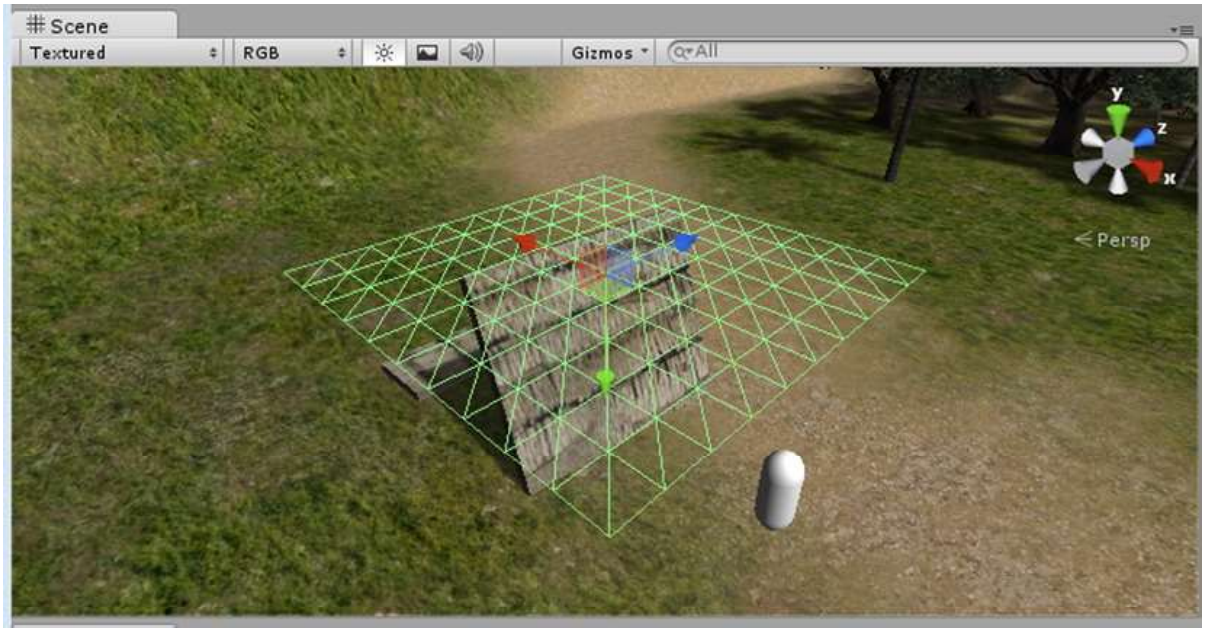

Figure 5-34. *The Plane's Mesh Collider showing now that the Mesh Renderer has been removed*

Another experiment worth trying is to see what happens to the First Person Controller when the platform rises through it.

14. Move the plane so it is in the path of the rising platform.

15. Click Play and jump onto the platform.

When the First Person Controller gets to the plane, it is pushed down through the platform. Apparently, the First Person Controller is inheriting the movement from the platform, and collision detection against the plane is registered.

A final test might provide the means for keeping the player from jumping off a moving platform.

16. Set the Z rotation of the plane to **90** degrees in the Inspector (see Figure 5-35).

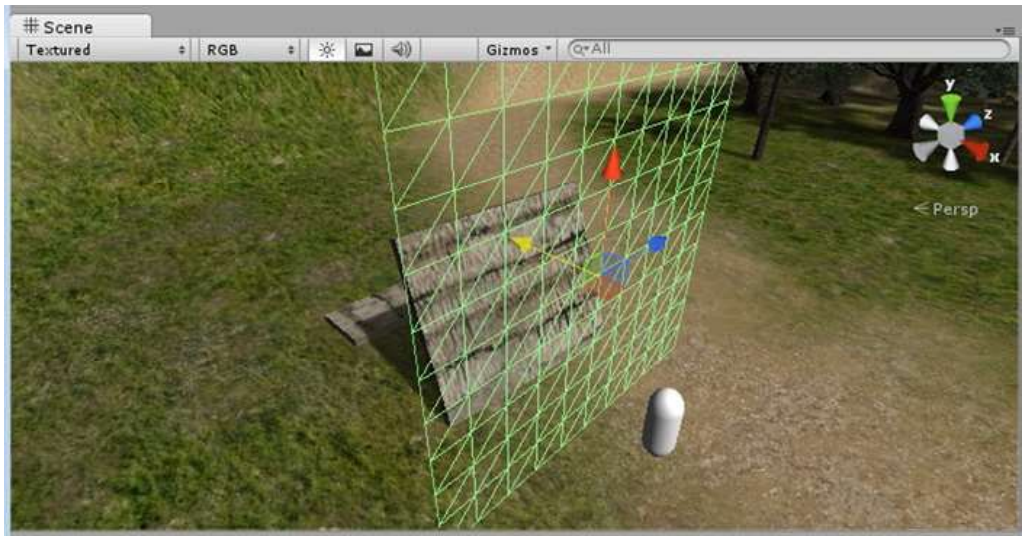

Figure 5-35. *The Plane as an invisible wall*

17. Click Play and try to drive through the invisible plane.

You will find that you can pass through it from the back side but not the front side.

If you surround the platform with planes for collision, you could allow the player to get onto the platform, but not off, thereby preventing him from jumping off midway.

The problem then becomes how to let him disembark when he reaches the plane's destination.

18. Stop Play mode.
19. Select the plane.
20. In the Inspector, in the Plane's Mesh Collider component, check Is Trigger.
21. Click Play.
22. Now try going through the plane from both sides.

The collider no longer acts as a barrier.

Let's create a simple script to toggle the `IsTrigger` variable off and on with a key press. Before assuming that a particular functionality will work, you should always test first.

23. Select the My Scripts folder.
24. Create a new JavaScript.
25. Name it **`ToggleIsTrigger`**.
26. Open it in the script editor.

In this script, you will watch for a key down in the `Update` function. When one is detected, you will call another function that you will create—a user-defined function. If you think that you'll use a particular procedure over and over and call it from different places or objects, it may be worth putting in its own function. An example might be a three-way light switch. Three separate switches can toggle the light off and on, but only the light contains the code to handle the functionality. Each of the switches would call the light's on/off function. In the case of the `IsTrigger` toggle, you might test it with a key press, but later you might have to press a button to toggle it or even let it be triggered by some other event. You will name the function `ToggleTrigger`.

