CHAPTER 6

Cursor Control

In point-and-click games, cursors are a large part of the user experience. Besides showing the user where the cursor pick point is, they often give the user clues as to when they are over an action object and even the type of interaction expected. If it's done well, cursor functionality in games is taken for granted by the player. Surprisingly, there are a lot of design decisions you will have to make to achieve that level of invisibility. In this chapter, you will be confronting many of the issues and testing various solutions, as you find out how they are handled in the Unity game engine.

Currently, Unity has a few different systems for creating GUI (graphical user interface) elements. The older system uses gameObjects that live in 2D space on the screen. The other system is fully script-based and doesn't exist until you are in Play mode. A third system is under development and was not yet available for beta when this book was being updated. To make the decision more difficult, in 4.0, Unity provided a means of replacing hardware cursors with your own images. While this solves a lot of issues, the size is limited to 32×32 pixels on some operating systems. You will be experimenting with a few different options.

As with most problem solving, it is usually best to start simple and then add more or better functionality a little at a time. You will start by manipulating the default operating system cursor.

Cursor Visibility

One of the more confusing problems when first learning to script in Unity is knowing where to put the bit of code shown in the Scripting Reference or other sample source for testing. Because the cursor is not related to a particular object in the scene, the code to control it could go in any script's `Update` function, because it must be checked every frame. For this test, you will create an empty GameObject to hold your small test script.

1. Open the TerrainTest scene in Unity that you created in the previous chapter.
2. Save the scene as **CursorControl** in the Scenes folder.
3. Deactivate the Plane, the Wall, and TheTrigger at the top of the Inspector.
4. Save the scene.

Before going any further, let's also turn the print statement on a script that stays active.

5. In the FPAdventureController script, delete or comment out the `if(Input.GetAxis("Turn")) print...` line.
6. Save the script.

Now you're ready for some more tests.

7. Create an empty GameObject.
8. Name it **ScriptHolder**.

9. In the My Scripts folder, create a new JavaScript.
10. Name it **HideCursor**.
11. Open it in the script editor.

It's time for a design decision. You know you don't want the cursor to appear when the player is navigating the scene, so you have to check for the virtual buttons involved in navigation: Horizontal, Vertical, and Turn. If any of them is being pressed, the cursor should be hidden. The decision is whether to have it show during mouse look, while the player is stationary. Because looking is a type of navigation, let's include the virtual ML Enable button in the conditional (but you may leave it out if you wish).

1. To hide the operating system cursor, add the following lines inside the `Update` function:

```
if (Input. GetButton ("Horizontal") || Input. GetButton ("Vertical") ||
  Input. GetButton ("Turn") || Input.GetButton("ML Enable") ){
  // a navigation key is being pressed, so hide the cursor
  Screen.showCursor = false; // hide the operating system cursor
  }
```

2. Just above the last curly bracket of the `Update` function, add:

```
else {
// no navigation keys are being pressed, so show the cursor again
Screen.showCursor = true; // hide the operating system cursor
}
```

3. Save the script.

■ **Note** In your conditional, you have used `Input.GetButton`, because it returns a true-or-false value when the virtual button or key is down. You could also use `Input.GetAxis`, even though it returns numbers, which may be less intuitive at this stage but slightly more efficient. If you eventually plan on creating apps for iPhone or iPad, `Input.GetAxis` is the one to use, because mobiles have no mouse button functionality!

---

4. Drag it onto the ScriptHolder object in the Hierarchy view.
5. Click Play and test navigation.

The cursor may or may not disappear as expected. If not, it should in maximize mode.

6. Turn off Play mode.
7. At the top of the Game window, turn on Maximize on Play (Figure 6-1).

Figure 6-1. *Maximize on Play*

8. Now click Play.

The Game view window is maximized (Figure 6-2), and now the cursor behaves as expected: it disappears when you are moving.

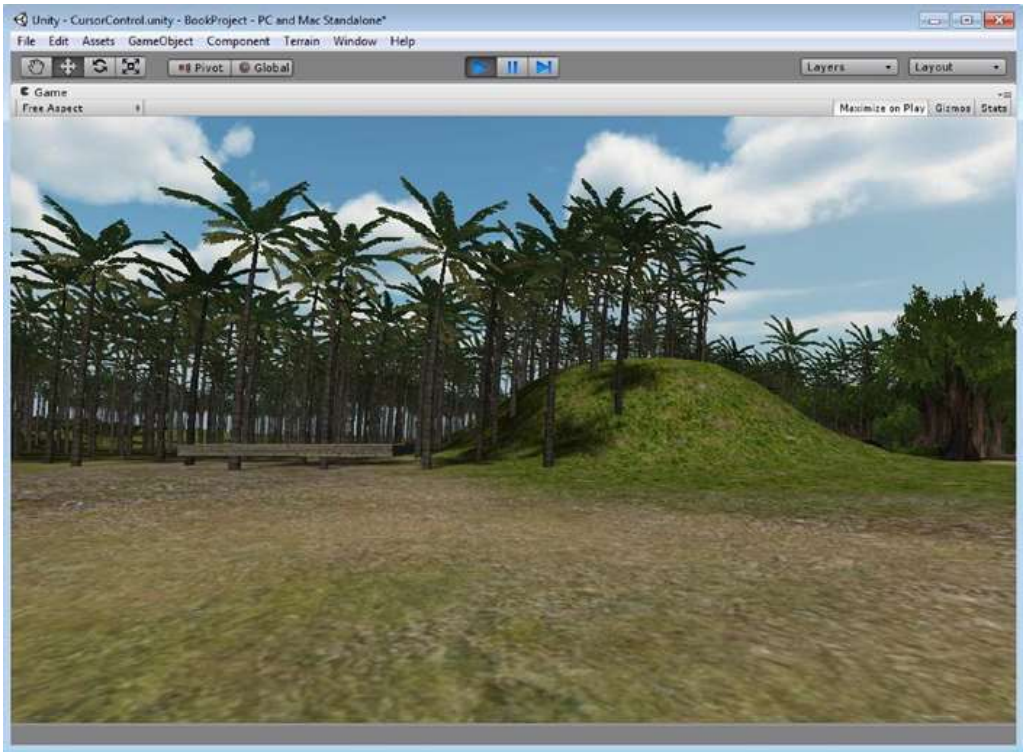


Figure 6-2. *The maximized game window*

9. Turn off Play mode and turn off Maximize on Play.

Let's see what happens when the First Person Controller is riding on the platform.

10. Click Play again.
11. Drive the First Person Controller onto the platform.

The cursor is visible while the First Person Controller is passively riding the platform. On the off chance you want to have an action object that the player must interact with while en route, such as picking an orchid off a rocky cliff face or stealing an egg out of a bird's nest, you can consider it acceptable.

12. Stop playback.

Custom Cursors

As mentioned, a big part of point-and-click adventure games is the cursor. In the first-person variety, when navigation stops, the user is free to scan the objects in the view, or frustum, to see if any objects are action objects.

■ **Tip** The viewing *frustum* is the area of the 3D world that can be seen within the game's viewport at runtime or in Play mode. If the player turns, and an object is no longer visible in the scene, it is considered "out of the frustum." The state of an object in respect to the frustum is important in that any manner of things can be done to improve frame rate, such as stopping calculations on looping animations.

For your purposes, you will consider an action object something that can be put into inventory or will animate when picked; it's something that is a critical part of the game/story.

Typically, to identify an action object, the cursor might change color or appearance. In more stylized games, the action object itself might change color or tint. Once the cursor picks an object, it will usually either be turned off while the object animates or, if the object is an inventory object, it (or a 2D icon representing it) could become the cursor. One of the fun features of adventure games is finding out which items can be combined to make a new, more useful object.

In Unity, 2D sprites and GUI (Graphical User Interface) objects are handled quite differently than 3D mesh objects. Unity's camera-layering system allows you to overlay mesh objects onto your 3D world space, but the mesh objects rarely appear scaled to fit the objects with which you wish to interact, depending on the player's location in the scene. Using 2D textures as cursors will involve a little work but provides a great amount of flexibility, especially if you wish to use cursors larger than 32×32 pixels.

Cameras have a built-in GUI layer that automatically draws 2D objects into 2D screen space, layered on top of the 3D world. UnityGUI is a purely scripted means of drawing 2D objects onto the screen; it's handy for menus and controls but is less intuitive for artists and designers with no scripting experience. To start, you'll experiment with a tangible object that can be seen and manipulated from the Hierarchy view.

GUI Texture Cursor

1. In the Project view, create a new folder.
2. Name it **Adventure Textures**.
3. From the Assets menu, select Import New Asset.

4. From the book's Chapter 6 Assets folder, select GamePointer.tif (Figure 6-3).

Figure 6-3. *The GamePointer image in the Inspector*

5. Select the image in the Project view.

Texture Importer

The Texture Importer comes with several handy presets, as well as the Advanced option for full control. Let's take a quick look at it, just to see the possibilities.

