16 DEVELOPING THE MR. STICK MAN GAME

Now that we've created the images for our Mr. Stick Man Races for the Exit game, we can begin to develop the code. The description of the game in the previous chapter gives us a basic idea of what we need: a stick figure that can run and jump, and platforms that he must jump to.

We'll need code to display the stick figure and move it across the screen, as well as to draw platforms. But before we write that code, we need to create the canvas to display our background image.

CREATING THE GAME CLASS

First, we'll create a class called Game, which will be our program's main controller. The Game class will have an __init__ function for initializing the game and a mainloop function for doing the animation.

SETTING THE WINDOW TITLE AND CREATING THE CANVAS

In the first part of the __init__ function, we'll set the window title and create the canvas. As you'll see, this part of the code is similar to the code that we wrote for the Bounce! game in Chapter 13. Open your editor and enter the following code, and then save your file as *stickmangame.py*. Make sure you save it in the directory we created in Chapter 15 (called *stickman*).

```
from tkinter import *
import random
import time

class Game:
    def __init__(self):
        self.tk = Tk()
        self.tk.title("Mr. Stick Man Races for the Exit")
        self.tk.resizable(0, 0)
        self.tk.wm_attributes("-topmost", 1)
        self.canvas = Canvas(self.tk, width=500, height=500, \
                highlightthickness=0)
        self.canvas.pack()
        self.tk.update()
        self.canvas_height = 500
        self.canvas_width = 500
```

In the first half of this program (from from tkinter import * to self.tk.wm_attributes), we create the tk object and then set the window title with self.tk.title to ("Mr. Stick Man Races for the Exit"). We make the window fixed (so it can't be resized) by calling the resizable function, and then we move the window in front of all other windows with the wm_attributes function.

Next, we create the canvas with the self.canvas = Canvas line, and call the pack and update functions of the tk object. Finally, we create two variables for our Game class, height and width, to store the height and width of the canvas.

NOTE *The backslash (\) in the* `self.canvas = Canvas` *line is used only to separate the long line of code. It's not required, but I've included it here for readability since the entire line won't fit on the page.*

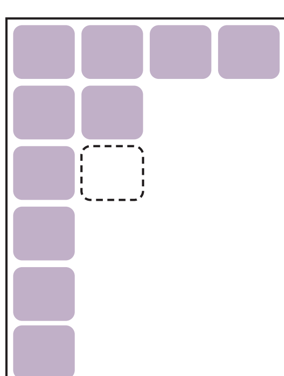
FINISHING THE __INIT__ FUNCTION

Now enter the rest of the `__init__` function into the *stickfiguregame .py* file that you just created. This code will load the background image and then display it on the canvas:

```
        self.tk.update()
        self.canvas_height = 500
        self.canvas_width = 500
❶       self.bg = PhotoImage(file="background.gif")
❷       w = self.bg.width()
        h = self.bg.height()
❸       for x in range(0, 5):
❹           for y in range(0, 5):
❺               self.canvas.create_image(x * w, y * h, \
                        image=self.bg, anchor='nw')
❻       self.sprites = []
        self.running = True
```

At ❶, we create the variable `bg`, which contains a `PhotoImage` object—the background image file called *background.gif* that we created in Chapter 15. Next, beginning at ❷, we store the width and height of the image in the variables `w` and `h`. The `PhotoImage` class functions `width` and `height` return the size of the image once it has been loaded.

Next come two loops inside this function. To understand what they do, imagine that you have a small square rubber stamp, an ink pad, and a large piece of paper. How are you going to fill the paper with colored squares using the stamp? Well, you could just randomly cover the page with stamps until it's filled. The result would be a mess, and it would take a while to complete, but it would fill the page. Or you could start stamping down the page in a column and then move back to the top and start stamping down the page in the next column, as shown on the right.



The background image we created in the previous chapter is our stamp. We know that the canvas is 500 pixels across and 500 pixels down, and that we created a background image of 100 pixels square. This tells us that we need five columns across and five rows down to fill the screen with images. We use the loop at ❸ to calculate the columns across, and the loop at ❹ to calculate rows going down.

At ❺, we multiply the first loop variable `x` by the width of the image (`x * w`) to determine how far across we're drawing, and then multiply the second loop variable `y` by the height of the image (`y * h`) to calculate how far down to draw. We use the `create_image` function of the `canvas` object (`self.canvas.create_image`) to draw the image on the screen using those coordinates.

Finally, beginning with ❻, we create the variables `sprites`, which holds an empty list, and `running`, which contains the Boolean value `True`. We'll use these variables later in our game code.

CREATING THE MAINLOOP FUNCTION

We'll use the `mainloop` function in the `Game` class to animate our game. This function looks a lot like the main loop (or animation loop) we created for the Bounce! game in Chapter 13. Here it is:

```
        for x in range(0, 5):
            for y in range(0, 5):
                self.canvas.create_image(x * w, y * h, \
                    image=self.bg, anchor='nw')
        self.sprites = []
        self.running = True

    def mainloop(self):
❶       while 1:
❷           if self.running == True:
❸               for sprite in self.sprites:
❹                   sprite.move()
❺           self.tk.update_idletasks()
            self.tk.update()
            time.sleep(0.01)
```

At ❶, we create a `while` loop that will run until the game window is closed. Next, at ❷, we check to see if the variable `running` is equal to `True`. If it is, we loop through any sprites in the list of sprites (`self.sprites`) at ❸, calling the function `move` for each one at ❹. (Of course, we have yet to create any sprites, so this code

wouldn't do anything if you ran the program now, but it will be useful later.)

The last three lines of the function, beginning at ❺, force the tk object to redraw the screen and sleep for a fraction of a second, as we did with the Bounce! game in Chapter 13.

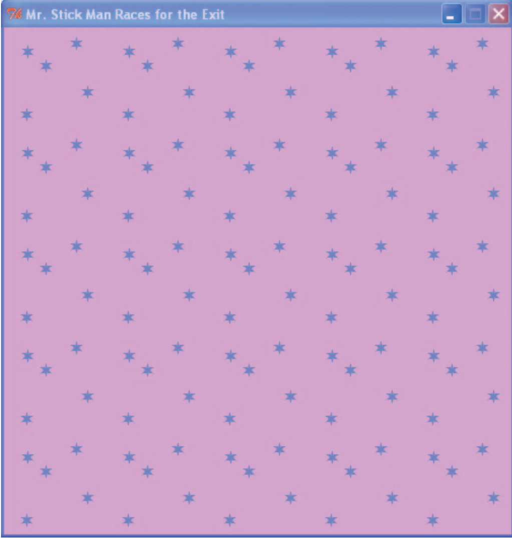
So that you can run this code, add the following two lines (note that there's no indentation required for these two lines) and save the file.

```
g = Game()
g.mainloop()
```

NOTE *Be sure to add this code to the bottom of your game file. Also, make sure that your images are in the same directory as the Python file. If you created the* stickman *directory in Chapter 15 and saved all your images there, the Python file for this game should be there as well.*

This code creates an object of the Game class and saves it as the variable g. We then call the mainloop function on the new object to draw the screen.

Once you've saved the program, run it in IDLE by choosing **Run ▸ Run Module**. You will see a window appear with the background image filling the canvas.

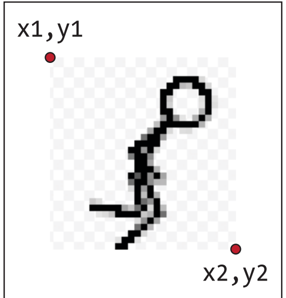


We've added a nice background for our game, and created an animation loop that will draw sprites for us (once we've created them).

CREATING THE COORDS CLASS

Now we'll create the class that we'll use to specify the position of something on our game screen. This class will store the top-left (`x1` and `y1`) and bottom-right (`x2` and `y2`) coordinates of any component of our game.

Here's how you might record the position of the stick figure image using these coordinates:



We'll call our new class `Coords`, and it will contain only an `__init__` function, where we pass the four parameters (`x1`, `y1`, `x2`, and `y2`). Here's the code to add (put it at the beginning of the *stickmangame.py* file):

```
class Coords:
    def __init__(self, x1=0, y1=0, x2=0, y2=0):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
```

Notice that each parameter is saved as an object variable of the same name (`x1`, `y1`, `x2`, and `y2`). We'll be using objects of this class shortly.

CHECKING FOR COLLISIONS

Once we know how to store the position of our game sprites, we need a way to tell if one sprite has collided with another, like when Mr. Stick Man jumps around the screen and bangs into one of the platforms. To make this problem easier to solve, we can break it down into two smaller problems: checking if sprites are colliding vertically and checking if sprites are colliding horizontally. Then we can combine our two smaller solutions to easily see if two sprites are colliding in any direction!

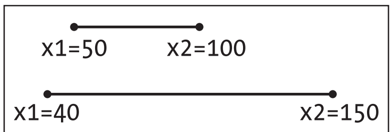
SPRITES COLLIDING HORIZONTALLY

First, we'll create the `within_x` function to determine if one set of x coordinates (`x1` and `x2`) has crossed over another set of x coordinates (again, `x1` and `x2`). There's more than one way to do this, but here's a simple approach which you can add just below the `Coords` class:

```
class Coords:
    def __init__(self, x1=0, y1=0, x2=0, y2=0):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

def within_x(co1, co2):
❶    if co1.x1 > co2.x1 and co1.x1 < co2.x2:
❷        return True
❸    elif co1.x2 > co2.x1 and co1.x2 < co2.x2:
❹        return True
❺    elif co2.x1 > co1.x1 and co2.x1 < co1.x2:
        return True
❻    elif co2.x2 > co1.x1 and co2.x2 < co1.x2:
        return True
❼    else:
❽        return False
```

The `within_x` function takes the parameters `co1` and `co2`, both `Coords` objects. At ❶, we check to see if the leftmost position of the first coordinate object (`co1.x1`) is between the leftmost position (`co2.x1`) and the rightmost position (`co2.x2`) of the second coordinate object. We return `True` at ❷ if it is.

Let's take a look at two lines with overlapping x coordinates to understand how this works. Each line starts at `x1` and finishes at `x2`.



The first line in this diagram (`co1`) starts at pixel position 50 (`x1`) and finishes at 100 (`x2`). The second line (`co2`) starts at position 40 and finishes at 150. In this case, because the `x1` position of the first line is between the `x1` and `x2` positions of the second line, the first `if` statement in the function would be true for these two sets of coordinates.