27. Change the `Update` function as follows:

```
function Update () {

   // call the ToggleTrigger function if the player presses the t key
   if (Input.GetKeyDown ("t")) ToggleTrigger();

}
```

■ **Tip** When a method or object-specific function returns a Boolean value, true or false, there is no need use the `==` operator. In the conditional you just wrote, the `if(Input.GetKeyDown("t"))` could also be written as `if(Input.GetKeyDown("t") == true)` and produce the same results.

Now, let's create the `ToggleTrigger` function. You'll need to check the state of the collider's `isTrigger` variable and then reset it, depending on the result. At its simplest, you could write it in just a few lines.

28. Add the following somewhere below the `Update` function:

```
function ToggleTrigger () {

   if (collider.isTrigger == true) collider.isTrigger = false;
   else collider.isTrigger = true;

}
```

When you are first testing code and functionality, it's worth adding some print statements, so you can see what's happening. Note that the `if conditional` doesn't require curly brackets as long as only one command is given, if the condition evaluates to true (the `if`), or only one is specified, when the condition does not evaluate to true (the else).

Think of the `if` and else as hands that can each hold only one object. The curly brackets are like a shopping bag—the hands still hold a single object (the shopping bag), but the bag can contain lots of objects.

29. Change the `ToggleTrigger` function to include a couple of print statements:

```
function ToggleTrigger () {

        if (collider.isTrigger == true) {
                collider.isTrigger = false;
                print ("Closed");
        }
        else {
                collider.isTrigger = true;
                print ("Opened");
        }
}
```

30. Save the script.

■ **Tip** Function names always start with a capital letter. Like variable names, function names may not contain spaces, start with numbers or special characters, or be reserved words. To make the names easier to read, you will often see underscores instead of spaces, or the first letter of each word capitalized.

---

31. Drag the `ToggleIsTrigger` script onto the Plane object.
32. Click Play.
33. Select the Plane, then click in the Game window to change focus.
34. Watch its Is Trigger parameter in the Inspector.
35. Press the t key to toggle the Is Trigger off and on. Try driving forward and backward through the invisible plane.

You should be able to drive through from both sides when Is Trigger is turned on, but only from the back side if Is Trigger is off.

So far, you have used the IsTrigger parameter for one of its side effects—to turn off physical collision. Its intended purpose is to allow an object, such as the First Person Controller, to trigger an event or action by passing into or out of a (frequently) non-rendered object containing a collider.

Returning to our hypothetical light example, picture a long hallway with sensor lights. Each light has a zone that when entered turns the light on and when exited turns the light off. The zones are Box colliders, and you can use the `OnTriggerEnter` and `OnTriggerExit` functions to trap the events and send the message to the light.

Without going to the trouble of creating collision walls for the platform, you can still get a feel for how you can use a collider to manipulate the isTrigger parameter of other colliders. Let's set up a GameObject that will turn the isTrigger parameter of the plane off and on when the platform enters and exits its collider.

36. Stop Play mode.

37. Deactivate the Wall object by unchecking the box at the top of the Inspector (see Figure 5-36).

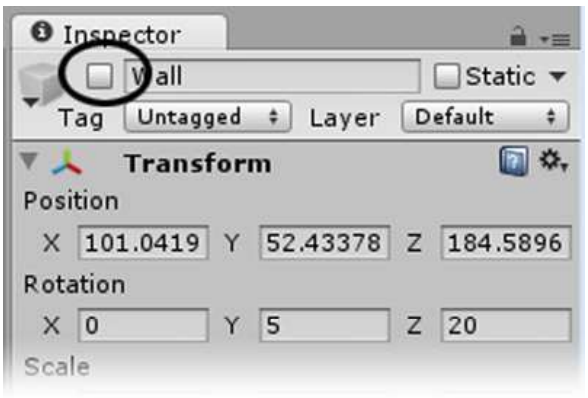

Figure 5-36. *The activate checkbox for the Wall object*

■ **Tip** Deactivating an object not only prevents it from being drawn in the scene, it also disables its collider and any other components. Note that the object's name is grayed out in the Hierarchy. You will make good use of the Active property of GameObjects when you start adding interactive objects as the game progresses.

---

1. Select the PlatformTarget in the Project view and turn on the Mesh Renderer.
2. Create an Empty GameObject.
3. Name it **TheTrigger**.
4. With TheTrigger still selected, choose Physics from the Components menu and add a Box Collider.
5. Scale the collider by using the X, Y, and Z Size parameters in the Box Collider component in the Inspector.
6. Position the collider so it is high enough off the ground to prevent the First Person Controller from running into it and so it is stretched between the two platform targets as shown in Figure 5-37.