1. In the Texture Type drop-down at the top of the Texture Importer, select Advanced (Figure 6-4).

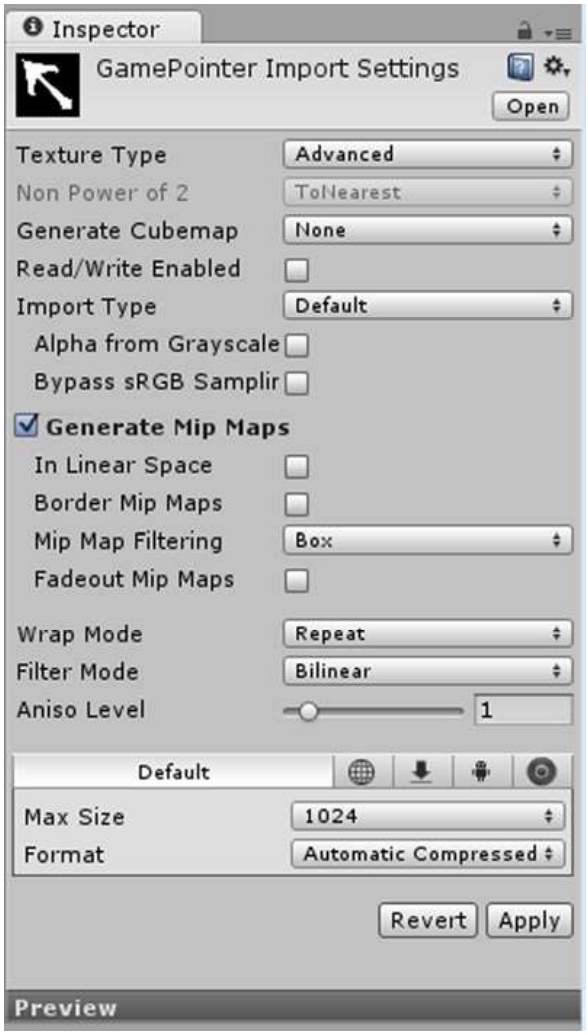
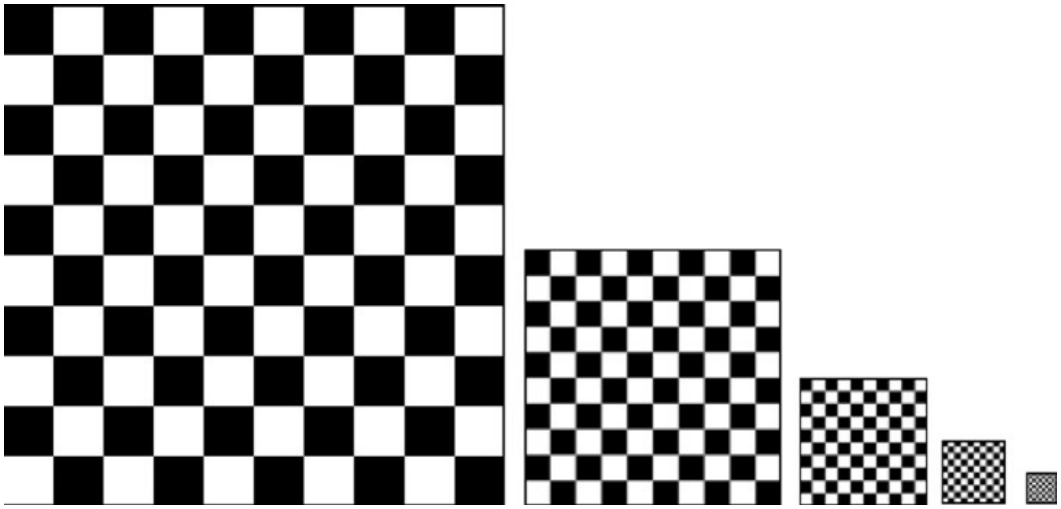


Figure 6-4. *The advanced parameters*

Take a look at the parameters in the Inspector (see preceding Figure 6-4). Because the texture will only be used as a cursor in 2D screen space, it doesn't have to be Mip mapped. This will save memory.

MIP MAPPING

Mip mapping is when an image is copied and reduced in size and clarity to provide blurrier images at greater distances. This reduces the nasty flickering as the view is changed. As an example, picture a black-and-white checker texture. As the texture gets farther back into 3D space, there are fewer and fewer pixels to represent the black checkers and the white checkers. At some point, it is a tossup as to whether the pixel will be black or white. In one frame, it may be black, but in the next, it may be white. This causes a flickering known as *artifacting*. By scaling the image down by powers of two, the successive images are blurrier and blurrier, eliminating or greatly reducing artifacting.



Mip Maps for a checker texture at 256, 128, 64, and 32

You may also notice in the Preview window that the image is shown as an RGBA Compressed DXT5 image. Unity converts image formats to the .dds format, so feel free to use larger file size images, such as .tif, .psd, and others.

■ **Tip** Unity automatically updates images when changes are made to them, so you may indulge yourself and leave them in a layered format in the Assets folder.

Fortunately, your needs for the GamePointer image are simple, so you can use a preset.

2. From the Texture Type drop-down, select GUI.
3. Just above the Preview window, click Apply.

The image is ready to use (Figure 6-5).

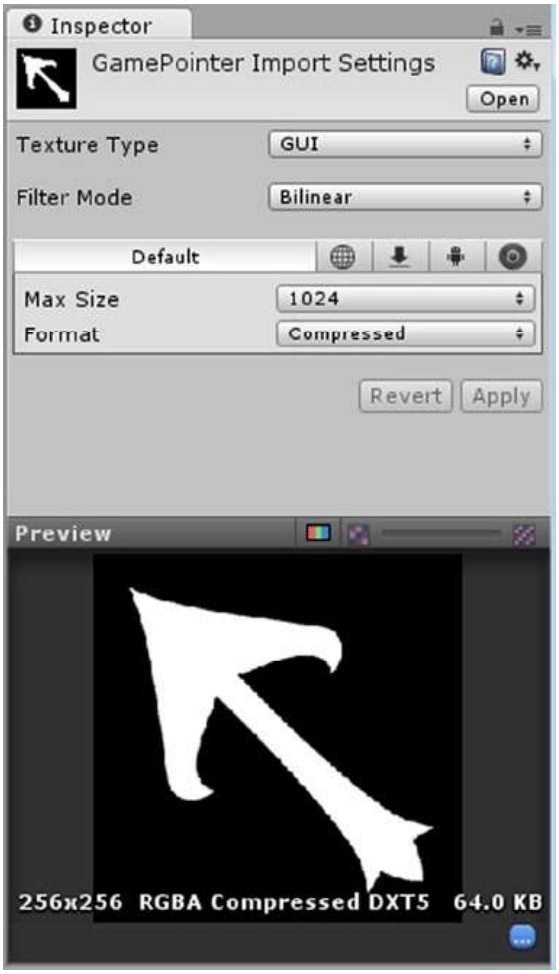


Figure 6-5. *The texture as GUI import type*

4. With the image still selected in the Project view, from the GameObject menu ➤ Create Other, select GUI Texture (Figure 6-6).

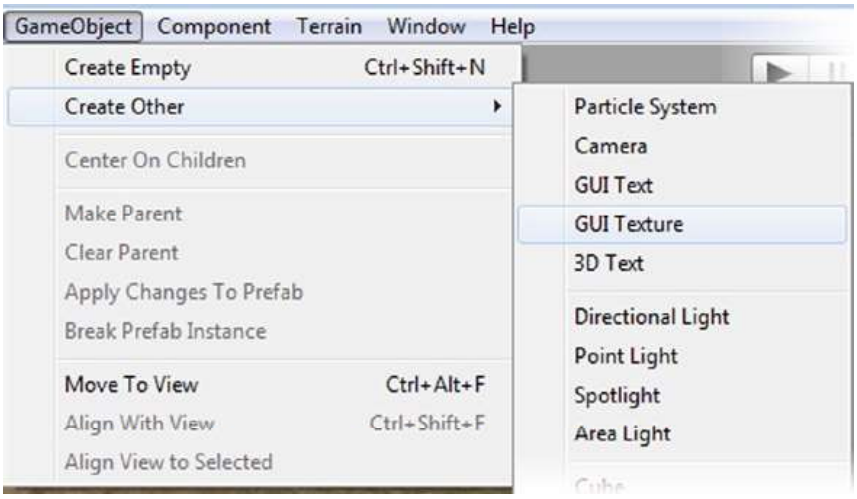


Figure 6-6. *New GUI Texture*

The image appears in the middle of the Game view (Figure 6-7).

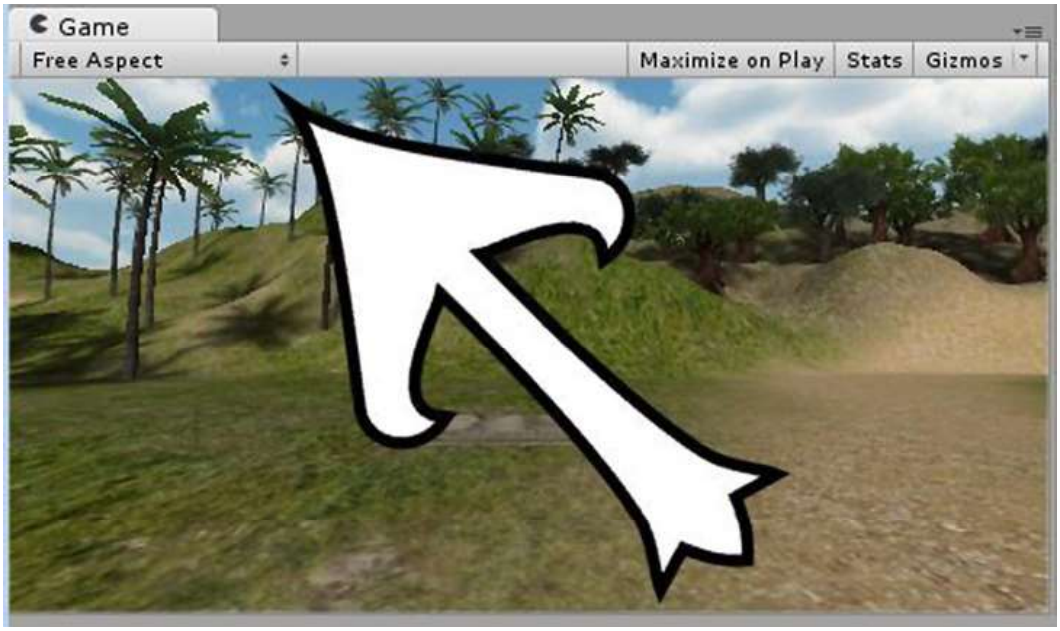


Figure 6-7. *The new GUI Texture in the Game view*

When created by selecting the texture first, the new GUI Texture object is given the name of the texture.

5. If you choose to use your own texture, rename the new GUI Texture object GamePointer.

■ **Tip** When an image is selected before the GUI Texture gameObject is created, the sprite, or screen blit, will be made the exact size of the image and will be given the image's name as well.

A bit of history: The term *blit* originated as an acronym (Berryman Logical Image Technique), whereby copying a rectangle on the screen and *blitting* it into another position on a computer screen was an early means of 2D animation.