With the `elif` at ❸, we see whether the rightmost position of the first line (`co1.x2`) is between the leftmost position (`co2.x1`) and rightmost position (`co2.x2`) of the second. If it is, we return `True` at ❹. The two `elif` statements at ❺ and ❻ do almost the same thing: They check the leftmost and rightmost positions of the second line (`co2`) against the first (`co1`).

If none of the `if` statements match, we reach `else` at ❼, and return `False` at ❽. This is effectively saying, "No, the two coordinate objects do not cross over each other horizontally."

To see an example of the function working, look back at the diagram showing the first and second lines. The `x1` and `x2` positions of the first coordinate object are 40 and 100, and the `x1` and `x2` positions of the second coordinate object are 50 and 150. Here's what happens when we call the `within_x` function that we wrote:

```
>>> c1 = Coords(40, 40, 100, 100)
>>> c2 = Coords(50, 50, 150, 150)
>>> print(within_x(c1, c2))
True
```

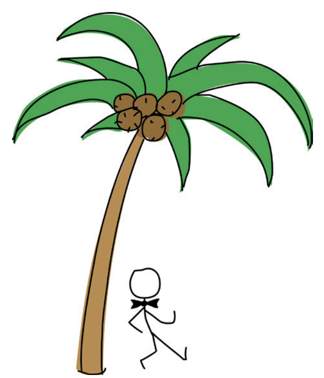
The function returns `True`. This is the first step to being able to determine whether one sprite has bumped into another. For example, when we create a class for Mr. Stick Man and for the platforms, we will be able to tell if their x coordinates have crossed one another.

It's not really good programming practice to have lots of `if` or `elif` statements that return the same value. To solve this problem, we can shorten the `within_x` function by surrounding each of its conditions with parentheses, separated by the `or` keyword. If you want a slightly neater function, with a few less lines of code, you can change the function so it looks like this:

```
def within_x(co1, co2):
    if (co1.x1 > co2.x1 and co1.x1 < co2.x2) \
            or (co1.x2 > co2.x1 and co1.x2 < co2.x2) \
            or (co2.x1 > co1.x1 and co2.x1 < co1.x2) \
            or (co2.x2 > co1.x1 and co2.x2 < co1.x2):
        return True
    else:
        return False
```

To extend the `if` statement across multiple lines so that we don't end up with one really long line containing all the conditions, we use a backslash (\), as shown above.

SPRITES COLLIDING VERTICALLY



We also need to know if sprites collide vertically. The `within_y` function is very similar to the `within_x` function. To create it, we check whether the `y1` position of the first coordinate has crossed over the `y1` and `y2` positions of the second, and then vice versa. Here's the function to add (put it below the `within_x` function)—this time we'll write it using the shorter version of the code (rather than lots of `if` statements):

```
def within_y(co1, co2):
    if (co1.y1 > co2.y1 and co1.y1 < co2.y2) \
            or (co1.y2 > co2.y1 and co1.y2 < co2.y2) \
            or (co2.y1 > co1.y1 and co2.y1 < co1.y2) \
            or (co2.y2 > co1.y1 and co2.y2 < co1.y2):
        return True
    else:
        return False
```

Putting It All Together: Our Final Collision-Detection Code

Once we've determined whether one set of x coordinates has crossed over another, and done the same for y coordinates, we can write functions to see whether a sprite has hit another sprite and on which side. We'll do this with the functions `collided_left`, `collided_right`, `collided_top`, and `collided_bottom`.

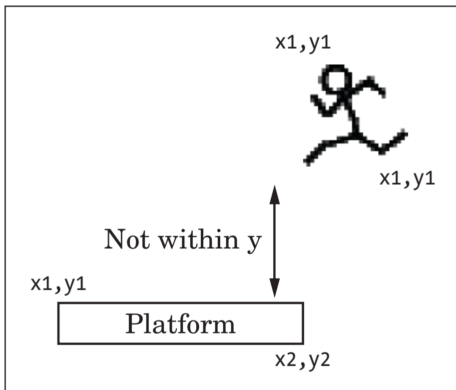
The collided_left Function

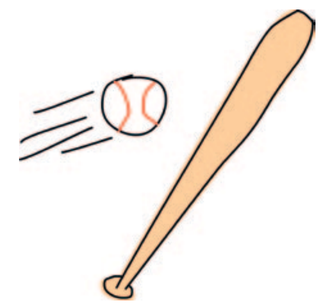
Here's the code for the `collided_left` function, which you can add below the two `within` functions we just created:

```
❶ def collided_left(co1, co2):
❷     if within_y(co1, co2):
❸         if co1.x1 <= co2.x2 and co1.x1 >= co2.x1:
❹             return True
❺     return False
```

This function tells us whether the left-hand side (the `x1` value) of a first coordinate object has hit another coordinate object.

The function takes two parameters: `co1` (the first coordinate object) and `co2` (the second coordinate object). As you can see at ❶, we check whether the two coordinate objects have crossed over vertically, using the `within_y` function at ❷. After all, there's no point in checking whether Mr. Stick Man has hit a platform if he is floating way above it, like this:





At ❸, we see if the value of the leftmost position of the first coordinate object (`co1.x1`) has hit the `x2` position of the second coordinate object (`co2.x2`)—that it is less than or equal to the `x2` position. We also check to make sure that it hasn't gone past the `x1` position. If it has hit the side, we return `True` at ❹. If none of the `if` statements are true, we return `False` at ❺.

THE COLLIDED_RIGHT FUNCTION

The `collided_right` function looks a lot like `collided_left`:

```
def collided_right(co1, co2):
❶     if within_y(co1, co2):
❷         if co1.x2 >= co2.x1 and co1.x2 <= co2.x2:
❸             return True
❹     return False
```

As with `collided_left`, we check to see if the y coordinates have crossed over each other using the `within_y` function at ❶. We then check to see if the `x2` value is between the `x1` and `x2` positions of the second coordinate object at ❷, and return `True` at ❸ if it is. Otherwise, we return `False` at ❹.

THE COLLIDED_TOP FUNCTION

The `collided_top` function is very similar to the two functions we just added.

```
def collided_top(co1, co2):
❶     if within_x(co1, co2):
❷         if co1.y1 <= co2.y2 and co1.y1 >= co2.y1:
              return True
      return False
```

The difference is that this time, we check to see if the coordinates have crossed over horizontally, using the `within_x` function at ❶. Next, at ❷, we see if the topmost position of the first coordinate (`co1.y1`) has crossed over the `y2` position of the second coordinate, but not its `y1` position. If so, we return `True` (meaning that yes, the top of the first coordinate has hit the second coordinate).

THE COLLIDED_BOTTOM FUNCTION

Of course, you knew that one of these four functions had to be just a bit different, and it is. Here's the collided_bottom function:

```
def collided_bottom(y, co1, co2):
❶    if within_x(co1, co2):
❷        y_calc = co1.y2 + y
❸        if y_calc >= co2.y1 and y_calc <= co2.y2:
❹            return True
❺    return False
```

This function takes an additional parameter, y, a value that we add to the y position of the first coordinate. At ❶, we see if the coordinates have crossed over horizontally (as we did with collided_top). Next, we add the value of the y parameter to the first coordinate's y2 position, and store the result in the variable y_calc at ❷. If at ❸ the newly calculated value is between the y1 and y2 values of the second coordinate, we return True at ❹ because the bottom of coordinate co1 has hit the top of coordinate co2. However, if none of the if statements are true, we return False at ❺.

We need the additional y parameter because Mr. Stick Man could fall off a platform. Unlike with the other collided functions, we need to be able to test to see if he *would* collide at the bottom, rather than whether he already has. If he walks off a platform and keeps floating in midair, our game won't be very realistic; so as he walks, we check to see if he has collided with something on the left or right. However, when we check below him, we see if he would collide with the platform; if not, he needs to go crashing down!

CREATING THE SPRITE CLASS

We'll call the parent class for our game items Sprite. This class will provide two functions: move to move the sprite and coords to return the sprite's current position on the screen. Here's the code for the Sprite class.

```
class Sprite:
❶    def __init__(self, game):
❷        self.game = game
❸        self.endgame = False
❹        self.coordinates = None
```

```
❺     def move(self):
❻         pass
❼     def coords(self):
❽        return self.coordinates
```

The Sprite class's __init__ function defined at ❶ takes a single parameter: game. This parameter will be the game object. We need it so that any sprite we create will be able to access the list of other sprites in the game. We store the game parameter as an object variable at ❷.

At ❸, we store the object variable endgame, which we'll use to indicate the end of the game. (At the moment, it's set to False.) The final object variable, coordinates at ❹, is set to nothing (None).

The move function defined at ❺ does nothing in this parent class, so we use the pass keyword in the body of this function at ❻. The coords function at ❼ simply returns the object variable coordinates at ❽.

So our Sprite class has a move function that does nothing and a coords function that returns no coordinates. It doesn't sound very useful, does it? However, we know that any classes that have Sprite as their parent will always have move and coords functions. So, in the main loop of the game, when we loop through a list of sprites, we can call the function move, and it won't cause any errors. Why not? Because each sprite has that function.

NOTE *Classes with functions that don't do very much are actually quite common in programming. In a way, they're a kind of agreement or contract that makes sure all the children of a class provide the same sort of functionality, even if in some cases the functions in the child classes do nothing.*

ADDING THE PLATFORMS

Now we'll add the platforms. We'll call our class for platform objects PlatformSprite, and it will be a child class of Sprite. The __init__ function for this class will take a game parameter (as the Sprite

parent class does), as well as an image, x and y positions, and the image width and height. Here's the code for the PlatformSprite class:

```
❶ class PlatformSprite(Sprite):
❷     def __init__(self, game, photo_image, x, y, width, height):
❸         Sprite.__init__(self, game)
❹         self.photo_image = photo_image
❺         self.image = game.canvas.create_image(x, y, \
                  image=self.photo_image, anchor='nw')
❻         self.coordinates = Coords(x, y, x + width, y + height)
```

When we define the PlatformSprite class at ❶, we give it a single parameter: the name of the parent class (Sprite). The __init__ function, at ❷, has seven parameters: self, game, photo_image, x, y, width, and height.

At ❸, we call the __init__ function of the parent class, Sprite, using self and game as the parameter values, because other than the self keyword, the Sprite class's __init__ function takes only one parameter: game.

At this point, if we were to create a PlatformSprite object, it would have all the object variables from its parent class (game, endgame, and coordinates), simply because we've called the __init__ function in Sprite.