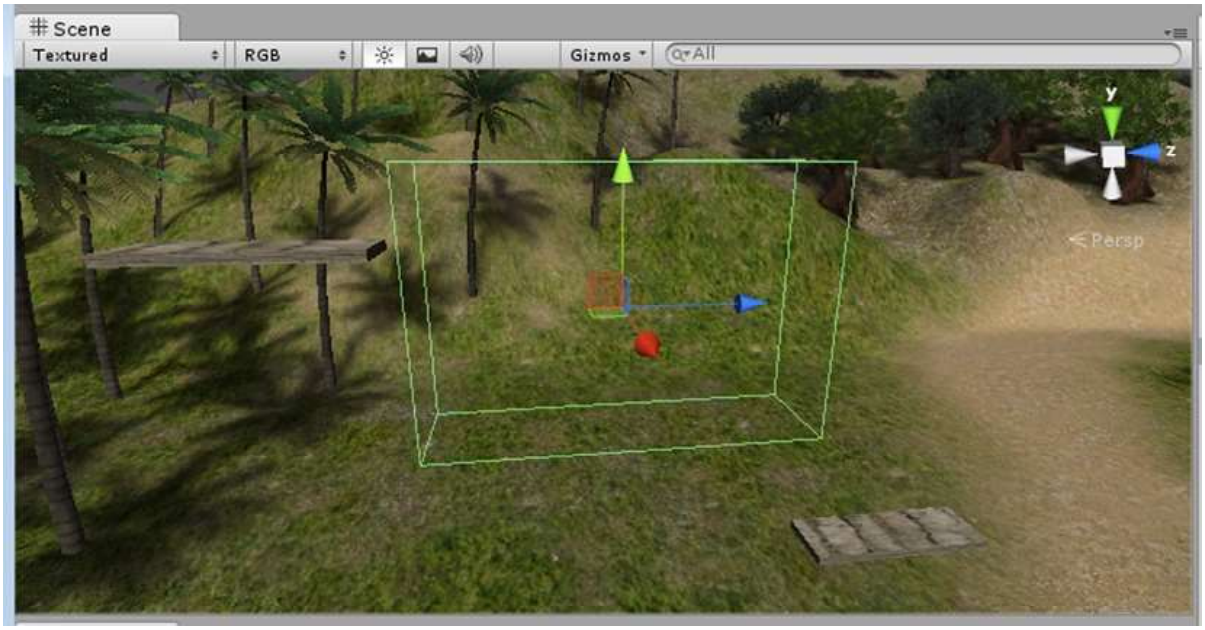

Figure 5-37. *The Trigger's collider sized and positioned*

Make sure to check the collider in a Top view for alignment and position (see Figure 5-38).

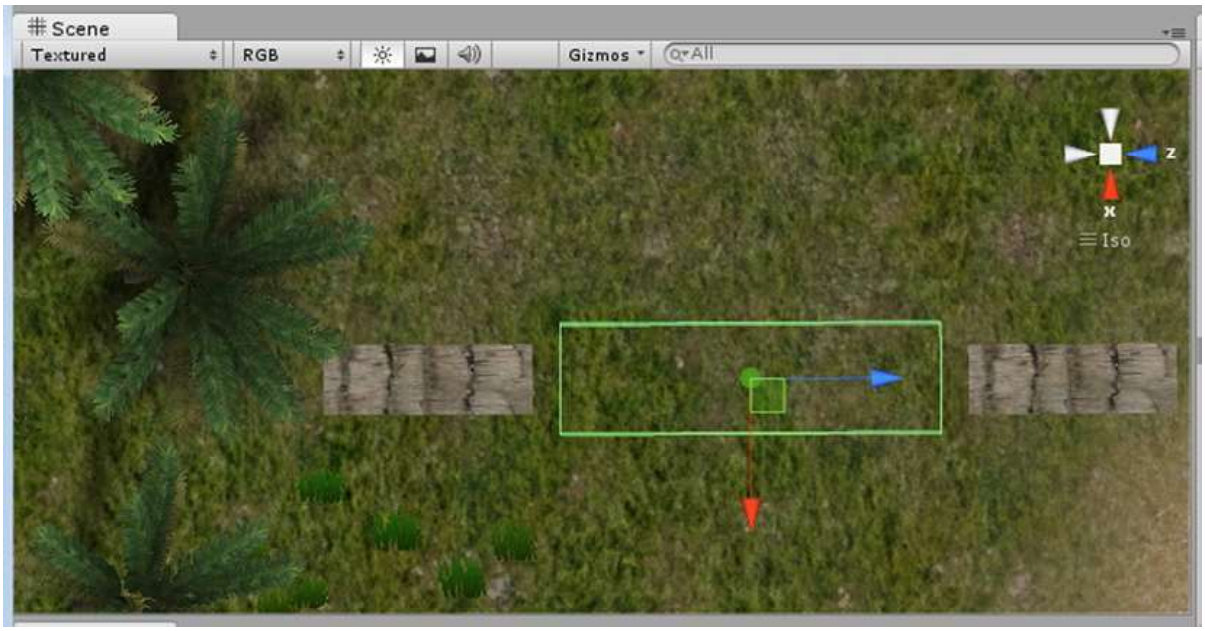

Figure 5-38. *The Top view of the collider*

7. Make sure TheTrigger is selected, so you can see the collider in the Scene view.
8. Click Play.
9. Move the First Person Controller onto the moving platform.

The First Person Controller is pushed off the platform as the platform moves into the collider (see Figure 5-39).

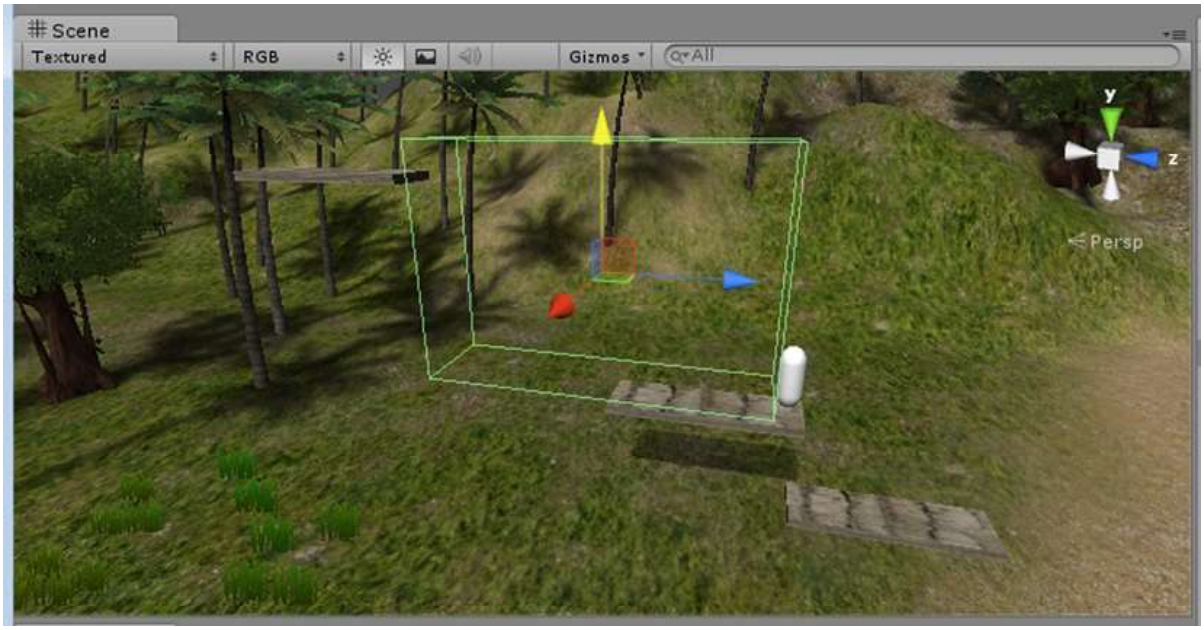

Figure 5-39. *First Person Controller being pushed off the platform*

10. Stop Play mode.
11. Check the Is Trigger parameter in the Inspector.
12. Click Play and test.

The First Person Controller no longer gets pushed off the platform (see Figure 5-40).