---

6. Click in the Game view to move the focus to that window.
7. Press the spacebar to toggle the view to maximum.

Note that the GUI Texture remains the same size.

8. Toggle the view back and adjust the Game view window size by hovering the cursor over the edge until you get a resize icon, then drag the window smaller (Figure 6-8).

Figure 6-8. *The squashed Game window*

The pointer image remains the same size.

Before you can decide how big the GUI Texture object should be, you need to decide on the size of the window when the game is published. As you may remember from the Configuration window that popped up before your test game was played, the user can be allowed to change the screen size. As you may have guessed, because 2D UI objects do not scale to keep their size relative to the screen size, allowing the user to change it would mean a lot of extra scripting on your part. While you may choose to go that route eventually, for now you'll look into constraining the resolution and preventing the player from making changes.

■ **Note** Later on, you'll see that you can force screen mode and screen size in the Player Settings, but this gives you a good way to learn about the Screen functions that are useful throughout the project, so let's experiment with them.

If you do a search for resolution in the scripting help, you will see several useful resolution related topics. Let's start with `Screen.SetResolution`.

The code and description looks pretty straightforward. You feed it an x and a y dimension and use a Boolean true/false to tell it to use full screen or not. The question is the usual, if you are new to scripting and/or Unity: where to use the code and what function to put it under.

Since this is something that should be handled at the start, when the game is loaded, let's create an object just for holding scene settings.

1. From the GameObject menu, create a new Empty GameObject.
2. Name it **Control Center**.
3. In the Adventure Scripts folder, create a new JavaScript and name it **GameManager**.
4. Open it in the editor.

The GameManager object will be referenced by name by various other objects throughout this project and any other games where you use these scripts. You may not rename this object.

So far, you have put code in the `Update` function, so it will be evaluated every frame, or the `FixedUpdate` function, when it needs to be evaluated at a constant rate. For the starting settings, however, you need to look at two new functions: the `Awake` function and the `Start` function.

The `Awake` function is used to initialize any variables or scene states before the game starts but after all objects have been created. It is always called before the `Start` function. The `Start` function is where you can pass variables to scripts that will need them. As with `Update`, any script can have its own `Awake` and `Start` functions, so theoretically, you could set the size of the game window anywhere. Logically, you will want to be able to find the scene settings script or scripts quickly, hence the creation of the `GameManager` script.

A published exe file will allow the user to change the screen resolution on startup. If you have a HUD or other GUI elements that could be impacted by view resolution, you may have to add code to internally compensate for screen resolution. For now, however, you'll just force a resolution override. You may wish to comment it out after you have tested it. Unity defaults to using pixel size and location with its GUI elements but allows you to position parent groups from the sides or center, so this may not impact the visual layout of your GUI as the screen is resized.

1. In the GameManager script, change the `Start` function to an `Awake` function.
2. Add the following line inside its curly brackets, so it looks as follows:

```
function Awake () {

   Screen.SetResolution (1280, 800, false); // false means not full screen, used windowed
}
```

3. Save the script.
4. Drag it onto the Control Center object.
5. At the top of the Game view, click the left-most button.

There is no option for the new resolution, just screen ratios. To get the desired size to show up for the Game view you will have to change the project settings.

6. From Edit, Project Settings, Player, in the Resolution and Presentation section, set the Default Width and Height to **1280×800**.
7. Now you can choose the Standalone (1280×800) option (Figure 6-9).

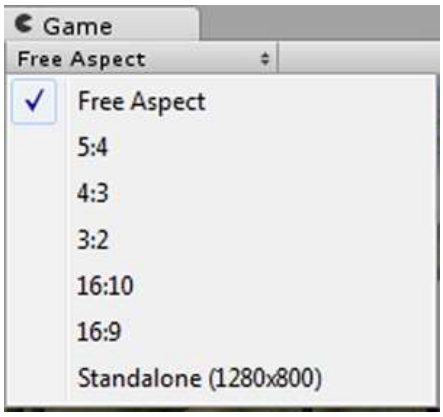


Figure 6-9. *Game view settings*

8. Toggle Maximize on Play (Figure 6-10).

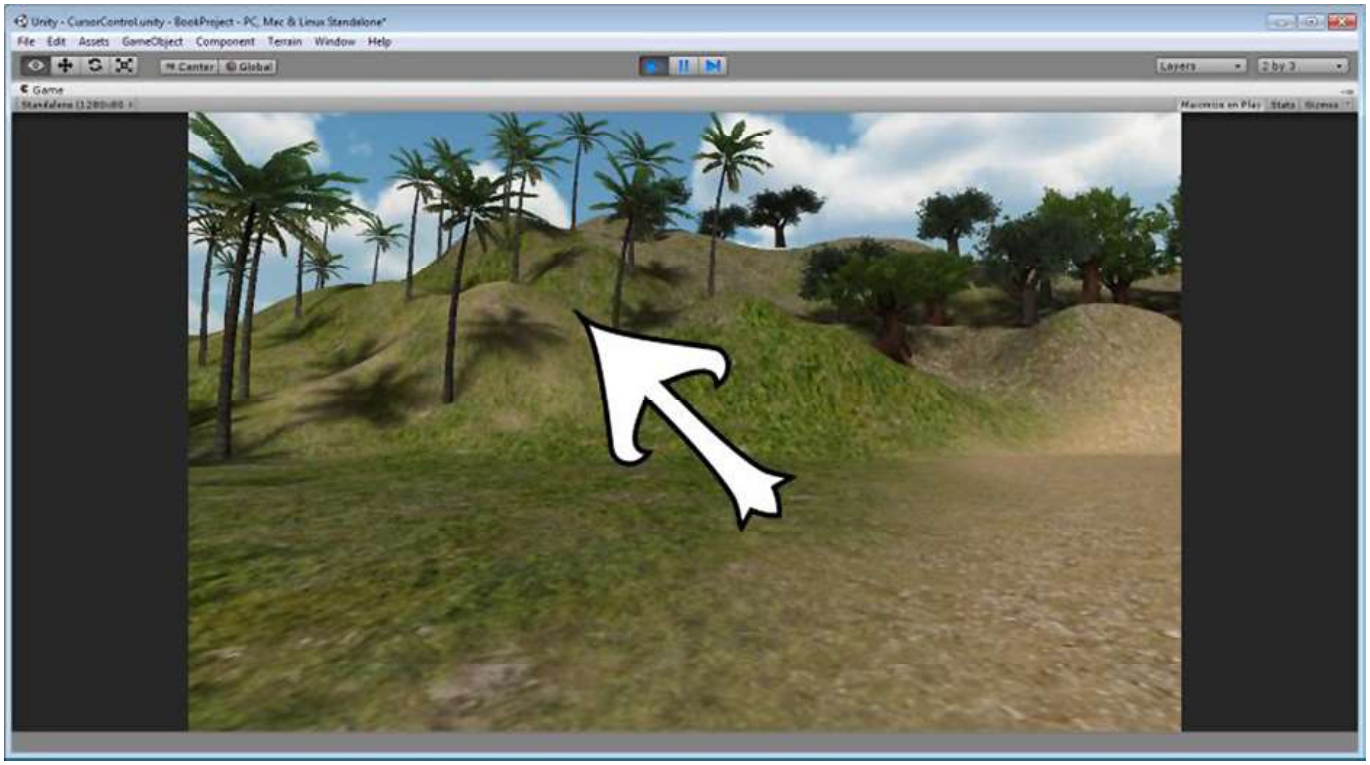


Figure 6-10. *The window at game resolution and/or aspect ratio*

9. Click Play.

Now you can see how big the cursor will appear. In your case, it happily occupies a large portion of your Game window. Obviously, you must make some adjustments. These can be made in either the Import settings, where you can specify a maximum size, or in the GUI Texture's parameters.

10. Stop playback.
11. Select the GamePointer texture in the Project view.
12. Set its Max Size to **32** and click Apply.

The newly resized image is stretched up to fit the original Width and Height.

13. Select the GamePointer object.
14. In the Inspector, set the Pixel Inset's Height and Width to **32**.
15. Set the X and Y Pixel Inset values to **-16** (half the texture size) to center it again (see Figure 6-11, left).

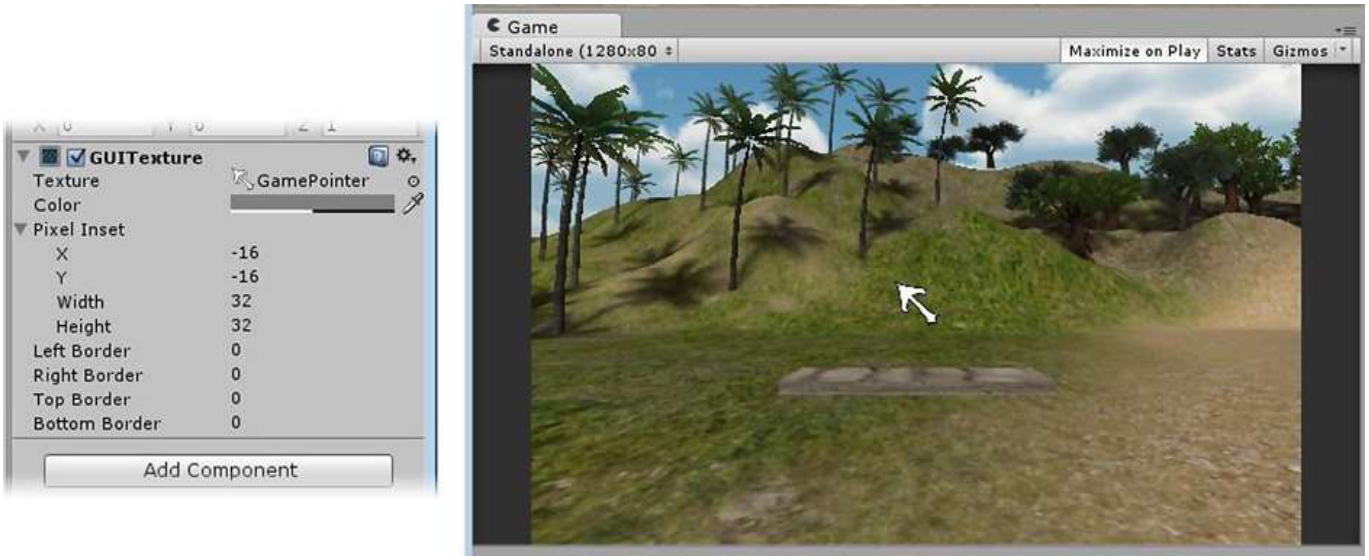


Figure 6-11. *The resized GUI Texture object*

As you can see in the Game view image of Figure 6-11, allowing some program (such as a game engine) to scale an image down may yield less than desirable results: note that the edges are jagged and nasty looking.