At ❹, we save the photo_image parameter as an object variable, and at ❺ we use the canvas variable of the game object to draw the image on screen with create_image.

Finally, we create a Coords object with the x and y parameters as the first two arguments. We then add the width and height parameters to these parameters for the second two arguments at ❻.

Even though the coordinates variable is set to None in the Sprite parent class, we have changed it in our PlatformSprite child class to a real Coords object, containing the real location of the platform image on the screen.

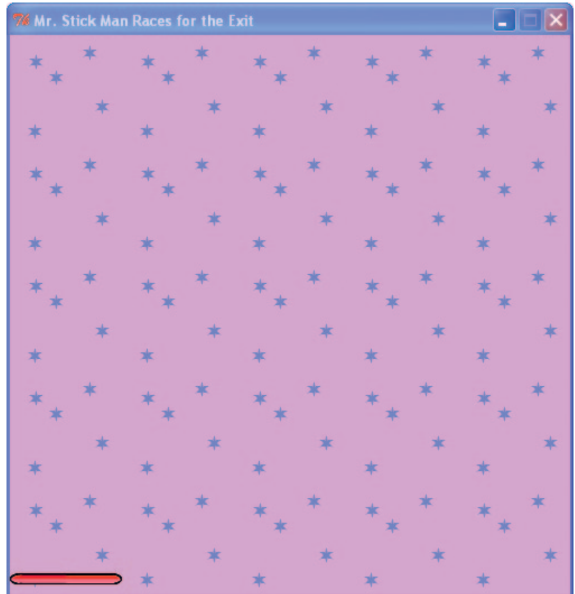
ADDING A PLATFORM OBJECT

Let's add a platform to the game to see how it looks. Change the last two lines of the game file (*stickmangame.py*) as follows:

```
❶ g = Game()
❷ platform1 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
      0, 480, 100, 10)
❸ g.sprites.append(platform1)
❹ g.mainloop()
```

As you can see, lines ❶ and ❹ have not changed, but at ❷, we create an object of the `PlatformSprite` class, passing it the variable for our game (g), along with a `PhotoImage` object (which uses the first of our platform images, *platform1.gif*). We also pass it the position where we want to draw the platform (0 pixels across and 480 pixels down, near the bottom of the canvas), along with the height and width of our image (100 pixels across and 10 pixels high). We add this sprite to the list of sprites in our `game` object at ❸.

If you run the game now, you should see a platform drawn at the bottom-left side of the screen, like this:



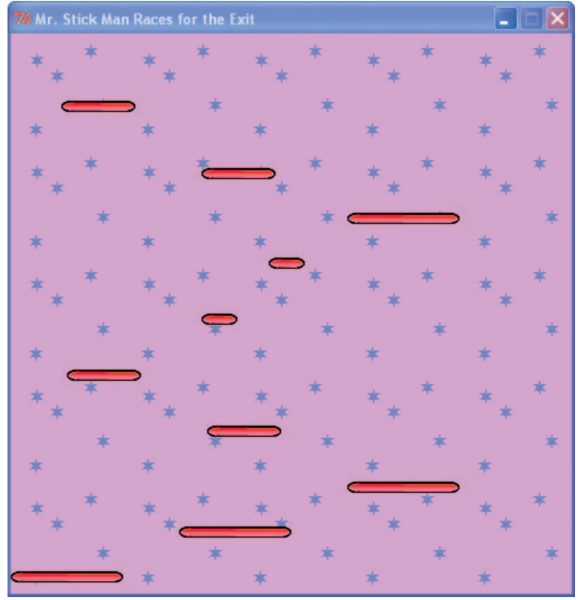
ADDING A BUNCH OF PLATFORMS

Let's add a whole bunch of platforms. Each platform will have different x and y positions, so that they will be drawn scattered around the screen. Here's the code to use:

```
g = Game()
platform1 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
    0, 480, 100, 10)
platform2 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
    150, 440, 100, 10)
platform3 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
    300, 400, 100, 10)
platform4 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
    300, 160, 100, 10)
```

```
platform5 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
    175, 350, 66, 10)
platform6 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
    50, 300, 66, 10)
platform7 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
    170, 120, 66, 10)
platform8 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
    45, 60, 66, 10)
platform9 = PlatformSprite(g, PhotoImage(file="platform3.gif"), \
    170, 250, 32, 10)
platform10 = PlatformSprite(g, PhotoImage(file="platform3.gif"), \
    230, 200, 32, 10)
g.sprites.append(platform1)
g.sprites.append(platform2)
g.sprites.append(platform3)
g.sprites.append(platform4)
g.sprites.append(platform5)
g.sprites.append(platform6)
g.sprites.append(platform7)
g.sprites.append(platform8)
g.sprites.append(platform9)
g.sprites.append(platform10)
g.mainloop()
```

We create a lot of `PlatformSprite` objects, saving them as variables `platform1`, `platform2`, `platform3`, and so on, up to `platform10`. We then add each platform to the variable `sprites`, which we created in our `Game` class. If you run the game now, it should look like this:



We've created the basics of our game! Now we're ready to add our main character, Mr. Stick Man.

WHAT YOU LEARNED

In this chapter, you created the `Game` class and drew the background image onto the screen like a kind of wallpaper. You learned how to determine whether a horizontal or vertical position is within the bounds of two other horizontal or vertical positions by creating the functions `within_x` and `within_y`. You then used these functions to create new functions to determine whether one coordinate object had collided with another. We'll use these functions in the next chapters when we animate Mr. Stick Man and need to detect whether he has collided with a platform as he moves around the canvas.

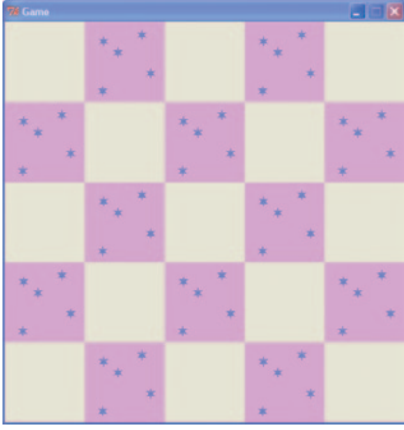
We also created a parent class `Sprite` and its first child class, `PlatformSprite`, which we used to draw the platforms onto the canvas.

PROGRAMMING PUZZLES

The following coding puzzles are some ways that you can experiment with the game's background image. Check your answers at *http://python-for-kids.com/*.

#1: CHECKERBOARD

Try changing the `Game` class so that the background image is drawn like a checkerboard:



#2: Two-Image Checkerboard

Once you've figured out how to create a checkerboard effect, try using two alternating images. Come up with another wallpaper image (using your graphics program), and then change the `Game` class so it displays a checkerboard with two alternating images instead of one image and the blank background.

#3: Bookshelf and Lamp

You can create different wallpaper images to make the background of the game look more interesting. Create a copy of the background image, and then draw a simple bookshelf on it. Or you could draw a table with a lamp or a window. Then dot these images around the screen by changing the `Game` class so that it loads (and displays) three or four different wallpaper images.

17
CREATING MR. STICK MAN

In this chapter, we'll create the main character of our Mr. Stick Man Races for the Exit game. This will require the most complicated coding we've done so far, because Mr. Stick Man needs to run left and right, jump, stop when he runs into a platform, and fall when he runs off the edge of a platform. We'll use event bindings for the left and right arrow keys to make the stick figure run left and right, and we'll have him jump when the player presses the spacebar.

Initializing the Stick Figure

The __init__ function for our new stick figure class will look a lot like it does in the other classes in our game so far. We start by giving our new class a name: StickFigureSprite. As with previous classes, this class has a parent class: Sprite.

```
class StickFigureSprite(Sprite):
    def __init__(self, game):
        Sprite.__init__(self, game)
```

This code looks like what we wrote for the PlatformSprite class in Chapter 16, except that we're not using any additional parameters (other than self and game). The reason is that, unlike with the PlatformSprite class, there will be only one StickFigureSprite object used in the game.

Loading the Stick Figure Images

Because we have a lot of platform objects on the screen, which each can use a different-sized image, we pass the platform image as a parameter of the PlatformSprite's __init__ function (kind of like saying, "Here, Platform Sprite, use this image when you draw yourself on the screen."). But since there's only one stick figure on the screen, it doesn't make sense to load the image outside the sprite and then pass it in as a parameter. The StickFigureSprite class will know how to load its own images.

The next few lines of the __init__ function do this very job: They load each of the three left images (which we'll use to animate the stick figure running left) and the three right images (used to animate the stick figure running right). We need to load these images now, because we don't want to have to load them every time we display the stick figure on the screen (doing so would take too long and make our game run slowly).

```
class StickFigureSprite(Sprite):
    def __init__(self, game):
        Sprite.__init__(self, game)
❶       self.images_left = [
            PhotoImage(file="figure-L1.gif"),
            PhotoImage(file="figure-L2.gif"),
            PhotoImage(file="figure-L3.gif")
        ]
❷       self.images_right = [
            PhotoImage(file="figure-R1.gif"),
            PhotoImage(file="figure-R2.gif"),
            PhotoImage(file="figure-R3.gif")
        ]
❸       self.image = game.canvas.create_image(200, 470, \
                image=self.images_left[0], anchor='nw')
```

This code loads each of the three left images, which we'll use to animate the stick figure running left, and the three right images, which we'll use to animate the stick figure running right.

At ❶ and ❷, we create the object variables `images_left` and `images_right`. Each contains a list of `PhotoImage` objects that we created in Chapter 15, showing the stick figure facing left and right.

We draw the first image at ❸ with `images_left[0]` using the canvas's `create_image` function at position (200, 470), which puts the stick figure in the middle of the game screen, at the bottom of the canvas. The `create_image` function returns a number that identifies the image on the canvas. We store this identifier in the object variable `image` for later use.

SETTING UP VARIABLES

The next part of the `__init__` function sets up some more variables that we'll be using later in this code.

```
        self.images_right = [
            PhotoImage(file="figure-R1.gif"),
            PhotoImage(file="figure-R2.gif"),
            PhotoImage(file="figure-R3.gif")
        ]
        self.image = game.canvas.create_image(200, 470, \
            image=self.images_left[0], anchor='nw')
❶       self.x = -2
❷       self.y = 0
❸       self.current_image = 0
```

```
❹         self.current_image_add = 1
❺         self.jump_count = 0
❻         self.last_time = time.time()
❼         self.coordinates = Coords()
```

At ❶ and ❷, the object variables `x` and `y` will store the amount we'll be adding to the stick figure's horizontal (`x1` and `x2`) or vertical (`y1` and `y2`) coordinates when he is moving around the screen.