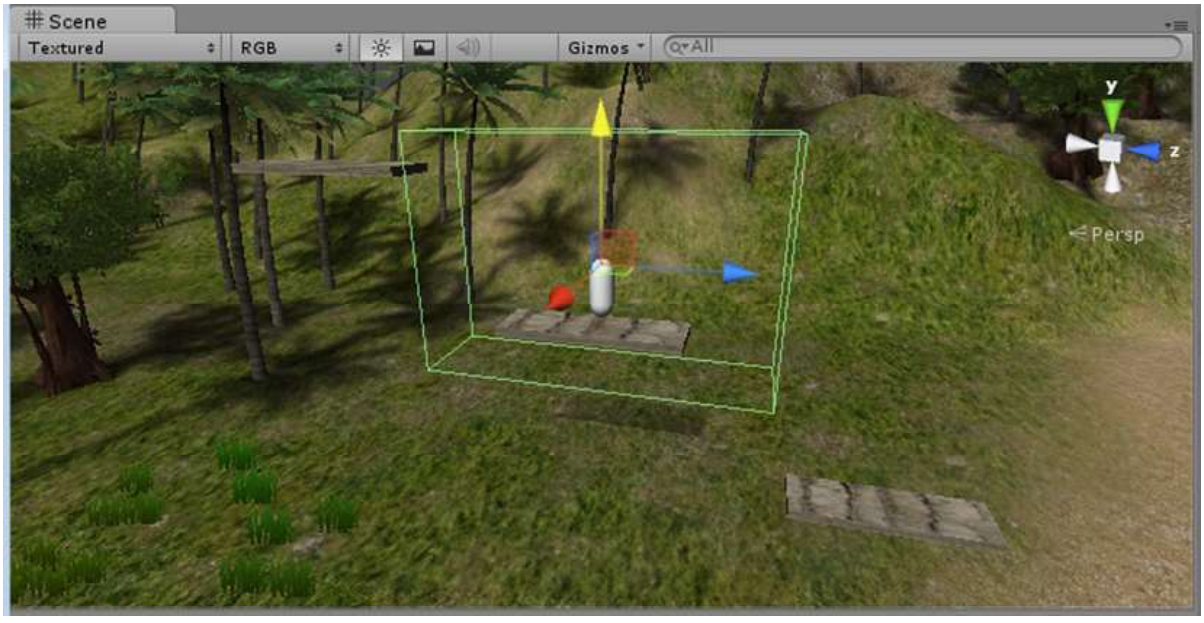

Figure 5-40. *The First Person Controller is no longer affected by the collider*

Now, you need to create a script that sends a message to the plane to tell it to toggle its Is Trigger parameter.

1. Stop Play mode.
2. Select the My Scripts folder.
3. Create a new JavaScript.
4. Name it **SendMessageOnTrigger** .
5. Open it in the editor.
6. Delete the default functions and add the following:

```
var sendToObject  : GameObject;

function OnTriggerEnter (object : Collider) {

        // call the function on the specified object
    sendToObject.SendMessage("ToggleTrigger");
}
```

Let's see what is happening with this function. The variable `sendToObject` is defined as a GameObject type, but the value is not yet provided. `SendToObject` will be the object that contains the script with the `ToggleTrigger` function you need to call. As with the platform targets, you will have to drag and drop the actual object into it in the Inspector. The reason you can't specify the object when you declare the variable is that variables are read into the system before any of the scene objects are loaded and, therefore, do not yet exist.

The `OnTriggerEnter` function is an event-type function that is called when another object's collider first enters its collider. With the argument `object : Collider`, the colliding object is returned in case you need to do any condition-checking with it. The argument, the local variable `object,` is declared as a Collider type. It gets its value, the colliding object's collider, when the event happens. Because it is a local variable, it is destroyed as soon as the function is finished, thereby freeing memory. You don't have to use the word *var* because it is understood that the argument is a variable.

The guts of the function, `SendToObject.SendMessage("ToggleTrigger")`, looks through the `sendToObject`'s scripts for one with a function named `ToggleTrigger`. When it finds one, it calls that script.

1. Stop Play mode.
2. Save the script.
3. Drag it onto TheTrigger.
4. Drag the Plane object onto its Send To Object parameter in the Inspector.
5. Click Play and watch the status line to see when the plane is "opened" or "closed."

Nothing appears to be happening.

6. Drive the First Person Controller onto the platform and let it go for a ride.

Now, the status line reports when the plane's Is Trigger parameter is being toggled.

The Scripting Reference for `OnTriggerEnter` tells us that "trigger events are only sent if one of the colliders also has a rigidbody attached." You can't see it, but the First Person Controller has its own simplified version of a rigidbody. To make the platform trigger the `OnTriggerEnter` event, you will have to add a rigidbody component to it.

7. Stop Play mode.
8. Select the Platform Group object.

9. From the Component menu ➤ Physics, select Rigidbody.
10. Click Play.

The intersection with TheTrigger is now registering, but the platform is sinking lower and interacting with the ground.

1. Stop Play mode.
2. Uncheck the Rigidbody's Use Gravity parameter in the Inspector.
3. Click Play.

The Platform triggers the intersection and no longer sinks.
Let's not forget to test it with the First Person Controller.

4. Drive the First Person Controller to the platform and jump aboard.

If the First Person Controller doesn't jump cleanly onto the platform, the platform skews wildly and the First Person Controller eventually falls off. There seems to be a small problem . . . .