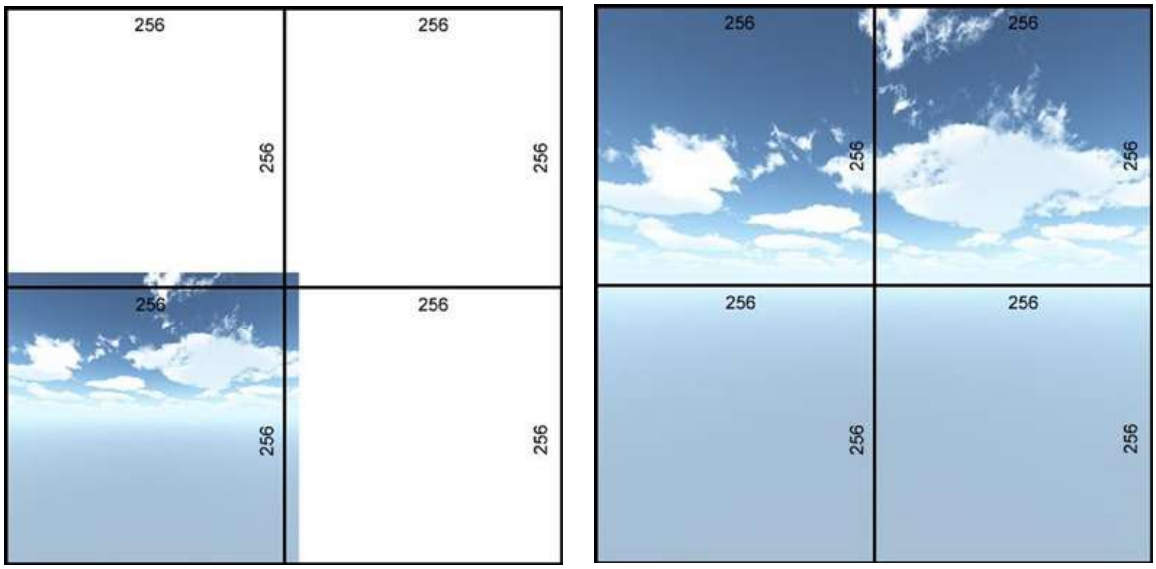
1. Select the Adventure Textures folder in the Project view.
2. From the Assets menu, select Import New Asset.
3. Import GamePointerSm from the Chapter 6 Assets folder.
4. In the Inspector, change its Texture Type to GUI.
5. Click Apply (you will be prompted to do so if you forget).
6. Select the GamePointer in the Hierarchy view.
7. Drag the GamePointerSm texture onto the Texture field or use the Browse icon next to the currently loaded GamePointer texture to select it from the Asset browser.

The new cursor image does not appear as jagged as its shrunken predecessor did (Figure 6-12).

Figure 6-12. *The smaller version of the GamePointer texture*

BASE 2

If you are at all familiar with game texture assets, you may recognize that most are in powers of two. This is because computer memory is arranged in base-two sized blocks. Images in pixel sizes that are powers of two fill the block entirely. An example would be an image that is 256×256 pixels. If that image were just one pixel larger in each direction, it would use up a 512×512 sized block—nearly four times as much memory! Therefore, whenever possible, the final image should be made in sizes such as 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, etc.



Memory usage

The image on the left takes up the same amount of memory as the image on the right. As soon as the image is even 1 pixel larger than the base-two size, it uses the next chunk of memory.

8. Set the Game window back to Free Aspect.
9. Turn off Maximize on Play.
10. Save your scene and project.

Color Cues

Now that you have shrunk your cursor into an acceptable size, let's try a few experiments on the cursor texture or image.

1. Select the GamePointer object in the Hierarchy view.
2. In the Inspector, find the Color parameter in the GUITexture section and click on the color swatch to bring up the Color dialog.
3. Change the color while watching the results in the Game view (Figure 6-13).

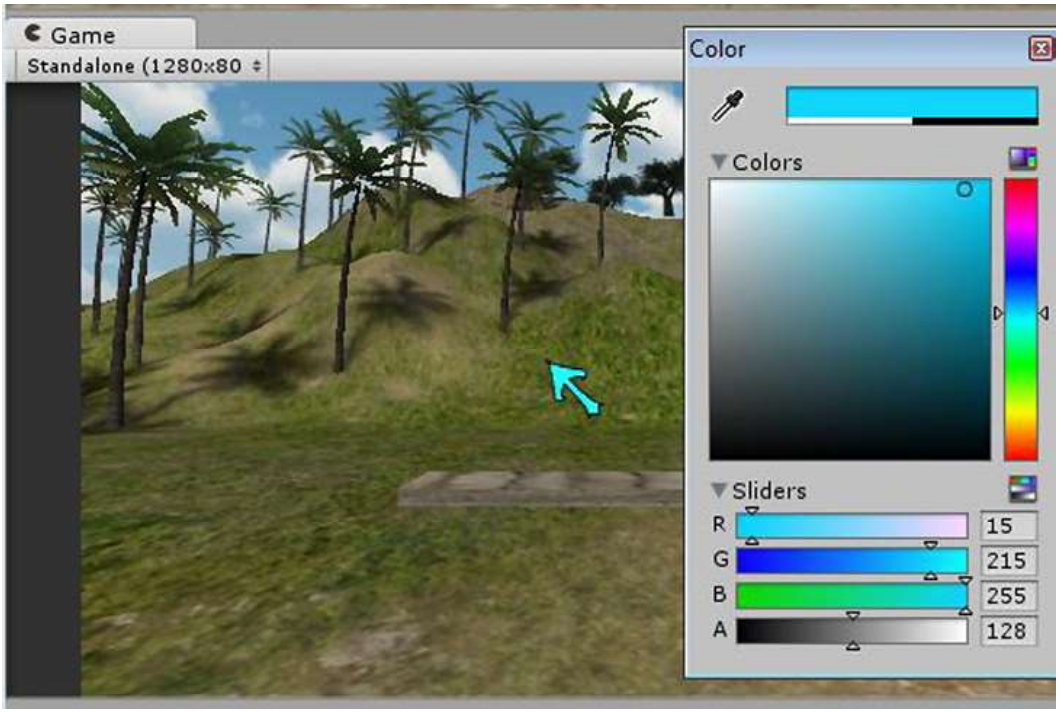


Figure 6-13. *Changing the cursor color*

Cursor Position

In Unity, in order to use a custom texture as a cursor, you have to turn off the OS (operating system) cursor, then track its position and feed it to your custom GUI Texture. Because there are several tasks that must be managed with your cursor, it makes sense to create a script for that very purpose. Vector2 type variable holds an x and a y value.

1. Create a new JavaScript in your My Scripts folder.
2. Name it **CustomCursor**.
3. Open it in the script editor.

Getting and assigning a cursor position is obviously something that needs to be done at least every frame, so it belongs inside the `Update` function. You will also add offsets to put the corner of the pointer in the correct position.

4. Add the following code inside the `Update` function so it looks as follows:

```
function Update () {
   // gets the current cursor position as a Vector2 type variable
   var pos = Input.mousePosition;

   // feed its x and y positions back into the GUI Texture object's parameters
   guiTexture.pixelInset.x = pos.x;
   guiTexture.pixelInset.y = pos.y - 32; // offset to top
}
```

5. Save the script.
6. Drag it onto the GamePointer object.
7. Click Play and move the cursor to the middle of the Game view.

Your brand-new cursor may not show up.

8. Move the cursor to the bottom left of the view.

The custom cursor now comes to the center of the viewport. It looks like you've got an offset issue.

9. Stop playback.
10. At the top of the Inspector, look at the values for the GamePointer's X and Y Position transforms.

They are both 0.5, which explains the large offset. For a GUI Texture object, the Position is in normalized screen space; 1.0 equates to 100% of the length or width.

1. Set them both to **0.0**, to line the custom cursor up with the OS cursor.

■ **Tip** GUI transforms are shown as a percentage of screen; full width and height is 1.0.

---

2. Click Play and test again.

If curiosity has gotten the best of you, and you have tweaked the X and Y Pixel Inset, you will have discovered that it reflects the screen position offset from the transforms.

3. Toggle off Maximize on Play.
4. Click Play again.
5. Move the cursor about the viewport, watching the Pixel Inset values change in the Inspector.
6. Stop Play mode.

If you are seeing a bit of ghosted operating system cursor, you will notice a bit of lag. On the positive side, changing the color of the texture will be easy.

Hardware Cursor

Let's take a minute to access the lag issue. In an adventure-type game, this probably isn't an issue. In other game genres, however, it could be a problem. In this section, you will give up the option to have a cursor larger than 32×32 pixels, in favor of allowing the operating system to take care of the cursor for you. A good feature of a hardware cursor is always drawn on top of everything else in the scene.

1. Select the GamePointer object and deactivate it in the Inspector.

Before you can assign a texture for the hardware cursor, you will have to change its Texture Type.

2. Select the GamePointerSm texture in the Project view.
3. Change the Texture Type to Cursor and click Apply.
4. From the Edit menu, select Game Settings, Player and load the GamePointerSm texture into the Default Cursor (Figure 6-14).