As you learned in Chapter 13, in order to animate something with the `tkinter` module, we add values to the object's x or y position to move it around the canvas. By setting `x` to –2, and `y` to 0, we subtract 2 from the x position later in the code and add nothing to the vertical position, to make the stick figure run to the left.

NOTE *Remember that a negative x number means move left on the canvas, and a positive x number means move right. A negative y number means move up, and a positive y number means move down.*

At ❸, we create the object variable `current_image` to store the image's index position as currently displayed on the screen. Our list of left-facing images, `images_left`, contains *figure-L1.gif*, *figure-L2.gif*, and *figure-L3.gif*. Those are index positions 0, 1, and 2.

At ❹, the variable `current_image_add` will contain the number we'll add to that index position stored in `current_image` to get the next index position. For example, if the image at index position 0 is displayed, we add 1 to get the next image at index position 1, and then add 1 again to get the final image in the list at index position 2. (You'll see how we use this variable for animation in the next chapter.)

The variable `jump_count` at ❺ is a counter we'll use while the stick figure is jumping. The variable `last_time` will record the last time we changed the image when animating our stick figure. We store the current time using the `time` function of the `time` module at ❻.

At ❼, we set the `coordinates` object variable to an object of the `Coords` class, with no initialization parameters set (`x1`, `y1`, `x2`, and `y2` are all 0). Unlike with the platforms, the stick figure's coordinates will change, so we'll set these values later.

BINDING TO KEYS

In the final part of the __init__ function, the bind functions bind a key to something in our code that needs to be run when the key is pressed.

```
        self.jump_count = 0
        self.last_time = time.time()
        self.coordinates = Coords()
        game.canvas.bind_all('<KeyPress-Left>', self.turn_left)
        game.canvas.bind_all('<KeyPress-Right>', self.turn_right)
        game.canvas.bind_all('<space>', self.jump)
```

We bind <KeyPress-Left> to the function turn_left, <KeyPress-Right> to the function turn_right, and <space> to the function jump. Now we need to create those functions to make the stick figure move.

TURNING THE STICK FIGURE LEFT AND RIGHT

The turn_left and turn_right functions make sure that the stick figure is not jumping, and then set the value of the object variable x to move him left and right. (If our character is jumping, our game doesn't allow us to change his direction in midair.)

```
        game.canvas.bind_all('<KeyPress-Left>', self.turn_left)
        game.canvas.bind_all('<KeyPress-Right>', self.turn_right)
        game.canvas.bind_all('<space>', self.jump)

❶   def turn_left(self, evt):
❷       if self.y == 0:
❸           self.x = -2

❹   def turn_right(self, evt):
❺       if self.y == 0:
❻           self.x = 2
```

Python calls the turn_left function when the player presses the left arrow key, and it passes an object with information about what the player did as a parameter. This object is called an *event object*, and we give it the parameter name evt.

NOTE *The event object isn't important for our purposes, but we need to include it as a parameter of our functions (at ❶ and ❹) or we'll get an error because Python is expecting it to be there. The event object contains things like the x and y positions of the mouse (mouse event), a code identifying a particular key (keyboard event), and other information. For this game, none of that information is useful, so we can safely ignore it.*

To see if the stick figure is jumping, we check the value of the y object variable at ❷ and ❺. If the value is not 0, the stick figure is jumping. In this example, if the value of y is 0, we set x to –2 to run left (❸) or we set it to 2 to run right (❻), because setting the value to –1 or 1 wouldn't make the stick figure move across the screen fast enough. (Once you have the animation working for your stick figure, try changing this value to see what difference it makes.)

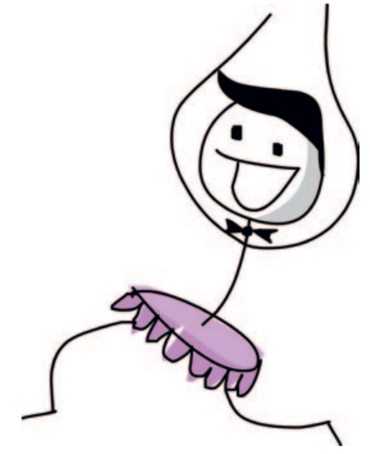
MAKING THE STICK FIGURE JUMP

The `jump` function is very similar to the `turn_left` and `turn_right` functions.

```
    def turn_right(self, evt):
        if self.y == 0:
            self.x = 2

    def jump(self, evt):
❶       if self.y == 0:
❷           self.y = -4
❸           self.jump_count = 0
```

This function takes a parameter evt (the event object), which we can ignore because we don't need any more information about the event. If this function is called, we know it's because the spacebar was pressed.

Because we want our stick figure to jump only if he is not already jumping, at ❶ we check to see if y is equal to 0. If the stick figure is not jumping, at ❷ we set y to –4 (to move him



vertically up the screen), and we set jump_count to 0 at ❸. We'll use jump_count to make sure the stick figure doesn't just keep jumping forever. Instead, we'll let him jump for a specific count and then have him come down again, as if gravity were pulling him. We'll add this code in the next chapter.

WHAT WE HAVE SO FAR

Let's review the definitions of the classes and functions we now have in our game, and where they should be in your file.

At the top of your program, you should have your import statements, followed by the Game and Coords classes. The Game class will be used to create an object which will be the main controller for our game, and objects of the Coords class are used to hold the positions of things in our game (like the platforms and Mr. Stick Man):

```
from tkinter import *
import random
import time

class Game:
    ...
class Coords:
    ...
```

Next, you should have the within functions (which tell whether the coordinates of one sprite are "within" the same area of another sprite), the Sprite parent class (which is the parent class of all the sprites in our game), the PlatformSprite class, and the beginning of the StickFigureSprite class. PlatformSprite was used to create platform objects, which our stick figure will jump across, and we created one object of the StickFigureSprite class, to represent the main character in our game:

```
def within_x(co1, co2):
    ...
def within_y(co1, co2):
    ...
def collided_left(co1, co2):
    ...
def collided_right(co1, co2):
    ...
def collided_top(co1, co2):
    ...
```

```
def collided_bottom(y, co1, co2):
    ...
class Sprite:
    ...
class PlatformSprite(Sprite):
    ...
class StickFigureSprite(Sprite):
    ...
```

Finally, at the end of your program, you should have code that creates all the objects in our game so far: the game object itself and the platforms. The final line is where we call the mainloop function.

```
g = Game()
platform1 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
    0, 480, 100, 10)
...
g.sprites.append(platform1)
...
g.mainloop()
```

If your code looks a bit different, or you're having trouble getting it working, you can always skip ahead to the end of Chapter 18, where you'll find the full listing for the entire game.

What You Learned

In this chapter, we began working on the class for our stick figure. At the moment, if we created an object of this class, it wouldn't really do much besides loading the images it needs for animating the stick figure, and setting up a few object variables to be used later in the code. This class contains a couple of functions for changing the values in those object variables based on keyboard events (when the player presses the left or right arrow, or the spacebar).

In the next chapter, we'll finish our game. We'll write the functions for the StickFigureSprite class to display and animate the stick figure, and move him around the screen. We'll also add the exit (the door) that Mr. Stick Man is trying to reach.

18 COMPLETING THE MR. STICK MAN GAME

In the previous three chapters, we've been developing our game: Mr. Stick Man Races for the Exit. We created the graphics, and then wrote code to add the background image, platforms, and stick figure. In this chapter, we'll fill in the missing pieces to animate the stick figure and add the door.

You'll find the full listing for the complete game at the end of this chapter. If you get lost or become confused when writing some of this code, compare your code with that listing to see where you might have gone wrong.

Animating the Stick Figure

So far, we've created a basic class for our stick figure, loading the images we'll be using and binding keys to some functions. But none of our coding will do anything particularly interesting if you run our game at this point.

Now we'll add the remaining functions to the `StickFigureSprite` class we created in Chapter 17: `animate`, `move`, and `coords`. The `animate` function will draw the different stick figure images, `move` will determine where the character needs to move to, and `coords` will return the stick figure's current position. (Unlike with the platform sprites, we need to recalculate the position of the stick figure as he moves around the screen.)

Creating the Animate Function

First, we'll add the `animate` function, which will need to check for movement and change the image accordingly.

Checking for Movement

We don't want to change the stick figure image too quickly in our animation or its movement won't look realistic. Think about a flip animation, drawn in the corner of a notepad—if you flip the pages too quickly, you may not get the full effect of what you've drawn.

The first half of the `animate` function checks to see if the stick figure is running left or right, and then uses the `last_time` variable to decide whether to change the current image. This variable will help us control the speed of our animation. The function will go after the `jump` function, which we added to our `StickFigureSprite` class in Chapter 17.

```
    def jump(self, evt):
        if self.y == 0:
            self.y = -4
            self.jump_count = 0

    def animate(self):
❶       if self.x != 0 and self.y == 0:
❷           if time.time() - self.last_time > 0.1:
```

```
❸                 self.last_time = time.time()
❹                 self.current_image += self.current_image_add
❺                 if self.current_image >= 2:
❻                     self.current_image_add = -1
❼                 if self.current_image <= 0:
❽                     self.current_image_add = 1
```

In the if statement at ❶, we check to see if x is not 0 in order to determine whether the stick figure is moving (either left or right), and we check to see if y is 0 in order to determine that the stick figure is not jumping. If this if statement is true, we need to animate our stick figure; if not, he's standing still, so there's no need to keep drawing. If the stick figure isn't moving, we drop out of the function, and the rest of the code in this listing is ignored.

At ❷, we calculate the amount of time since the animate function was last called, by subtracting the value of the last_time variable from the current time, using time.time(). This calculation is used to decide whether to draw the next image in the sequence, and if the result is greater than a tenth of a second (0.1), we continue with the block of code at ❸. We set the last_time variable to the current time, basically resetting the stopwatch to start timing again for the next change of image.

At ❹, we add the value of the object variable current_image_add to the variable current_image, which stores the index position of the currently displayed image. Remember that we created the current_image_add variable in the stick figure's __init__ function in Chapter 17, so when the animate function is first called, the value of the variable has already been set to 1.

At ❺, we check to see if the value of the index position in current_image is greater than or equal to 2, and if so, we change the value of current_image_add to –1 at ❻. The process is similar at ❼—once we reach 0, we need to start counting up again, which we do at ❽.