5. Stop Play mode.
6. In the Rigidbody component, check Is Kinematic.
7. Click Play and take the First Person Controller for a ride on the platform again.

Everything works as expected now. While the Rigidbody is marked isKinematic, it will not be affected by collisions, forces, or any other part of physX. Is Kinematic tells the engine not to use any of the regular physics parameters but allows collision testing to be more efficient.

■ **Rule** A Static Collider is a GameObject that has a Collider but not a Rigidbody. You should never move a Static Collider on a frame-by-frame basis. Moving Static Colliders will cause an internal recomputation in PhysX that is quite expensive and which will result in a big drop in performance. On top of that, the behavior of waking up other Rigidbodies based on a Static Collider is undefined. Colliders that move should always be Kinematic Rigidbodies, if you do not want any physics calculations performed.

---

8. Select the PlatformTarget prefab in the Project view and deactivate its Mesh Renderer.

Now that you've had a first look at triggering events, let's get back to the `SendMessageOnTrigger` script. The script toggles the plane's Is Trigger parameter only when the collider is entered. If you were using the script to affect collision walls around the platform, you would want the exit event to toggle the plane's Is Trigger as well. As you might guess, there is an `OnTriggerExit` function.

1. Copy the `OnTriggerEnter` function and paste it below the original.
2. Change `Enter` in the name to **`Exit`**.
3. Save the script.
4. Watch the status line.

The message from the Plane's `ToggleTrigger` function shows that it is now being called when the platform enters the collider and exits it.
Let's not forget the First Person Controller.
Jump the First Person Controller onto the platform and watch the status line.

Now both objects, the First Person Controller and the platform, trigger the function call, so the messages change quickly or don't appear to change at all. If you open the Console, you'll see that the function is working. (The Console can be opened through the Window menu or by the keyboard shortcut found there.) It is now overworking, in fact. This is where the argument that reports the colliding object comes in handy.

5. Inside both `OnTrigger` functions, just above the `sendToObject.SendMessage line`, add the following line:

```
print (object.name); // this is the name of the object triggered the event
```

6. Save the script and watch the results in the Console (see Figure 5-41).

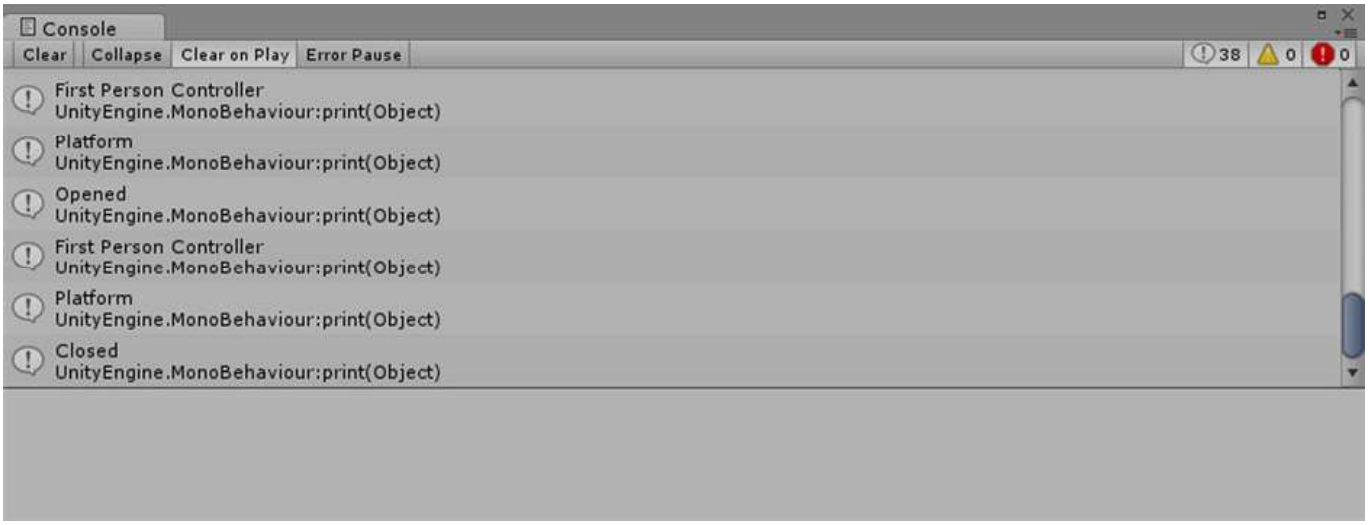

Figure 5-41. *The triggering objects being reported in the Console*

Object Names

The most important concept to understand here is that an object's name does not represent the object. The name is only another parameter of the object. This is another reason why you can't type an object name as the value of a variable. The name is of type String. It is just a string of characters and does not represent the actual object.

For the script, you need to determine if the colliding object is the one you are interested in. In a shooter-type game, you're less interested in the colliding object specifically and more interested in generalities. If the object is a projectile, it needs to send a message to whatever it hit to take damage points or even be destroyed. For this adventure game, you need to be more specific. Let's add a variable that holds the only object we want to trigger the send message. You may be thinking that the Platform Group is the collider, because that was what we added the rigidbody to, but a check of the Console printout shows us that the Platform itself is what has the collider and was passed in as an argument (another good reason to print out the colliding objects).