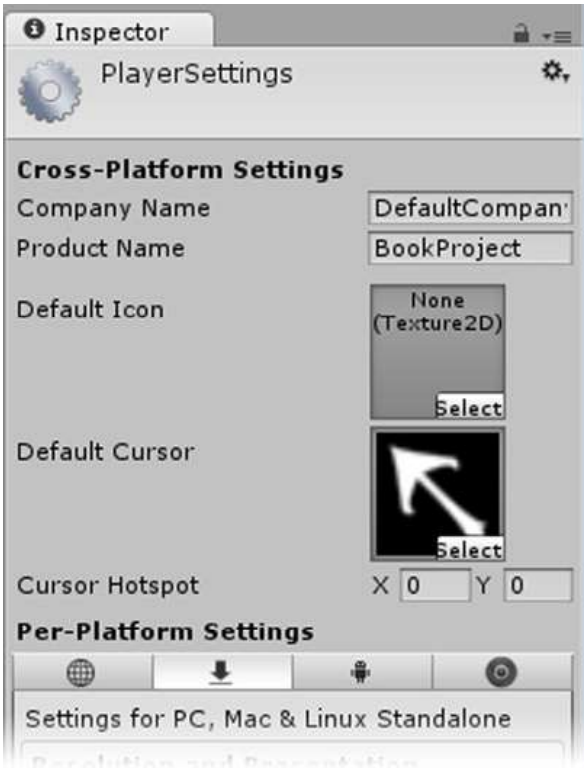


Figure 6-14. *Assigning a hardware cursor*

5. Move the cursor into the Game view, to see it using the new cursor texture.
6. Click Play and try navigating the scene.

As expected, this one is turned off by the Hide Cursor script when the player moves through the scene. The downside to the hardware cursor is that there is no way to quickly change its color. You could change the texture for a mouseover stand-in during runtime, but that would mean making duplicates for all of the textures that will eventually be used as cursors. It would also leave you without an easy way to let the player choose the mouseover color. It is also limited to 32×32 pixels on some operating systems, which will be small for representing action objects as cursors.

UnityGUI Cursor

A third option for the cursor is to use a scripted-only cursor. It has the advantage of always drawing on top of GUI Texture objects and can easily be set to draw on top of other UnityGUI elements (including text). The earlier GUI Texture object method will eventually be useful when you create the 2D inventory screen, but the cursor draw order issue means that you will need a better solution for your cursor. A downside of the UnityGUI cursor is that it exists only in code, so there is nothing tangible to work within the Hierarchy or other views.

Without delving deeply into UnityGUI for the time being, you can at least get a simple cursor up and running. Because the code for the cursor could theoretically be put anywhere, let's go ahead and put it on the GameManager script, so it can always be easily found.

UnityGUI elements are created in their own function, the `OnGUI` function. To start, you will need a couple of variables to store the default texture, and since the cursor will eventually change, the current cursor texture.

1. Open the GameManager script in the editor.
2. Add the following variables:

```
var defaultCursor : Texture;          // the default cursor texture, the arrow
internal var currentCursor : Texture; // the current cursor texture
```

3. In the `Start` function, assign the default cursor as the current cursor:

```
function Start () {

        currentCursor = defaultCursor; // assign the default as the current cursor
}
```

4. Create the `OnGUI` function:

```
function OnGUI () {

   GUI.DrawTexture (Rect(Screen.width/2, Screen.height/2,64,64), currentCursor);
// draw the cursor in the middle of the screen
}
```

5. Save the script.

■ **Tip** Function order in the script generally doesn't matter, but it's traditional to list the functions in the order that they are used. After the script's variables, an `Awake` function would be first, then the `Start` function, followed by the `Update`, the `FixedUpdate`, and then event-based and user-defined functions.

Before you can see anything, you will have to assign the texture and then click Play. But let's look at the `GUI.DrawTexture` line first. It starts by defining the size and position of a rectangle on the screen, then tells it which texture to put in the rectangle. Note that the size is set to 64×64 pixels. The arrow cursor is fine at 32×32, but when you have inventory objects used as cursors, you will want to use a bigger image. Rather than constantly scaling the `rect` up and down, it will make more sense to put the small arrow texture on a larger texture. The texture you will use is 128×128, but it will automatically be scaled down to f it the rectangle's space.

6. Select the Adventure Textures folder, right-click and select Import New Asset.
7. Bring in the GamePointerOffset texture from the Chapter 6 Assets folder.
8. In the Inspector, set its Texture Type to GUI and click Apply (Figure 6-15).

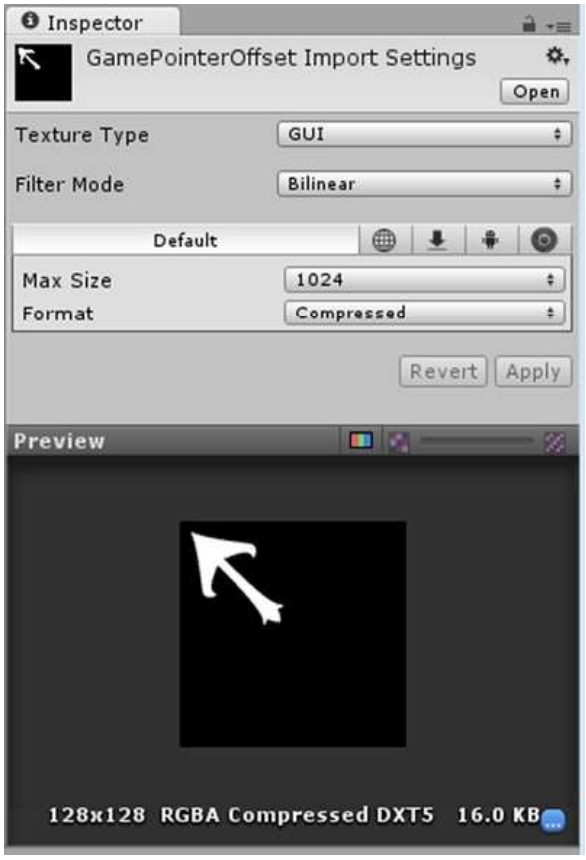


Figure 6-15. *The offset cursor texture*

9. Select the Control Center object in the Hierarchy view and drag the GamePointerOffset texture onto the GameManager's Default Cursor parameter.
10. Click Play.

The cursor image, or the upper-left corner of it to be exact, appears in the center of the viewport. Next, it needs to be matched to the operating system's location, just as you did with the GUI Texture version. As before, there will be some lag, but in this type of game, the pros outweigh the cons. Note how the `pos` variable, a Vector2 Type, uses dot notation to access its two values.

11. In the OnGUI function, change the DrawTexture line to the following:

```
var pos : Vector2 = Input.mousePosition; //get the location of the cursor
GUI.DrawTexture (Rect(pos.x, Screen.height - pos.y,64,64), currentCursor);
// draw the cursor there
```

12. Save the script, click Play, and test the new cursor.

In the Game view, with the hardware system cursor now matching the UnityGUI cursor, it's hard to tell how well it is working.

1. Stop Play mode.
2. From the Edit menu, select Game Settings, Player and click the Settings button on the Default Cursor thumbnail.
3. Select None from the top of the texture list.

Now when you see the ghosting in the Game view, you can tell that it is the hardware or operating system cursor.

You will see that the hardware cursor disappears when you are navigating the scene, but your new custom cursor does not. Thinking back, you may remember that you added code to your ScriptHolder script to hide and show the operating system cursor. Let's start by turning off the hardware cursor at startup.

4. At the top of the Start function, turn off the operating system cursor:

```
Screen.showCursor = false; // hide the operating system cursor
```

Now you will have to use part of the hide code from the earlier HideCursor script. Unlike regular objects, UnityGUI code is not enabled or activated, so it will require a conditional to allow it to be skipped over when not needed. You will run the conditional in the Update function as before, but will only switch a flag with it.

5. Add the flag variable for the navigating conditional:

```
var navigating : boolean;   //flag for navigation state
```

6. Add the code to set the flag in the Update function:

```
if (Input.GetAxis("Horizontal") || Input. GetAxis ("Vertical") ||
   Input. GetAxis ("Turn") || Input.GetButton("ML Enable") ){
  // a navigation key is being pressed
 navigating = true; // player is moving
 }
```

7. Just above the last curly bracket of the Update function, add:

```
else {
// no navigation keys are being pressed
navigating = false; // player is stationary
}
```

8. Wrap the DrawTexture code into the navigation conditional:

```
if (!navigating) { // if not navigating
   var pos : Vector2 = Input.mousePosition; //get the location of the cursor
   GUI.DrawTexture (Rect(pos.x, Screen.height - pos.y,64,64), currentCursor);
   // draw the cursor }
```

9. Save the script.
10. Click Play and test.

■ **Tip** Flags are merely variables whose values are watched for use in conditionals. A variable named `doorLocked` could be considered a flag; if `doorLocked` is true, the user will not be able to interact with the doorknob to open the door. If `doorLocked` is false, not only could the user open it, but other events (such as randomly spawned monsters coming through the door) could be generated as well.

Flags can track more than simple true/false states. A sidewalk café might need a flag for the weather. If it is sunny and hot, the variable weather = 0; if it is raining, weather = 1; and if it is overcast, weather = 2. If weather < 2, the umbrellas need to be put out. The weather constantly changes.

If you are still seeing the occasional flicker from the hardware cursor, there's one more trick you can use. Having the ability to assign your own texture means you can give it an almost fully alpha channel. A single pixel, not quite black, does the trick.

1. Select the Adventure Textures folder, right-click and select Import New Asset.
2. Bring in the AlmostBlankAlpha texture from the Chapter 6 Assets folder.
3. Set its Texture Type to **Cursor** and click Apply.
4. Load it in as the hardware cursor.

The hardware cursor is no longer showing, and the new cursor hides during navigation, so you are good to go.

Object-to-Object Communication

Now comes the tricky part and one of the most crucial concepts to understand in Unity: how to access one object's variables from another object's scripts. This will enable all of the individual objects to access generic settings, allow events on one object to trigger events on other objects, and generally tie all the game information together.

As mentioned, you can gain access to parents and children of GameObjects without too much trouble, but in order to allow "unrelated" objects to communicate, you must "introduce" them to each other first.

Hardcoding the objects into each other's code will require the object to exist in the scene; otherwise, you will get an error. Your game will have a few objects that must be named correctly, in order for them to be accessed this way. Hopefully, you have been heeding the warning about which object names you may not change. In this section, you will set up the functionality to keep track of whether the player is currently navigating through the scene or not in the GameManager script. This way, the action objects can access it whenever their scripts are activated.