NOTE *If you're having trouble figuring out how to indent this code, here's a hint: There are 8 spaces at the beginning of ❶ and 20 spaces at the beginning of ❽.*

To help you understand what's going on in the function so far, imagine that you have a sequence of colored blocks in a line on the floor. You move your finger from one block to the next, and each block that your finger points to (1, 2, 3, 4, and so on) has a number

(the current_image variable). The number of the block your finger moves to (it points at one block at a time) is the number stored in the variable current_image_add. When your finger moves one way up the line of blocks, you're adding 1 each time, and when it hits the end of the line and moves back down, you're subtracting 1 (that is, adding –1).

The code we've added to our animate function performs this process, but instead of colored blocks, we have the three stick figure images for each direction stored in a list. The index positions of these images are 0, 1, and 2. As we animate the stick figure, once we reach the last image, we start counting down, and once we reach the first image, we need to start counting up again. As a result, we create the effect of a running figure.

The following shows how we move through the list of images, using the index positions we calculate in the animate function.

Position 0	Position 1	Position 2	Position 1	Position 0	Position 1
Counting up	Counting up	Counting up	Counting down	Counting down	Counting up

CHANGING THE IMAGE

In the next half of the animate function, we change the currently displayed image, using the calculated index position.

```
    def animate(self):
        if self.x != 0 and self.y == 0:
            if time.time() - self.last_time > 0.1:
                self.last_time= time.time()
                self.current_image += self.current_image_add
                if self.current_image >= 2:
                    self.current_image_add = -1
                if self.current_image <= 0:
                    self.current_image_add = 1
❶       if self.x < 0:
❷           if self.y != 0:
❸               self.game.canvas.itemconfig(self.image, \
                        image=self.images_left[2])
❹           else:
❺               self.game.canvas.itemconfig(self.image, \
                        image=self.images_left[self.current_image])
```

```
❻        elif self.x > 0:
❼            if self.y != 0:
❽                self.game.canvas.itemconfig(self.image, \
                        image=self.images_right[2])
❾            else:
❿                self.game.canvas.itemconfig(self.image, \
                        image=self.images_right[self.current_image])
```

At ❶, if x is less than 0, the stick figure is moving left, so Python moves into the block of code shown at ❷ through ❺, which checks whether y is not equal to 0 (meaning the stick figure is jumping). If y is not equal to 0 (the stick figure is moving up or down—in other words, jumping), we use the canvas's itemconfig function to change the displayed image to the last image in our list of left-facing images at ❸ (images_left[2]). Because the stick figure is jumping, we'll use the image showing him in full stride to make the animation look a bit more realistic:

If the stick figure is not jumping (that is, y is equal to 0), the else statement starting at ❹ uses itemconfig to change the displayed image to whatever index position is in the variable current_image, as shown in the code at ❺.

At ❻, we see if the stick figure is running right (x is greater than 0), and Python moves into the block shown at ❼ through ❿. This code is very similar to the first block, again checking whether the stick figure is jumping, and drawing the correct image if so, except that it uses the images_right list.

GETTING THE STICK FIGURE'S POSITION

Because we'll need to determine where the stick figure is on the screen (since he is moving around), the coords function will differ from the other Sprite class functions. We'll use the coords function of the canvas to determine where the stick figure is, and then use those values to set the x1, y1 and x2, y2 values of the coordinates

variable we created in the `__init__` function at the beginning of Chapter 17. Here's the code, which can be added after the `animate` function:

```
        if self.x < 0:
            if self.y != 0:
                self.game.canvas.itemconfig(self.image, \
                        image=self.images_left[2])
            else:
                self.game.canvas.itemconfig(self.image, \
                        image=self.images_left[self.current_image])
        elif self.x > 0:
            if self.y != 0:
                self.game.canvas.itemconfig(self.image, \
                        image=self.images_right[2])
            else:
                self.game.canvas.itemconfig(self.image, \
                        image=self.images_right[self.current_image])

    def coords(self):
❶       xy = self.game.canvas.coords(self.image)
❷       self.coordinates.x1 = xy[0]
❸       self.coordinates.y1 = xy[1]
❹       self.coordinates.x2 = xy[0] + 27
❺       self.coordinates.y2 = xy[1] + 30
        return self.coordinates
```

When we created the `Game` class in Chapter 16, one of the object variables was the `canvas`. At ❶, we use the `coords` function of this `canvas` variable, with `self.game.canvas.coords`, to return the x and y positions of the current image. This function uses the number stored in the object variable `image`, the identifier for the image drawn on the canvas.

We store the resulting list in the variable `xy`, which now contains two values: the top-left x position stored as the `x1` variable of `coordinates` at ❷, and the top-left y position stored as the `y1` variable of `coordinates` at ❸.

Because all of the stick figure images we created are 27 pixels wide by 30 pixels high, we can determine what the `x2` and `y2` variables should be by adding the width at ❹ and the height at ❺ to the x and y numbers, respectively.

Finally, on the last line of the function, we return the object variable `coordinates`.

MAKING THE STICK FIGURE MOVE

The final function of the StickFigureSprite class, move, is in charge of actually moving our game character around the screen. It also needs to be able to tell us when the character has bumped into something.

STARTING THE MOVE FUNCTION

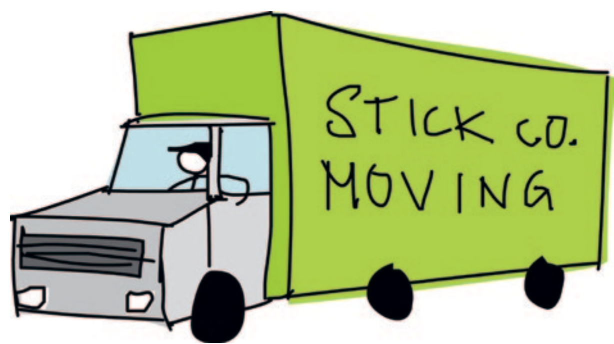
Here's the code for the first part of the move function—this will go after coords:

```
    def coords(self):
        xy = self.game.canvas.coords(self.image)
        self.coordinates.x1 = xy[0]
        self.coordinates.y1 = xy[1]
        self.coordinates.x2 = xy[0] + 27
        self.coordinates.y2 = xy[1] + 30
        return self.coordinates

    def move(self):
❶       self.animate()
❷       if self.y < 0:
❸           self.jump_count += 1
❹           if self.jump_count > 20:
❺               self.y = 4
❻       if self.y > 0:
❼           self.jump_count -= 1
```

At ❶, this part of the function calls the animate function we created earlier in this chapter, which changes the currently displayed image if necessary. At ❷, we see whether the value of y is less than 0. If it is, we know that the stick figure is jumping because a negative value will move him up the screen. (Remember that 0 is at the top of the canvas, and the bottom of the canvas is pixel position 500.)

At ❸, we add 1 to jump_count, and at ❹, we say that if the value of jump_count reaches 20, we should change y to 4 to start the stick figure falling again (❺).

At ❻, we see if the value of y is greater than 0 (meaning the character must be falling), and if it is, we subtract 1 from jump_count because once we've counted up to 20, we need to count back down again. (Move your hand slowly up in the air while counting to 20,



then move it back down again while counting down from 20, and you'll get a sense of how calculating the stick figure jumping up and down is supposed to work.)

In the next few lines of the move function, we call the coords function, which tells us where our character is on the screen and stores its result in the variable co. We then create the variables left, right, top, bottom, and falling. We'll use each in the remainder of this function.

```
        if self.y > 0:
            self.jump_count -= 1
        co = self.coords()
        left = True
        right = True
        top = True
        bottom = True
        falling = True
```

Notice that each variable has been set to the Boolean value True. We'll use these as indicators to check whether the character has hit something on the screen or is falling.

HAS THE STICK FIGURE HIT THE BOTTOM OR TOP OF THE CANVAS?

The next section of the move function checks whether our character has hit the bottom or top of the canvas. Here's the code:

```
        bottom = True
        falling = True
❶       if self.y > 0 and co.y2 >= self.game.canvas_height:
❷           self.y = 0
❸           bottom = False
```

```
❹        elif self.y < 0 and co.y1 <= 0:
❺            self.y = 0
❻            top = False
```

If the character is falling down the screen, y will be greater than 0, so we need to make sure it hasn't yet hit the bottom of the canvas (or it will vanish off the bottom of the screen). To do so, at ❶, we see if its y2 position (the bottom of the stick figure) is greater than or equal to the canvas_height variable of the game object. If it is, we set the value of y to 0 at ❷ to stop the stick figure from falling, and then set the bottom variable to False at ❸, which tells the remaining code that we no longer need to see if the stick figure has hit the bottom.

The process of determining whether the stick figure has hit the top of the screen is very similar to the way we determine whether he has hit the bottom. To do so, at ❹, we first see if the stick figure is jumping (y is less than 0), then we see if his y1 position is less than or equal to 0, meaning he has hit the top of the canvas. If both conditions are true, we set y equal to 0 at ❺ to stop the movement. We also set the top variable to False at ❻ to tell the remaining code that we no longer need to see if the stick figure has hit the top.

Has the Stick Figure Hit the Side of the Canvas?

We follow almost exactly the same process as in the preceding code to determine whether the stick figure has hit the left and right sides of the canvas, as follows:

```
        elif self.y < 0 and co.y1 <= 0:
            self.y = 0
            top = False
❶        if self.x > 0 and co.x2 >= self.game.canvas_width:
❷            self.x = 0
❸            right = False
❹        elif self.x < 0 and co.x1 <= 0:
❺            self.x = 0
❻            left = False
```

The code at ❶ is based on the fact that we know the stick figure is running to the right if x is greater than 0. We also know whether he has hit the right-hand side of the screen by seeing if the x2 position (co.x2) is greater than or equal to the width of the canvas stored in game_width. If both statements are true, we set x equal to 0 (to stop the stick figure from running), and we set the right variable to False at ❸.

COLLIDING WITH OTHER SPRITES

Once we've determined whether the figure has hit the sides of the screen, we need to see if he has hit anything else on the screen. We use the following code to loop through the list of sprite objects stored in the game object to see if the stick figure has hit any of them.