1. In the `SendMessageOnTrigger` script, add the following variable beneath the `var sendToObject` line at the top of the script:

```
var correctObject: GameObject; //this is the only object we allow to trigger the event
```

2. Save the script.

3. Drag the Platform object onto the Correct Object parameter in the Inspector (see Figure 5-42).

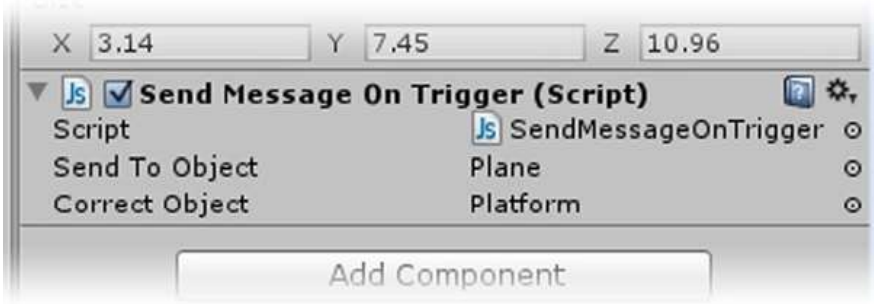

Figure 5-42. *The loaded parameters*

4. Change the `sendToObject.SendMessage` line in both `OnTrigger` functions to the following to add the conditional:

```
if(object == correctObject.collider) sendToObject.SendMessage("ToggleTrigger");
```

5. Now, the only time the `ToggleTrigger` function will be called is when the argument (a collider object) that is passed in on intersection matches the collider component of the object you specified as the correct object. Comment out or delete the two print statements.
6. Save the scene and save the project.

Your First Build

Now that you can travel around the environment and interact with some of it, you may as well try your first build. Because the project already contains three scenes, you'll need to specify which one you want.