While you could just duplicate the check-for-navigation keys being pressed in each script that needs that information, having the same condition checked in multiple scripts every frame (remember, it's inside an `Update` function) is a waste of resources and could slow frame rate. As it turns out, you've already done that twice (once for the HideCursor and once for the MouseLookRestricted), so you really ought only to calculate it once, if possible, and have whatever scripts need it access it from a single location.

FRAME RATE

When asked what affects frame rate (the number of frames per second that the engine can output), most people are quick to reply "poly count" (the number of polygons in the scene). While this is certainly a major contributor, in reality, almost everything contributes: the number of scripts being evaluated, the objects that are animating, the physics solutions being calculated, the GUI elements being rendered and monitored, the amount of textures stored in memory, the number of polygons within the view frustum at any one time, the network communications being sent back and forth, the shadows and shaders being calculated, and just about everything else. Unity Pro's Profiler is a great help in finding where resources are bottlenecking and can help you streamline your code for games that need high frame rates or need to run well on mobile platforms, where resources are more limited.

As you get deeper into your game creation, you will find other tasks and variables that will have to be accessed by multiple scripts, so you may as well use your GameManager script for that purpose.

GLOBAL VARIABLES

Some of you may already be familiar with the concept of global variables. Global variables in Unity are, at best, only global to the object they are on. While this makes it necessary for you to do a lot more work to access the variables from other objects, it makes it possible for the game to make better use of multiple processors and multithreading.

■ **Tip** Different scripts may use the same variable name, but it does not follow that the variables share the same value. To do so, one or both must update the other.

Vars and internal vars can share the same name as variables in other scripts. Even when they contain the same information, because they live in different scripts; they are not the same entity. Think of it as two people named John Smith. They both have the same name but live in different houses. John Smith on Maple Lane constantly checks with John Smith on Sixth Street to see if he is in his house or out traveling about. Depending on the latter's status, the former will take appropriate action.

In most cases, you can't refer to an object by name when declaring a variable. You *can*, however, drag and drop an object into the value field for the variable in the Inspector. While this may seem like an easy solution, when you start adding action objects to the game, it will quickly become tedious, dragging the same objects over and over.

It turns out that once the GameObjects have been loaded, you *can* find them by name, as long as they haven't been deactivated.

Mouseover Cursor Changes

Before you start working on mouseover events, you must have an understanding of how mouseover works. The most important concept is that rather than the cursor constantly firing rays into the scene and then evaluating what it's over, an object can report when the cursor is over it. While it is safe to say you will only want a limited number of objects to trigger the mouseover event, it will be well worth your while to make one generic reusable script that will

handle interactions. Because this script will get larger as you progress through the functionality of your game, let's start by defining some of your own functions, in order to keep things tidier.

Your first order of business is to work out the mechanics of your cursor color change. By definition, if you're going to change the color, you will have to define the default color, so you'll know what to return it to on mouseoff. As long as you're in the design process, you may as well plan on letting the author (and possibly the player, at some point) choose the mouseover color, rather than hardcode it in. Because that should be an option in the game, let's add it to the GameManager script.

1. Open the GameManager script.
2. At the top of the script add:

```
var mouseOverColor : Color = Color.green;  // mouse over color for the cursor
```

Note the use of a prepackaged color, `Color.green`. If you do a search through the scripting help for color and select the top choice, Color, you will see the other class variables.

Now you can easily change the color manually in the Inspector to the color of your choice at any time (Figure 6-16).

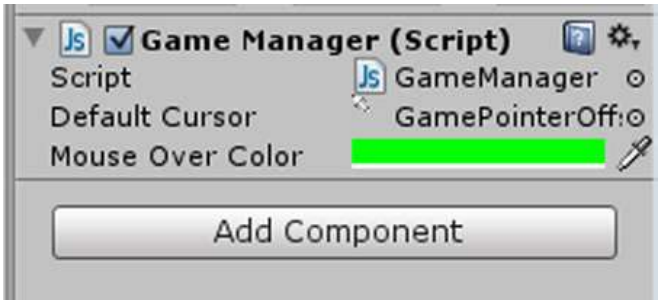


Figure 6-16. *The new mouseover color parameter*

You may decide to use your own cursor image, and that image may be more than just black-and-white. The best choice for the default color is white, so let's consider white the default color, as it will not interfere with the texture's image.

Now that the GameManager script knows what color to use for mouseover, you have to create your own function that can be called from other objects to change the cursor color when the mouse goes over and off the object. As you have seen before, user-defined functions look the same as the system and event functions, except that you can choose their names. You'll call this function `CursorColorChange`. It will be a simple toggle to switch back and forth between the default white color and the specified mouseover color.

Let's also look at a new concept, that of passing an *argument* or value to a variable. You used this with the `OnTriggerEnter` function when it reported the object that intersected with it, but this time, you'll be in charge of passing the argument as well as receiving it. You will start by putting the name of the variable or argument and its type inside the function's parentheses.

For the argument, you'll tell the function what state the cursor is in when you call the function, so it will know which color to set it to next. To change the color of a UnityGUI element, you'll need a variable to keep track of the current cursor color.

1. In the GameManager script, add the following variable:

```
internal var currentCursorColor : Color; // current tint color of cursor
```

2. Initialize the variable in the Start function:

```
currentCursorColor = Color.white; // start color to white
```

3. Add the following below the Update function:

```
function CursorColorChange (colorize: boolean) {

   if (colorize)  currentCursorColor = mouseOverColor;

   else  currentCursorColor = Color.white;
}
```

Unlike a regular gameObject, tinting UnityGUI elements changes all elements after the color change. First, you will assign the current color above the GUI.DrawTexture line, then you will change it back

4. Inside the OnGUI function, add the following above the GUI.DrawTexture line:

```
GUI.color = currentCursorColor; // set the cursor color to current
```

5. Below the GUI.DrawTexture line, add:

```
GUI.color = Color.white; // set the cursor color back to default
```

6. Save the script.

In using a Boolean type for the value that you pass into your function, you don't really have to think of it as true or false; it's just as useful to think of it as 0 or 1. The cursor is either white or the mouseover color, one of two states. Let's consider the default color, white, to be state 0 and the mouseover color to be state 1. Reading through the function, if it was passed a false (0), it will change the color to the mouseover color; otherwise (else), it was passed a true (1), so it will turn it back to white. Note that you don't have to tell it the argument is a variable, but you do have to define its Type.

You have had no control over the functions you have previously used; Awake and Start are called when you first start the game, and Update and FixedUpdate are called every frame or at least are dependent on frame rate. A user-defined function, in contrast, can be called whenever you wish.

You will be calling the CursorColorChange function from inside a couple of system functions, OnMouseEnter and OnMouseExit. But you first have to make a very generic script to put on every action object.

1. In the Adventurer Scripts folder, create a new JavaScript.
2. Name it **Interactor** and open it in the script editor.
3. Delete the default Update function.

This script will have to talk to the Control Center object to call the function you just created, so it needs to be "introduced" first. In the SendMessageOnTrigger script you created, you had to manually drag the target object onto its parameter in the Inspector. While that's fine for a one-off situation, you'll quickly get tired of repeating the same setup task for multiple objects. This time, you will be having the code do the assignment for you, using Find.

Rather than having to drag and drop it into the script every time you add it to a new action object, you can let Unity search the scene and Find it for you by name. Because Find searches the entire scene, it shouldn't be used in the Update function or any other place where it can adversely affect frame rate. Using it once in the Start function, you won't be slowing the rest of the game down.

4. Add the following variable declaration:

```
// Gain access to these objects
internal var controlCenter : GameObject; // the Control Center object
```

5. Find and assign the Control Center in the `Start` function:

```
controlCenter = GameObject.Find("Control Center"); // locate by name and assign the
object to the var
```

Next, you'll trigger the color change function on the GameManager script using a `SendMessage` when the cursor mouses over the object that holds the Interactor script.

6. Add the following function below the `Start` function:

```
function OnMouseEnter () {

    controlCenter.SendMessage("CursorColorChange", true); // colorize the pointer
}
```

7. Save the script.

`OnMouseEnter` and `OnMouseExit` are similar to the `OnTriggerEnter` and `OnTriggerExit` functions you used in the previous chapter. They are all event-type functions. As with their `OnTrigger` cousins, the `OnMouse` functions also require that the object has a Collider, before they will work.

When you used the `SendMessage` function or method in the previous chapter, the only thing you sent was the name of the function you wanted to call. This time, separated by a comma, you send the argument that your function is expecting, a type Boolean.

■ **Tip** User-defined functions can hold as many arguments as needed. `SendMessage`, however, can only send a single argument with the message.

Before you can see the fruits of all your labor, you'll have to create a test object to mouse over.

1. From the GameObject menu, Create Other, select Sphere.
2. Move the sphere in the Scene view until you can see it in the Game view (Figure 6-17).

Figure 6-17. *The sphere in view*

3. Drag the Interactor script onto the Sphere.
4. Select the Sphere.
5. Click Play and test the functionality.

The cursor changes color as it moves over the sphere, proving that the sphere was in contact with the Control Center object and sent the message that activated the color change function.

Next, you'll want to revert the color when the cursor moves off the object.

6. Add the following code to the Interactor script beneath the `OnMouseEnter` function:

```
function OnMouseExit () {

   controlCenter.SendMessage("CursorColorChange", false); // turn the pointer white

}
```

7. Save the script.

This function, as you would expect, is called when the cursor passes off the action object. Because your `CursorColorChange` function can handle either state, you send it a false this time, to tell it to carry out the `else` clause.

8. Click Play and test the mouseover functionality.

Everything works as expected (Figure 6-18).

Figure 6-18. *The mouseover color change*

Now comes the fun part. This is where you see the beauty of making scripts that can be used by multiple and distinct objects.

1. From the GameObject menu, create a Cube and a cylinder near the Sphere.
2. Select the Sphere and in the Inspector, right-click over the Interactor script component's name and choose Copy Component.
3. Select the Cube and the Cylinder; position the cursor over any component name (Transform will work fine); right-click and choose Paste Component As New.