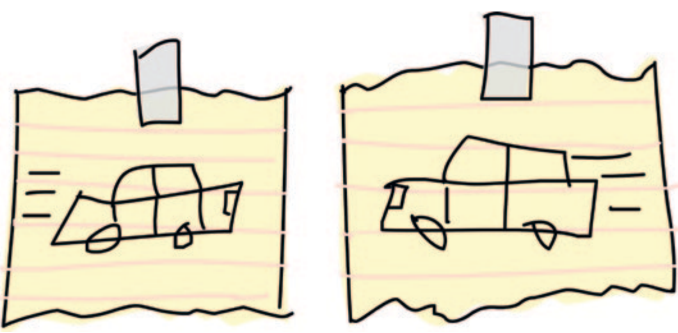
```
        elif self.x < 0 and co.x1 <= 0:
            self.x = 0
            left = False
❶       for sprite in self.game.sprites:
❷           if sprite == self:
❸               continue
❹           sprite_co = sprite.coords()
❺           if top and self.y < 0 and collided_top(co, sprite_co):
❻               self.y = -self.y
❼               top = False
```

At ❶, we loop through the list of sprites, assigning each one in turn to the variable sprite. At ❷, we say that if the sprite is equal to self (that's another way of saying, "if this sprite is the same as me"), we don't need to determine whether the stick figure has collided because he would have only hit himself. If the sprite variable is equal to self, we use continue to jump to the next sprite in the list.

Next, we get the coordinates of the new sprite by calling its coords function at ❹ and storing the results in the variable sprite_co.

Then the code at ❺ checks for the following:

- The stick figure has not hit the top of the canvas (the top variable is still true).
- The stick figure is jumping (the value of y is less than 0).
- The top of the stick figure has collided with the sprite from the list (using the collided_top function we created in Chapter 16).



If all of these conditions are true, we want the sprite to start falling back down again, so at ❻, we reverse the value of the y using minus (-). The top variable is set to False at ❼, because once the stick figure has hit the top, we don't need to keep checking for a collision.

Colliding at the Bottom

The next part of the loop checks to see if the bottom of our character has hit something:

```
            if top and self.y < 0 and collided_top(co, sprite_co):
                self.y = -self.y
                top = False
❶           if bottom and self.y > 0 and collided_bottom(self.y, \
                    co, sprite_co):
❷               self.y = sprite_co.y1 - co.y2
❸               if self.y < 0:
❹                   self.y = 0
❺               bottom = False
❻               top = False
```

There are three similar checks at ❶: whether the bottom variable is still set, whether the character is falling (y is greater than 0), and whether the bottom of our character has hit the sprite. If all three checks are true, we subtract the bottom y value (y2) of the stick figure from the top y value of the sprite (y1) at ❷. This might seem strange, so let's see why we do this.

Imagine that our game character has fallen off a platform. He moves down the screen 4 pixels each time the mainloop function runs, and the foot of the stick figure is 3 pixels above another platform. Let's say the stick figure's bottom (y2) is at position 57 and the top of the platform (y1) is at position 60. In this case, the collided_bottom function would return true, because its code will add the value of y (which is 4) to the stick figure's y2 variable, resulting in 61.

However, we don't want Mr. Stick Man to stop falling as soon as it looks like he will hit a platform or the bottom of the screen, because that would be like taking a huge jump off a step and stopping in midair, an inch above the ground. That may be a neat trick, but it won't look right in our game. Instead, if we subtract the character's y2 value (of 57) from the platform's y1 value (of 60) we get 3, the amount the stick figure should drop in order to land properly on the top of the platform.

At ❸, we make sure that the calculation doesn't result in a negative number; if it does, we set `y` equal to 0 at ❹. (If we let the number be negative, the stick figure would fly back up again, and we don't want that to happen in this game.)

Finally, we set the `top` ❻ and `bottom` ❺ flags to `False`, so we no longer need to check whether the stick figure has collided at the top or bottom with another sprite.

We'll do one more bottom check to see whether the stick figure has run off the edge of a platform. Here's the code for this `if` statement:

```
            if self.y < 0:
                self.y = 0
            bottom = False
            top = False
        if bottom and falling and self.y == 0 \
                and co.y2 < self.game.canvas_height \
                and collided_bottom(1, co, sprite_co):
            falling = False
```

Five checks here must all be true in order for the `falling` variable to be set to `False`:

- We still need to check that the `bottom` flag is set to `True`.
- We need to check whether the stick figure should be falling (the `falling` flag is still set to `True`).
- The stick figure isn't already falling (`y` is 0).
- The bottom of the sprite hasn't hit the bottom of the screen (it's less than the canvas height).
- The stick figure has hit the top of a platform (`collided_bottom` returns `True`).

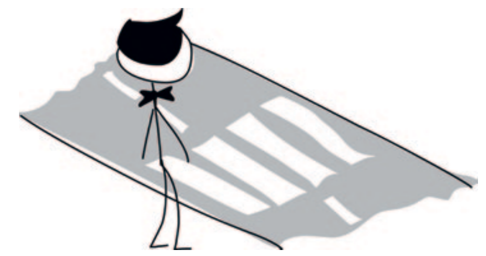
Then we set the `falling` variable to `False`.

CHECKING LEFT AND RIGHT

We've checked whether the stick figure has hit a sprite at the bottom or the top. Now we need to check whether he has hit the left or right side, with this code:

```
        if bottom and falling and self.y == 0 \
                and co.y2 < self.game.canvas_height \
                and collided_bottom(1, co, sprite_co):
            falling = False
```

```
❶             if left and self.x < 0 and collided_left(co, sprite_co):
❷                 self.x = 0
❸                 left = False
❹             if right and self.x > 0 and collided_right(co, sprite_co):
❺                 self.x = 0
❻                 right = False
```



At ❶, we see if we should still be looking for collisions to the left (`left` is still set to `True`) and whether the stick figure is moving to the left (`x` is less than 0). We also check to see if the stick figure has collided with a sprite using the `collided_left` function. If these three conditions are true, we set `x` equal to 0 at ❷ (to make the stick figure stop running), and set `left` to `False` at ❸, so that we no longer check for collisions on the left.

The code is similar for collisions to the right, as shown at ❹. We set `x` equal to 0 again at ❺, and `right` to `False` at ❻, to stop checking for right-hand collisions.

Now, with checks for collisions in all four directions, our `for` loop should look like this:

```
        elif self.x < 0 and co.x1 <= 0:
            self.x = 0
            left = False
        for sprite in self.game.sprites:
            if sprite == self:
                continue
            sprite_co = sprite.coords()
            if top and self.y < 0 and collided_top(co, sprite_co):
                self.y = -self.y
                top = False
            if bottom and self.y > 0 and collided_bottom(self.y, \
                    co, sprite_co):
                self.y = sprite_co.y1 - co.y2
                if self.y < 0:
                    self.y = 0
                bottom = False
                top = False
            if bottom and falling and self.y == 0 \
                    and co.y2 < self.game.canvas_height \
                    and collided_bottom(1, co, sprite_co):
                falling = False
```

```
            if left and self.x < 0 and collided_left(co, sprite_co):
                self.x = 0
                left = False
            if right and self.x > 0 and collided_right(co, sprite_co):
                self.x = 0
                right = False
```

We need to add only a few more lines to the move function, as follows:

```
            if right and self.x > 0 and collided_right(co, sprite_co):
                self.x = 0
                right = False
❶       if falling and bottom and self.y == 0 \
                and co.y2 < self.game.canvas_height:
❷           self.y = 4
❸       self.game.canvas.move(self.image, self.x, self.y)
```

At ❶, we check whether both the falling and bottom variables are set to True. If so, we've looped through every platform sprite in the list without colliding at the bottom.

The final check in this line determines whether the bottom of our character is less than the canvas height—that is, above the ground (the bottom of the canvas). If the stick figure hasn't collided with anything and he is above the ground, he is standing in mid-air, so he should start falling (in other words, he has run off the end of a platform). To make him run off the end of any platform, we set y equal to 4 at ❷.

At ❸, we move the image across the screen, according to the values we set in the variables x and y. The fact that we've looped through the sprites checking for collisions may mean that we've set both variables to 0, because the stick figure has collided on the left and with the bottom. In that case, the call to the move function of the canvas will do nothing.

It may also be the case that Mr. Stick Man has walked off the edge of a platform. If that happens, y will be set to 4, and Mr. Stick Man will fall downward.

Phew, that was a long function!

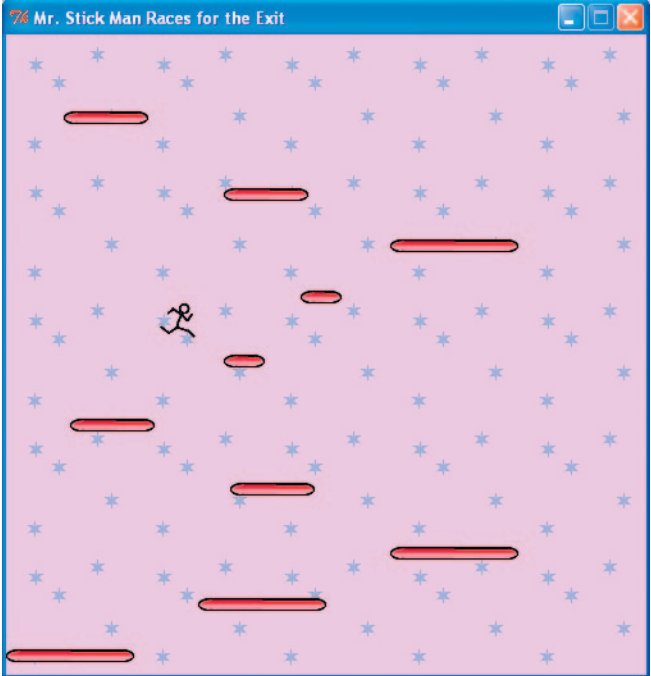
TESTING OUR STICK FIGURE SPRITE

Having created the StickFigureSprite class, let's try it out by adding the following two lines just before the call to the mainloop function.

```
❶ sf = StickFigureSprite(g)
❷ g.sprites.append(sf)
  g.mainloop()
```

At ❶, we create a StickFigureSprite object and set it equal to the variable sf. As we did with the platforms, we add this new variable to the list of sprites stored in the game object at ❷.

Now run the program. You will find that Mr. Stick Man can run, jump from platform to platform, and fall!



THE DOOR!

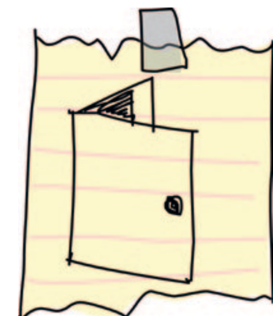
The only thing missing from our game is the door to the exit. We'll finish up by creating a sprite for the door, adding code to detect the door, and giving our program a door object.