1. From the File menu, select Build Settings (see Figure 5-43).

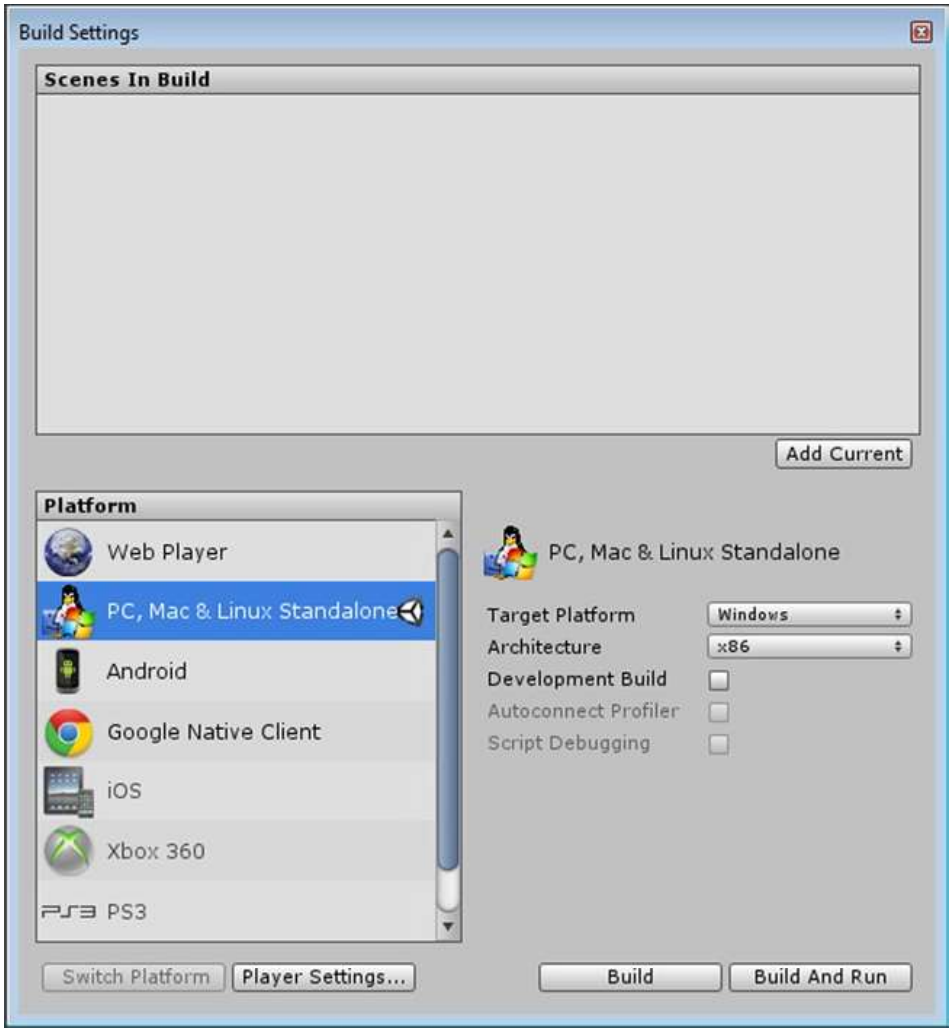

Figure 5-43. *The Build Settings window*

2. Click the Add Current button at the lower right of the Scenes In Build window (see Figure 5-44).

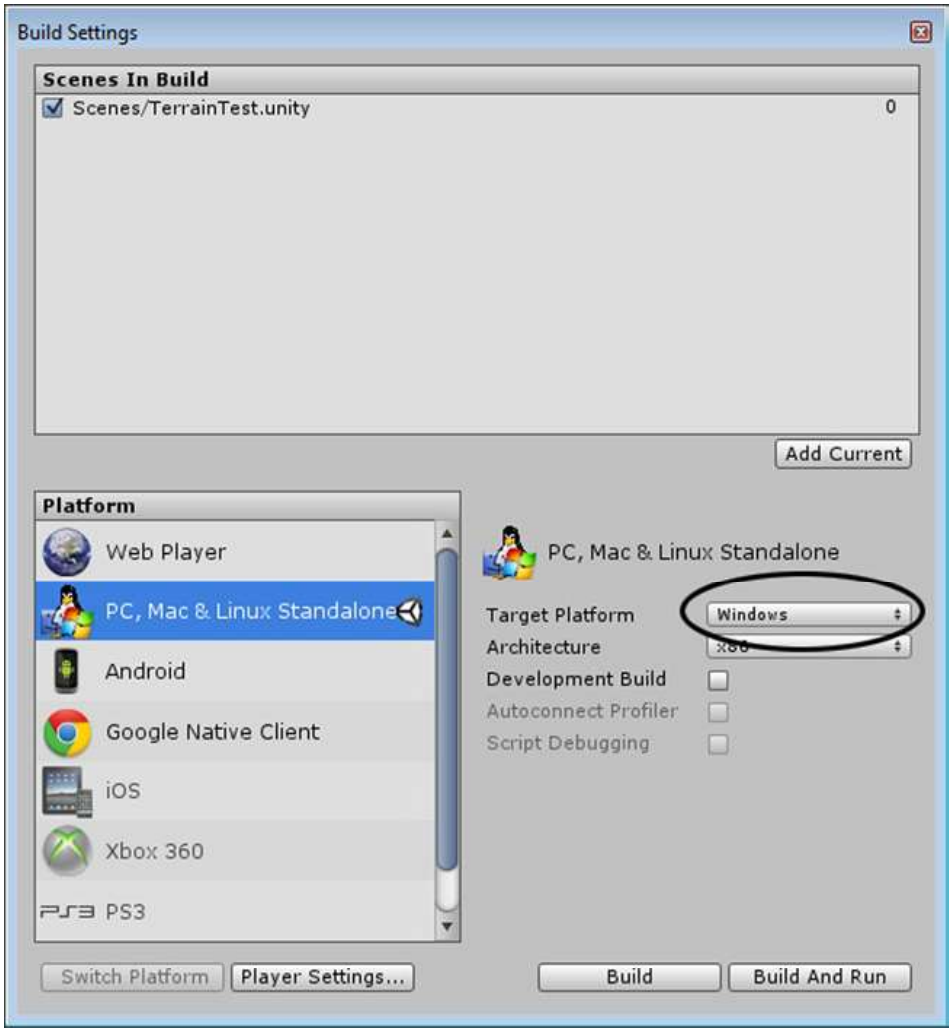

Figure 5-44. *The Build Settings window with the current scene loaded*

3. If there are any other scenes there that are checked, uncheck them.
4. Select your Target Platform as Windows or Mac.
5. Click Build and Run.

You will be prompted to give a file name and location.

6. Name your scene **Chapter5Navigation.exe**.

As indicated by the button name, the game will be built and then immediately run.

The first thing you'll see is the configuration window (see Figure 5-45), where the user can choose screen resolution and see or change the keyboard and mouse mapping. You will learn how to suppress this dialog to prevent the player from setting up conflicting keyboard controls later on.

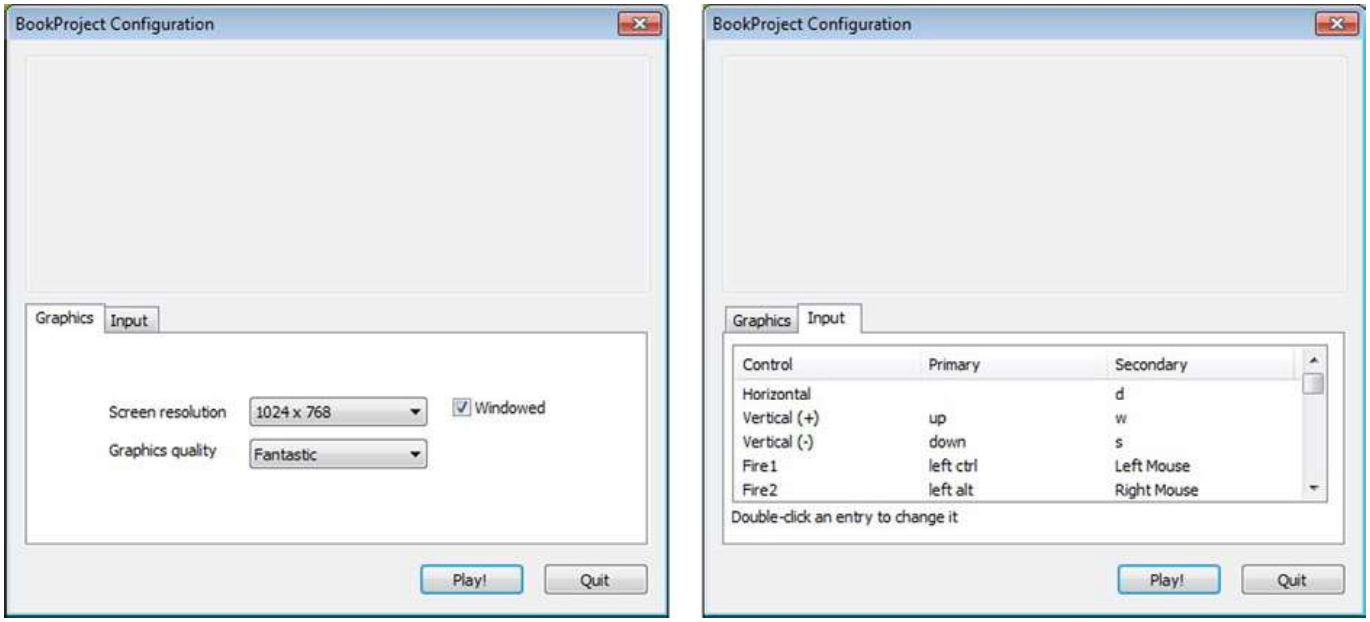

Figure 5-45. *The configuration and Input windows in Windows (It will be slightly different on a Mac.)*

7. Click the Input tab.

A pruned-down version of the Input Manager appears with all of the key assignments (see Figure 5-45).

When creating a Unity .exe for Windows deployment, you will have the .exe file and a folder of the same name that contains the data (see Figure 5-46). If you wish to share your file at that stage, you will need to distribute both the folder and the .exe. For Mac deployment, you'll get one package with everything in it.

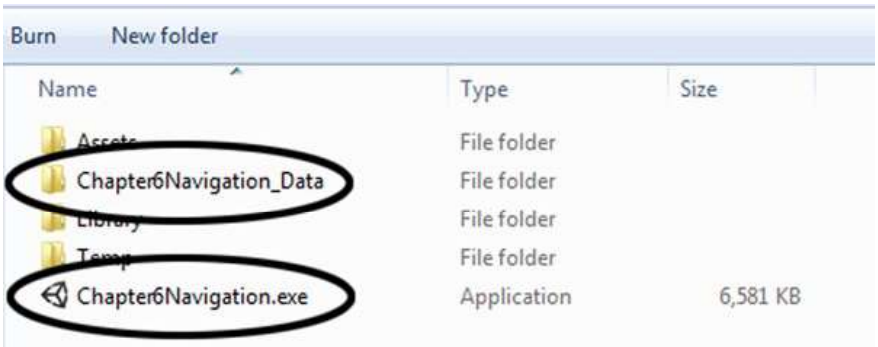

Figure 5-46. *The files associated with the runtime application in Windows*

8. Click Play and test the scene (see Figure 5-47).

Figure 5-47. *The scene running as an .exe*

Don't forget the invisible wall: the platform will toggle it off and on, or you can use the t key.