The Interactor component, with its current parameter settings and assignments, is copied to the two other objects.

4. Click Play and test the mouseover functionality with the new objects (Figure 6-19).

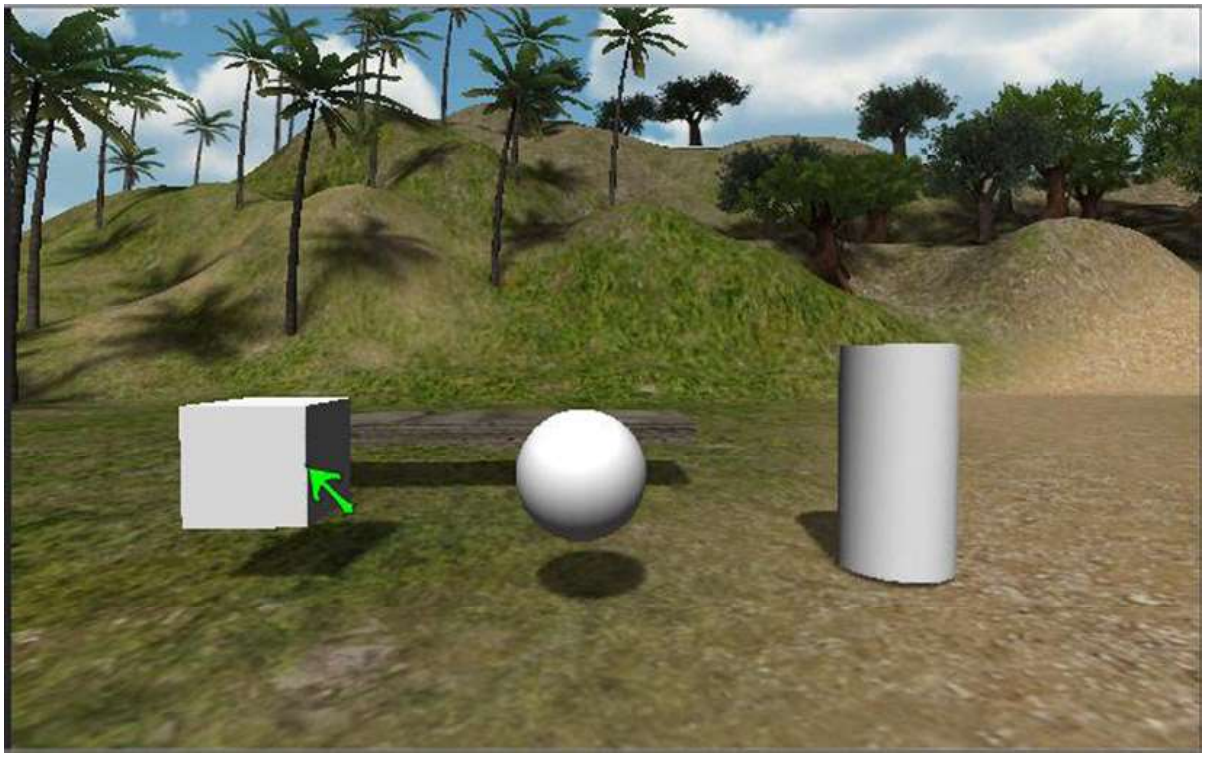


Figure 6-19. *The three action objects*

While the other scripts you have built will be reusable in future games, the Interactor script is a specialized script that will be applied to every action item in this particular game.

By exposing variables to the Inspector, you can customize the events associated with the individual action object, without changing the actual Interactor script. An example of this is an informative (or entertaining) message on mouseover, where the message differs for each object ("It is a solid-looking cube, approximately 1 meter square" or "It is a featureless sphere suspended in thin air"). Eventually, the Interactor script will contain the functionality to deal with all types of interaction, no matter what action object it resides on.

Because the same script is on all three objects, when you make a change to it, all objects using it are updated.

5. Back in the Game view, move backward from the objects you have added until they are only a few pixels in the viewport.

6. Stop and test the mouseover functionality.

It still works (Figure 6-20), which is probably not something you want.

Figure 6-20. *Mouseover triggered from far, far away…*

Distance Check

In the early days of point-and-click adventures, the game was based on a series of beautifully rendered high-resolution still shots (Cyan's Riven) or hand-painted toon-style backgrounds (LucasArts' Curse of Money Island). Hotspots (the area on the image where the item was in the picture) were created manually for each still shot.

In real-time 3D, the action object itself, because of its collider, becomes the hotspot, whether it is near or far. As you have discovered, this allows for the possibility of opening doors or picking up buckets, while several hundred meters away. While this could be part of the design in a game, to prevent your player from triggering things he or she can't even see, you'll have to limit the distance within which the player can interact with an object.

First, you'll have to calculate the distance the First Person Controller is from the action object, and then you'll check to see if it is within mouseover range. If it isn't within range, you must skip the mouseover functionality.

Let's start by creating a function that will calculate the distance from the object to the camera. This is a very typical user-defined function; it can be found in almost any game with enemies that can turn to face you (and shoot, if you are within range). This one uses some vector math, the dot product, to calculate the distance the camera is from the action object the script is on. The nice thing is that you don't have to understand the math that makes the script work to reap the benefit.

■ **Tip** It pays to watch or read through tutorials for other game genres (even if they don't interest you), because they may contain intriguing functionality that you can use in your game.

---

1. In the Interactor script, add the following user-defined function:

```
function DistanceFromCamera () {

   // get the direction the camera is heading so you only process stuff in the line of sight
   var heading : Vector3 = transform.position - cam.transform.position;
   //calculate the distance from the camera to the object
   var distance : float = Vector3.Dot(heading, cam.transform.forward);
   return distance;
}
```

■ **Note** In this function, you see the use of a `Vector3` type variable. It expects a three-part value, such as (X,Y,Z) or (R,G,B). A `Vector2` might expect (X,Y), and a `Vector4` might expect (R,G,B,A).

There's something else that's new in this function: it *returns* a value. When you look at how the function is called, you will notice that the value returned by the function is being assigned directly to a variable.

Before you can get this working, however, you will have to define `cam`, the camera it is referencing.

2. Add the following internal variable beneath the controlCenter variable declaration:

```
internal var cam : GameObject; // the main camera on the first person controller
```

3. Then add the following to the `Start`:

```
cam = GameObject.Find("Main Camera"); // find and assign the Main Camera to this variable
```

The last line inside the `DistanceFromCamera` function says `return distance`. The variable `distance` was declared and assigned a value in the previous line, so that is nothing new. But the word *return* is a reserved word.

Variables declared inside functions are *local* to that particular function. They are not accessible outside the function and are cleared on completion of the function. So in order to be able to access the distance variable, it would either have to be declared up with the regular script variables, or its value passed back out to whoever called the function in the first place.

On its own, *return* simply tells you to leave the function (and that will be useful very shortly). But when coupled with a variable, it sends the value of that variable back to where the function was called, before the local variable is cleared.

Right now, you want to check the distance from the camera, before you allow the cursor color change to happen.

1. Add the following lines of code *above* the `controlCenter.SendMessage` line inside the `OnMouseEnter` function:

```
if (DistanceFromCamera() > 7) return;
```

You call the function with `DistanceFromCamera()`, and it returns the distance it calculated in the function. You then immediately check to see if that returned value or distance is less than 7 meters. If not, the `return` bumps you out of the `OnMouseEnter` function before the color change function can be called. If it's evaluated as less than 7 meters, the rest of the contents of the `OnMouseEnter` function are carried out, and your cursor's color is changed.

You may decide that 7 meters is not the optimal distance for the mouseover to work. Rather than digging through the code each time to change it, it would make sense to create a variable to hold the distance you want to declare. Creating a variable near the top of the script will make it easier to change, as you add more functionality to your script, but if you expose it to the Inspector, it will also allow you to change the distance on an individual action object basis, which could possibly be beneficial.

2. Add the following line up at the top, above the other public variables:

```
// Pick and Mouseover Info
internal var triggerDistance : float = 7.0; // distance the camera must be to
the object before mouse over
```

3. Change the line in the `OnMouseEnter` function to use the variable, instead of the hardcoded value:

```
if (DistanceFromCamera() > triggerDistance) return;
```

You may decide that manually changing the trigger distance is a tedious way to hone in on the optimal distance. If you temporarily have the distance print out to the console, you can decide very quickly what it ought to be.

4. Add the following line above the `if` clause in the `OnMouseEnter` function:

```
print (DistanceFromCamera());
```

5. Save the Script.
6. Click Play.
7. Experiment with the trigger distance (Figure 6-21), by watching the results in the status line.