CREATING THE DOORSPRITE CLASS

You guessed it—we need to create one more class: DoorSprite. Here's the start of the code:

```
class DoorSprite(Sprite):
❶    def __init__(self, game, photo_image, x, y, width, height):
❷        Sprite.__init__(self, game)
❸        self.photo_image = photo_image
❹        self.image = game.canvas.create_image(x, y, \
                image=self.photo_image, anchor='nw')
❺        self.coordinates = Coords(x, y, x + (width / 2), y + height)
❻        self.endgame = True
```

As shown at ❶, the __init__ function of the DoorSprite class has parameters for self, a game object, a photo_image object, the x and y coordinates, and the width and height of the image. At ❷, we call __init__ as with our other sprite classes.



At ❸, we save the parameter photo_image using an object variable with the same name, as we did with PlatformSprite. We create a display image using the canvas create_image function and save the identifying number returned by that function using the object variable image at ❹.

At ❺, we set the coordinates of DoorSprite to the x and y parameters (which become the x1 and y1 positions of the door), and then calculate the x2 and y2 positions. We calculate the x2 position by adding half of the width (the width variable, divided by 2) to the x parameter. For example, if x is 10 (the x1 coordinate is also 10), and the width is 40, the x2 coordinate would be 30 (10 plus half of 40).

Why use this confusing little calculation? Because, unlike with the platforms, where we want Mr. Stick Man to stop running as soon as he collides with the side of the platform, we want him to stop in front of the door. (It won't look good if Mr. Stick Man stops running next to the door!) You'll see this in action when you play the game and make it to the door.

Unlike the x1 position, the y1 position is simple to calculate. We just add the value of the height variable to the y parameter, and that's it.

Finally, at ❻, we set the endgame object variable to True. This says that when the stick figure reaches the door, the game should end.

Detecting the Door

Now we need to change the code in the StickFigureSprite class of the move function that determines when the stick figure has collided with a sprite on the left or the right. Here's the first change:

```
        if left and self.x < 0 and collided_left(co, sprite_co):
            self.x = 0
            left = False
            if sprite.endgame:
                self.game.running = False
```

We check to see if the sprite that the stick figure has collided with has an endgame variable that is set to True. If it does, we set the running variable to False, and everything stops—we've reached the end of the game.

We'll add these same lines to the code that checks for a collision on the right. Here's the code:

```
        if right and self.x > 0 and collided_right(co, sprite_co):
            self.x = 0
            right = False
            if sprite.endgame:
                self.game.running = False
```

Adding the Door Object

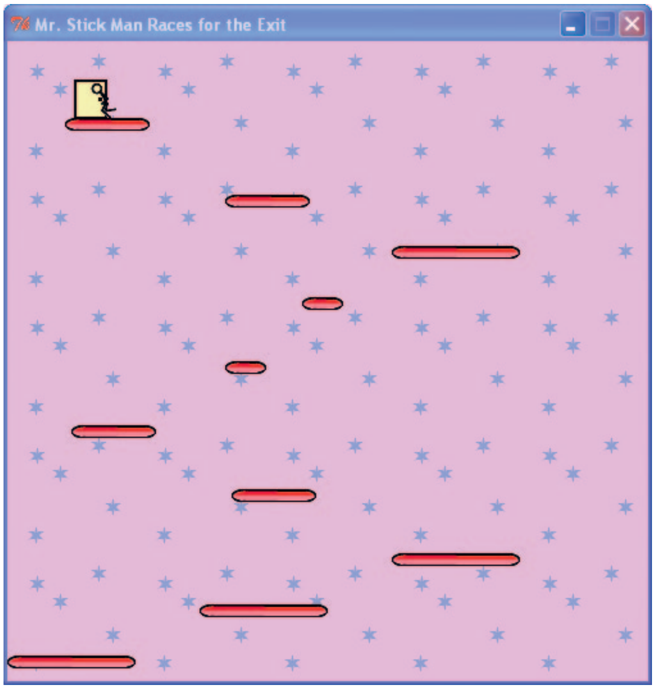
Our final addition to the game code is an object for the door. We'll add this before the main loop. Just before creating the stick figure object, we'll create a door object, and then add it to the list of sprites. Here's the code:

```
g.sprites.append(platform7)
g.sprites.append(platform8)
g.sprites.append(platform9)
g.sprites.append(platform10)
door = DoorSprite(g, PhotoImage(file="door1.gif"), 45, 30, 40, 35)
g.sprites.append(door)
sf = StickFigureSprite(g)
g.sprites.append(sf)
g.mainloop()
```

We create a door object using the variable for our game object, g, followed by a PhotoImage (the door image we created in Chapter 15). We set the x and y parameters to 45 and 30 to put the door on a

platform near the top of the screen, and set the width and height to 40 and 35. We add the door object to the list of sprites, as with all the other sprites in the game.

You can see the result when Mr. Stick Man reaches the door. He stops running in front of the door, rather than next to it, as shown here:



THE FINAL GAME

The full listing of our game is now a bit more than 200 lines of code. The following is the complete code for the game. If you have trouble getting your game to work, compare each function (and each class) to this listing and see where you've gone wrong.

```
from tkinter import *
import random
import time

class Game:
    def __init__(self):
        self.tk = Tk()
        self.tk.title("Mr. Stick Man Races for the Exit")
        self.tk.resizable(0, 0)
```

```
        self.tk.wm_attributes("-topmost", 1)
        self.canvas = Canvas(self.tk, width=500, height=500, \
                highlightthickness=0)
        self.canvas.pack()
        self.tk.update()
        self.canvas_height = 500
        self.canvas_width = 500
        self.bg = PhotoImage(file="background.gif")
        w = self.bg.width()
        h = self.bg.height()
        for x in range(0, 5):
            for y in range(0, 5):
                self.canvas.create_image(x * w, y * h, \
                        image=self.bg, anchor='nw')
        self.sprites = []
        self.running = True

    def mainloop(self):
        while 1:
            if self.running == True:
                for sprite in self.sprites:
                    sprite.move()
            self.tk.update_idletasks()
            self.tk.update()
            time.sleep(0.01)

class Coords:
    def __init__(self, x1=0, y1=0, x2=0, y2=0):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

def within_x(co1, co2):
    if (co1.x1 > co2.x1 and co1.x1 < co2.x2) \
            or (co1.x2 > co2.x1 and co1.x2 < co2.x2) \
            or (co2.x1 > co1.x1 and co2.x1 < co1.x2) \
            or (co2.x2 > co1.x1 and co2.x2 < co1.x2):
        return True
    else:
        return False

def within_y(co1, co2):
    if (co1.y1 > co2.y1 and co1.y1 < co2.y2) \
            or (co1.y2 > co2.y1 and co1.y2 < co2.y2) \
            or (co2.y1 > co1.y1 and co2.y1 < co1.y2) \
            or (co2.y2 > co1.y1 and co2.y2 < co1.y2):
        return True
```

```
    else:
        return False

def collided_left(co1, co2):
    if within_y(co1, co2):
        if co1.x1 <= co2.x2 and co1.x1 >= co2.x1:
            return True
    return False

def collided_right(co1, co2):
    if within_y(co1, co2):
        if co1.x2 >= co2.x1 and co1.x2 <= co2.x2:
            return True
    return False

def collided_top(co1, co2):
    if within_x(co1, co2):
        if co1.y1 <= co2.y2 and co1.y1 >= co2.y1:
            return True
    return False

def collided_bottom(y, co1, co2):
    if within_x(co1, co2):
        y_calc = co1.y2 + y
        if y_calc >= co2.y1 and y_calc <= co2.y2:
            return True
    return False

class Sprite:
    def __init__(self, game):
        self.game = game
        self.endgame = False
        self.coordinates = None
    def move(self):
        pass
    def coords(self):
        return self.coordinates

class PlatformSprite(Sprite):
    def __init__(self, game, photo_image, x, y, width, height):
        Sprite.__init__(self, game)
        self.photo_image = photo_image
        self.image = game.canvas.create_image(x, y, \
                image=self.photo_image, anchor='nw')
        self.coordinates = Coords(x, y, x + width, y + height)
```

```
class StickFigureSprite(Sprite):
    def __init__(self, game):
        Sprite.__init__(self, game)
        self.images_left = [
            PhotoImage(file="figure-L1.gif"),
            PhotoImage(file="figure-L2.gif"),
            PhotoImage(file="figure-L3.gif")
        ]
        self.images_right = [
            PhotoImage(file="figure-R1.gif"),
            PhotoImage(file="figure-R2.gif"),
            PhotoImage(file="figure-R3.gif")
        ]
        self.image = game.canvas.create_image(200, 470, \
                image=self.images_left[0], anchor='nw')
        self.x = -2
        self.y = 0
        self.current_image = 0
        self.current_image_add = 1
        self.jump_count = 0
        self.last_time = time.time()
        self.coordinates = Coords()
        game.canvas.bind_all('<KeyPress-Left>', self.turn_left)
        game.canvas.bind_all('<KeyPress-Right>', self.turn_right)
        game.canvas.bind_all('<space>', self.jump)

    def turn_left(self, evt):
        if self.y == 0:
            self.x = -2

    def turn_right(self, evt):
        if self.y == 0:
            self.x = 2

    def jump(self, evt):
        if self.y == 0:
            self.y = -4
            self.jump_count = 0

    def animate(self):
        if self.x != 0 and self.y == 0:
            if time.time() - self.last_time > 0.1:
                self.last_time= time.time()
                self.current_image += self.current_image_add
```

```
                if self.current_image >= 2:
                    self.current_image_add = -1
                if self.current_image <= 0:
                    self.current_image_add = 1
        if self.x < 0:
            if self.y != 0:
                self.game.canvas.itemconfig(self.image, \
                        image=self.images_left[2])
            else:
                self.game.canvas.itemconfig(self.image, \
                        image=self.images_left[self.current_image])
        elif self.x > 0:
            if self.y != 0:
                self.game.canvas.itemconfig(self.image, \
                        image=self.images_right[2])
            else:
                self.game.canvas.itemconfig(self.image, \
                        image=self.images_right[self.current_image])

    def coords(self):
        xy = self.game.canvas.coords(self.image)
        self.coordinates.x1 = xy[0]
        self.coordinates.y1 = xy[1]
        self.coordinates.x2 = xy[0] + 27
        self.coordinates.y2 = xy[1] + 30
        return self.coordinates

    def move(self):
        self.animate()
        if self.y < 0:
            self.jump_count += 1
            if self.jump_count > 20:
                self.y = 4
        if self.y > 0:
            self.jump_count -= 1
        co = self.coords()
        left = True
        right = True
        top = True
        bottom = True
        falling = True
        if self.y > 0 and co.y2 >= self.game.canvas_height:
            self.y = 0
            bottom = False
        elif self.y < 0 and co.y1 <= 0:
            self.y = 0
            top = False
```