Defining Boundaries

You may have discovered that you can fall off the edge of the world. Even with steep mountains around the edges, you can still use the jump technique to climb to the top and then jump off. If you think about what you've learned about collisions walls, you know you need something perfectly perpendicular to foil those pesky jumpers and climbers. You've seen a planar object used to provide a collision wall, but you've no doubt noticed that there's no Plane Collider, so you can assume it's not very economical. The best choice is a Box Collider for each side.

1. Use the Scene Gizmo to get to a Top view of the scene.
2. Zoom out in the Scene view until you can see the edges of the terrain, as shown in Figure 5-48.

Figure 5-48. *The edges of the known world*

3. Create an Empty GameObject.
4. Name it **Terrain Boundaries**.
5. Create another Empty GameObject.
6. Name it **Boundary1**.
7. Add a Box Collider to Boundary1.
8. Adjust the Collider's Size to create a thin wall at the edge of the terrain.
9. Drag it into the Terrain Boundaries object.
10. Clone Boundary1 to create the remaining three sides (see Figure 5-49).

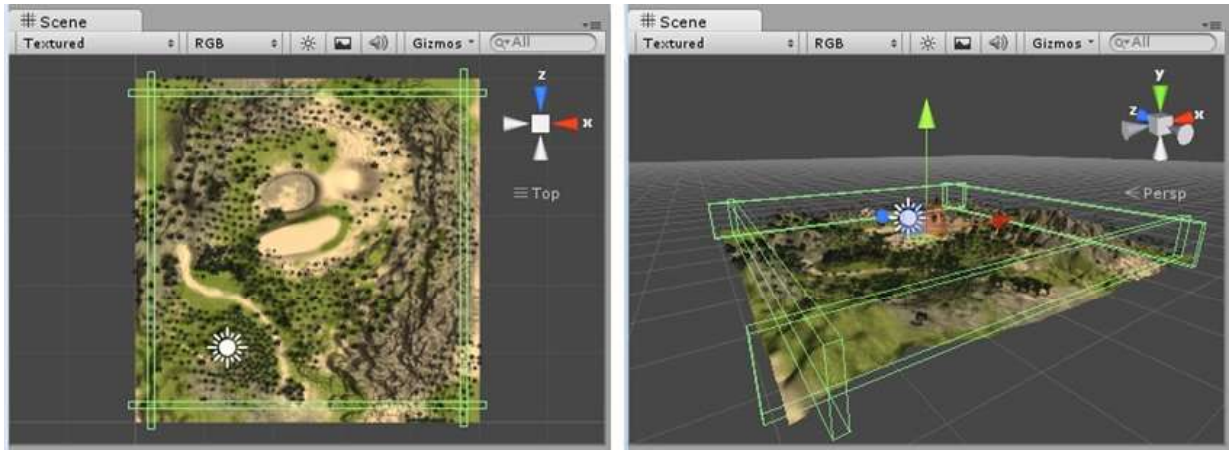

Figure 5-49. *The scene boundaries*

11. Save the scene and save the project.

Summary

In this chapter, you added player navigation to your test terrain by starting with the First Person Controller prefab and making a few changes. With the help of the Input Manager, you modified the shooter-type controls, WASD, to allow less-experienced gamers to use the arrow keys to turn instead of strafe. Next, you added the code for arrow turning in the FPS Input Controller script to create your own version of the script, the FPAdventurerController. Because the arrow turns needed to be frame rate–independent, you added that code to the `FixedUpdate` function rather than the regular `Update` function, as `FixedUpdate` is checked at set intervals rather than every frame.

In your version of the `MouseLook` script, `MouseLookRestricted`, you restricted mouse look to require the shift keys, the right mouse button, or any of the regular navigation keys. The restriction on the mouse look functionality will enable your player to interact with or pick objects in the scene without having the view spin wildly as he or she tries to put the cursor over a particular object.

By using the concept of the virtual buttons created in the Input Manager, you were able to use `Input.GetButton` after creating your own virtual buttons, instead of hardcoding each individual key or mouse button with `Input.GetKey` and `Input.GetMouseButton`. You used the logical and, &&, and the logical or, ||, to expand your conditional statements to check for multiple conditionals.

From another component of the First Person Controller, the `CharacterMotor` script, you found where you could fine-tune navigation speeds and functionality. As one of the options was to enable platform functionality, you experimented with a simple moving platform to explore its capabilities and limitations. You discovered that your platform didn't exhibit behavior sufficiently consistent to use without a lot of extra thought and design.

With the platform targets, you learned to create prefabs for objects that would have multiple instances in a scene, to allow for more economical handling. You found you could remove or disable components such as the Mesh Renderer or Colliders to streamline the functionality of your objects.

This introduced the concept of collision walls to block the player from going through objects. You discovered that a fully perpendicular wall will stop the First Person Controller from getting through an object with a collider, but that if the wall was not perpendicular, the player could possibly use the jump action to climb the wall. Rather than delving into mathematical calculations to prevent this, you found that a simple plane object with its Mesh Collider would effectively block the action.

During your experiments with the plane object, you found that a collider's Is Trigger parameter would let you pass through an object and even trigger an event, as long as one of the objects, or its children, also contained a rigidbody component. You learned that an object that moves without the help of physics should contain a rigidbody marked as Is Kinematic. With the help of the `SendMessage` function, you learned how to call functions when intersection was caught by the `OnTriggerEnter` and `OnTriggerExit` event functions. You also found that a GameObject's name was merely a parameter of the object, of type String, and did not represent the GameObject itself. Finally, you built your first .exe of the scene, only to discover you still needed to create collision walls around the perimeter of the terrain.

In the next chapter, you will be developing the cursor functionality so that you can have full control over its behavior. Cursors, after all, are key to the point-and-click adventure genre.