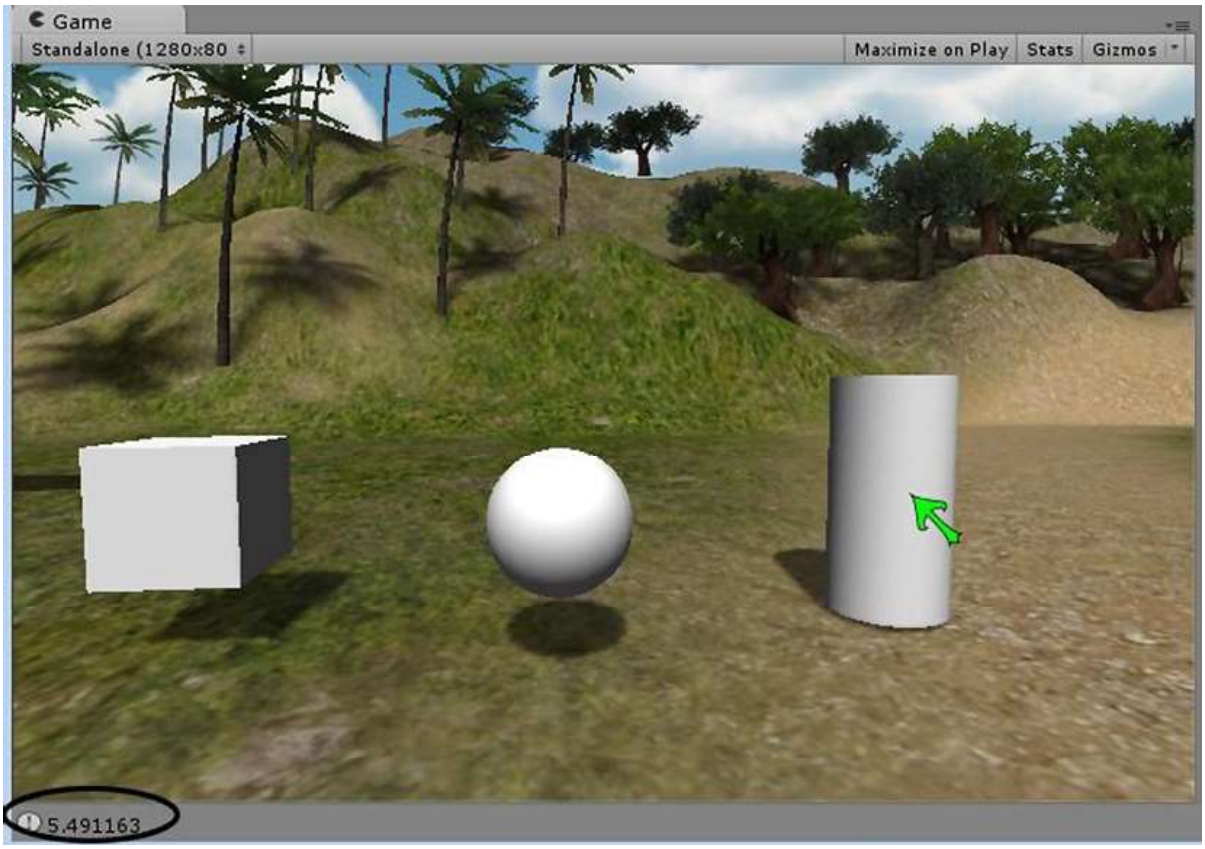


Figure 6-21. *Reporting distance on mouseover*

8. When you are happy with it, change the `triggerDistance` variable to your optimal value and remove `internal`, so the variable is exposed to the Inspector.
9. Remove or comment out the print line.
10. Save the script again.
11. Exit Play mode.

■ **Tip** As soon as you change a value in the Inspector, it will always override the value initialized in the script. If you change the value in the script and want to reset the value in the Inspector, you can change the variable to internal, then change it back again. This will effectively reset the default value in the Inspector.

This is also the first place to check, if you are changing the value of a variable inside the script but seeing no effect. Make sure that variable is set to internal.

Quick Publish

At this point, you may be getting itchy to see how everything runs, so let's take a few minutes to build an executable and try it out.

Because you are now using a different scene, CursorControl, you will have to Add Current in the Scenes to Build and uncheck TerrainTest.

1. Save the scene and the project.
2. From the File menu, select Build Settings.
3. Click Add Current.
4. Uncheck the other scenes in the Scenes in Build window (Figure 6-22).

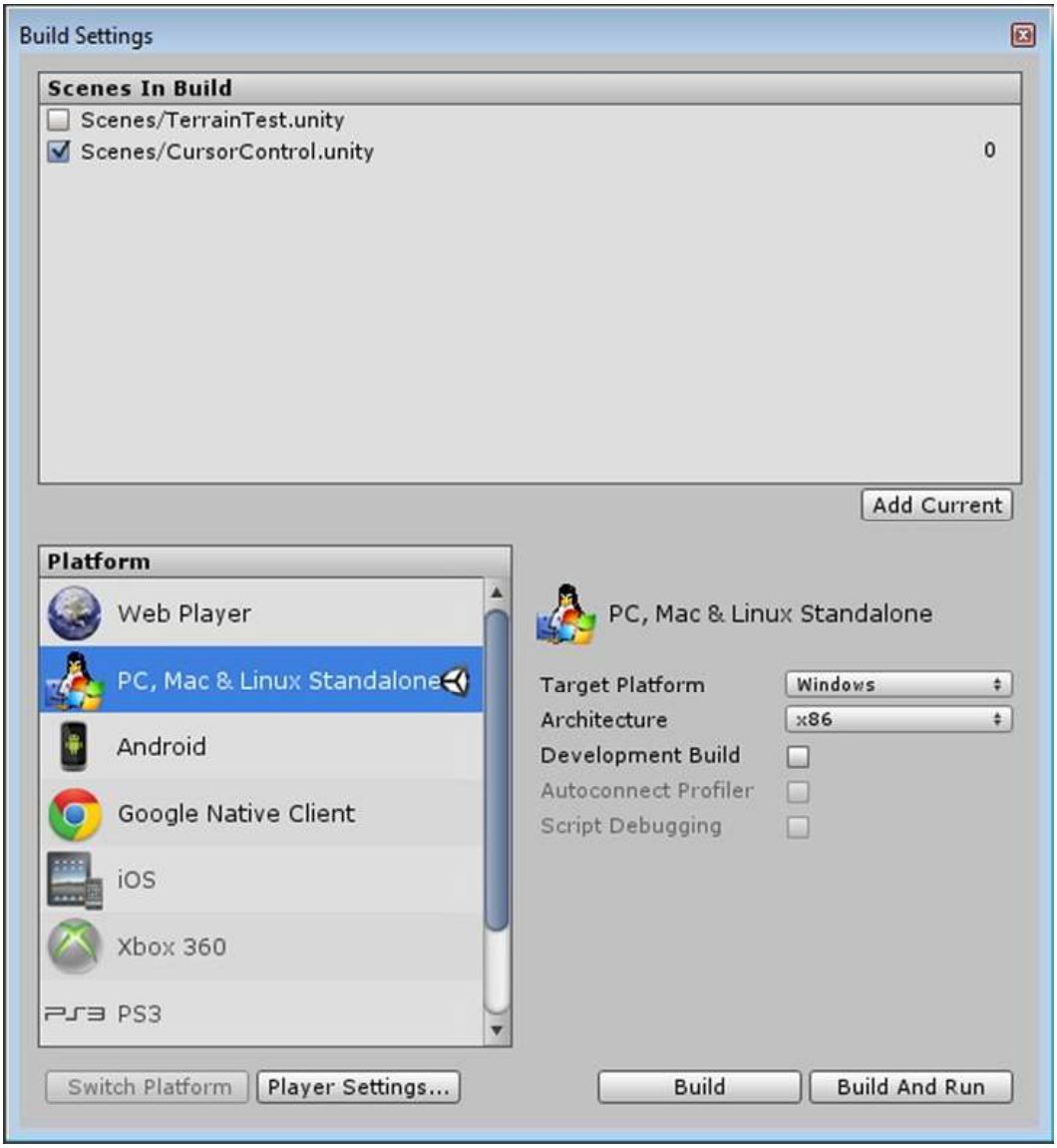


Figure 6-22. *Scenes in Build window*

5. Click Build And Run and save this exe as CursorControl.exe.
6. Test the build.

With the cursor responding to objects in the scene, your little scene is starting to look like a real game.

7. Close the Build Settings dialog.

Object Reaction to Mouseover

Another means of action-object identification, often seen in more stylized point-and-click adventures, such as Tales of Monkey Island, is to have the object itself change color. This method is particularly advantageous on mobile devices, such as phones, where the screen is small, as it provides an additional visual cue.

Owing to the need to include new features in the second version of this book, I have removed the section on color changes in action objects with a mouseover event.

If you are interested in attempting this functionality on your own at some point, you will have to get the object's material with: `GetComponent(MeshRenderer).material;`.

Substituting a new material on mouseover, you will realize that you could probably use the same material for all of the objects and merely change the texture. You can find the texture with: `GetComponent(MeshRenderer).material.mainTexture;`.

Using a Vertex Lit shader for the mouseover material will allow you to turn the Emissive value of the shader up, so the object is no longer affected by scene lighting, making it stand out nicely.

As long as each action item has a single material that has a main texture, the scripting stays fairly simple. If, however, only some objects and their materials meet the requirements, you will have to script around the differences. Another issue is that not every object has a MeshRenderer. Characters, for example, have a SkinnedMeshRenderer, and interactive cloth has a ClothRenderer. So you would have to find out which kind of renderer an action object had before you could get its material . . . or materials. You can see how the logic behind the object color change can get quite complicated.

Summary

In this chapter, you experimented with cursor control, finding that you could hide the operating system's cursor with a simple `Screen.showCursor = false`. After creating a variable called `navigating`, which checked your virtual buttons to see if any were active, you were able turn the cursor off while moving and back on as soon as you stopped.

To allow for more artistic license, you substituted a GUITexture object as a cursor with the help of the Texture Importer presets. You learned that GUI objects in Unity are not sized relevant to screen size and, so, explored forcing screen resolution via scripting. By getting the operating system's cursor's position with `Input.mousePosition`, you were able to set your own cursor's Pixel Inset values with `guiTexture.pixelInset.x` and `guiTexture.pixelInset.y`.

A downside to the GUI Texture cursor was a bit of lag, so you next tried using your own texture for the operating system or hardware cursor. It had great responsiveness and was guaranteed to draw on top of everything else in the scene but was too limited in size and color options.

Your final cursor test involved a purely scripted cursor, using the UnityGUI system. It also replaced the operating system cursor, so there was a bit of lag, but it had the ability to have its color and size adjusted, as needed. You learned that it will be able to draw on top of the rest of the scene with no problems.

Next, you started designing a script that would hold and manage the master game information and be able to communicate with other object's scripts. You found that after creating a variable to hold a GameObject, you could either drag one into it in the Inspector or use `GameObject.Find` in the `Start` function to locate the object you wanted to communicate with. Names, you discovered, are only properties of objects, materials, and components and representatives of their objects. The same variable names used in different scripts don't store the same values, unless you specifically update them.

With the introduction of `OnMouseEnter` and `OnMouseExit`, you were able to instigate color changes for your cursor, as it passed over and off your action objects. By using `GetComponent`, you found you could access variables from the script components you created, allowing you to keep important data on a single script that could be accessed by all.

In the next chapter, you'll work with imported objects and their animations and start to get them acting like actual game assets.