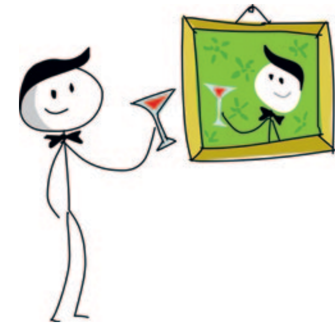
```
            if self.x > 0 and co.x2 >= self.game.canvas_width:
                self.x = 0
                right = False
            elif self.x < 0 and co.x1 <= 0:
                self.x = 0
                left = False
            for sprite in self.game.sprites:
                if sprite == self:
                    continue
                sprite_co = sprite.coords()
                if top and self.y < 0 and collided_top(co, sprite_co):
                    self.y = -self.y
                    top = False
                if bottom and self.y > 0 and collided_bottom(self.y, \
                        co, sprite_co):
                    self.y = sprite_co.y1 - co.y2
                    if self.y < 0:
                        self.y = 0
                    bottom = False
                    top = False
                if bottom and falling and self.y == 0 \
                        and co.y2 < self.game.canvas_height \
                        and collided_bottom(1, co, sprite_co):
                    falling = False
                if left and self.x < 0 and collided_left(co, sprite_co):
                    self.x = 0
                    left = False
                    if sprite.endgame:
                        self.game.running = False
                if right and self.x > 0 and collided_right(co, sprite_co):
                    self.x = 0
                    right = False
                    if sprite.endgame:
                        self.game.running = False
            if falling and bottom and self.y == 0 \
                    and co.y2 < self.game.canvas_height:
                self.y = 4
            self.game.canvas.move(self.image, self.x, self.y)

    class DoorSprite(Sprite):
        def __init__(self, game, photo_image, x, y, width, height):
            Sprite.__init__(self, game)
            self.photo_image = photo_image
            self.image = game.canvas.create_image(x, y, \
                    image=self.photo_image, anchor='nw')
            self.coordinates = Coords(x, y, x + (width / 2), y + height)
            self.endgame = True
```

```
g = Game()
platform1 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
    0, 480, 100, 10)
platform2 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
    150, 440, 100, 10)
platform3 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
    300, 400, 100, 10)
platform4 = PlatformSprite(g, PhotoImage(file="platform1.gif"), \
    300, 160, 100, 10)
platform5 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
    175, 350, 66, 10)
platform6 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
    50, 300, 66, 10)
platform7 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
    170, 120, 66, 10)
platform8 = PlatformSprite(g, PhotoImage(file="platform2.gif"), \
    45, 60, 66, 10)
platform9 = PlatformSprite(g, PhotoImage(file="platform3.gif"), \
    170, 250, 32, 10)
platform10 = PlatformSprite(g, PhotoImage(file="platform3.gif"), \
    230, 200, 32, 10)
g.sprites.append(platform1)
g.sprites.append(platform2)
g.sprites.append(platform3)
g.sprites.append(platform4)
g.sprites.append(platform5)
g.sprites.append(platform6)
g.sprites.append(platform7)
g.sprites.append(platform8)
g.sprites.append(platform9)
g.sprites.append(platform10)
door = DoorSprite(g, PhotoImage(file="door1.gif"), 45, 30, 40, 35)
g.sprites.append(door)
sf = StickFigureSprite(g)
g.sprites.append(sf)
g.mainloop()
```

WHAT YOU LEARNED

In this chapter, we completed our game, Mr. Stick Man Races for the Exit. We created a class for our animated stick figure and wrote functions to move him around the screen and animate him as he moves (changing from one image to the next to give the illusion of running). We've used basic collision detection to tell when he has hit the left or right sides of the canvas, and when he has hit

another sprite, such as a platform or a door. We've also added collision code to tell when he hits the top of the screen or the bottom, and to make sure that when he runs off the edge of a platform, he tumbles down accordingly. We added code to tell when Mr. Stick Man has reached the door, so the game comes to an end.



PROGRAMMING PUZZLES

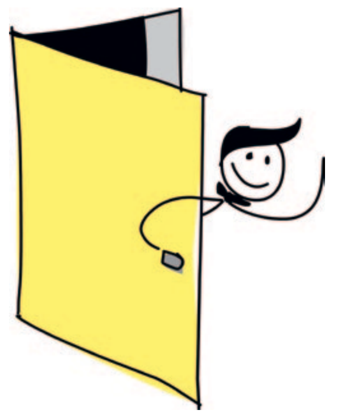
There's a lot more we can do to improve the game. At the moment, it's very simple, so we can add code to make it more professional looking and more interesting to play. Try adding the following features and then check your code at *http://python-for-kids.com/*.

#1: "YOU WIN!"

Like the "Game Over" text in the Bounce! game we completed in Chapter 14, add the "You Win!" text when the stick figure reaches the door, so players can see that they have won.

#2: ANIMATING THE DOOR

In Chapter 15, we created two images for the door: one open and one closed. When Mr. Stick Man reaches the door, the door image should change to the open door, Mr. Stick Man should vanish, and the door image should revert to the closed door. This will give the illusion that Mr. Stick Man is exiting and closing the door as he leaves. You can do this by changing the `DoorSprite` class and the `StickFigureSprite` class.



#3: MOVING PLATFORMS

Try adding a new class called `MovingPlatformSprite`. This platform should move from side to side, making it more difficult for Mr. Stick Man to reach the door at the top.

GLOSSARY

Sometimes, when you're programming for the first time, you'll encounter a new term that just doesn't make much sense. That lack of understanding can get in the way of making any real progress. But there's a simple solution to that problem!

I've created this glossary to help you through those times when a new word or term holds you up. You'll find definitions of many of the programming terms used in this book, so look here if you encounter a word that you don't understand.

animation The process of displaying a sequence of images fast enough that it looks like something is moving.

block A group of computer statements in a program.

Boolean A type of value that can be either true or false. (In Python, it's `True` or `False`, with capital `T` and `F`.)

call Run the code in a function. When we use a function, we say we are "calling" it.

canvas An area of the screen for drawing on. `canvas` is a class provided by the `tkinter` module.

child When we're talking about classes, we describe the relationships between classes as that of parents and children. A child class inherits the characteristics of its parent class.

class A description or definition of a type of thing. In programming terms, a class is a collection of functions and variables.

click Press one of the mouse buttons to push an on-screen button, select a menu option, and so on.

collision In computer games, when one character in the game crashes into another character or object on the screen.

condition An expression in a program that is a bit like a question. Conditions evaluate to true or false.

coordinates The position of a pixel on the screen. This is usually described as a number of pixels across the screen (x) and a number of pixels down (y).

degrees A unit of measurement for angles.

data Usually refers to information stored and manipulated by a computer.

dialog A dialog is typically a small window in an application that presents some contextual information, such as an alert or an error message, or asks you to respond to a question. For example, when you choose to open a file, the window that appears is usually the File dialog.

dimensions In the context of graphics programming, *two-dimensional* or *three-dimensional* refers to how images are displayed on a computer monitor. Two-dimensional (2D) graphics are flat images on a screen that have width and height—like

the old cartoons you might see on TV. Three-dimensional (3D) graphics are images on the screen that have width, height, and the appearance of depth—the sort of graphics you might see in a more realistic computer game.

directory The location of a group of files on the hard disk of your computer.

embed Replace values inside a string. The replaced values are sometimes called *placeholders*.

error When something goes wrong with a program on your computer, this is an error. When programming with Python, you might see all sorts of messages displayed in response to an error. If you enter your code incorrectly you might see an `IndentationError`, for example.

event Something that occurs when a program is running. For example, an event might be someone moving the mouse, clicking the mouse button, or typing on a keyboard.

exception A type of error that can occur when running a program.

execute Run some code, like a program, a small snippet of code, or a function.

frame One of a series of images that makes up an animation.

function A command in a programming language that is usually a collection of statements that perform some action.

hexadecimal A way of representing numbers, particularly in computer programming. Hexadecimal numbers are base 16, which means the numbers go from 0 through 9 and then A, B, C, D, E, and F.

horizontal The left and right directions on the screen (represented by x).

identifier A number that uniquely names something in a program. For example, in Python's `tkinter` module, the identifier is used to refer to shapes drawn on the canvas.

image A picture on the computer screen.

import In Python terms, importing makes a module available for your program to use.

initialize Refers to setting up the initial state of an object (that is, setting variables in the object when it is first created).

installation The process of copying a software application's files onto your computer so that the application is available for use.

instance The instance of a class—in other words, an object.

keyword A special word used by a programming language. Keywords are also referred to as *reserved words*, which basically means you can't use them for anything else (for example, you can't use a keyword as the name of a variable).

loop A repeated command or set of commands.

memory A device or component in your computer that is used to temporarily store information.

module A group of functions and variables.

null The absence of value (in Python, also referred to as `None`).

object The specific instance of a class. When you create an object of a class, Python sets aside some of your computer's memory to store information about a member of that class.

operator An element in a computer program used for mathematics or for comparing values.

parameter A value used with a function when calling it or when creating an object (when calling the Python `__init__` function, for example). Parameters are sometimes referred to as *arguments*.

parent When referring to classes and objects, the parent of a class is another class that functions and variables are inherited from. In other words, a child class inherits the characteristics of its parent class. When we're not talking Python, a parent is the person who tells you to brush your teeth before going to bed at night.

pixel A single point on your computer screen—the smallest dot that the computer is capable of drawing.

program A set of commands that tells a computer what to do.

scope The part, or section, of a program where a variable can be "seen" (or used). (A variable inside a function may not be visible to code outside the function.)

shell In computing, a shell is a command-line interface of some kind. In this book, "the Python shell" refers to the IDLE application.

software A collection of computer programs.

sprite A character or an object in a computer game.

string A collection of alphanumeric characters (letters, numbers, punctuation, and whitespace).

syntax The arrangement and order of words in a program.

transparency In graphics programming, part of an image that isn't displayed, meaning it doesn't overwrite whatever is displayed behind it.

variable Something used to store values. A variable is like a label for information held in the computer's memory. Variables aren't permanently tied to a specific value, hence the name "variable," meaning it can change.

vertical The up and down directions on the screen (represented by y).