



Computer Programming for Kids and Other Beginners

Warren and Carter Sande



 MANNING

Chapter 16. Graphics.....	1
Getting some help—Pygame.....	1
Pygame and IDLE.....	0
A Pygame window.....	2
A better ending.....	0
Drawing in the window.....	5
What’s the “flip”?.....	0
How to make a circle.....	0
Pygame surfaces.....	0
Colors in Pygame.....	0
Locations—screen coordinates.....	9
Size of shapes.....	10
Line width.....	0
Modern art?.....	0
Individual pixels.....	13
Connect the dots.....	0
Connect the dots, again.....	0
Drawing point-by-point.....	0
Images.....	17
Let’s get moving!.....	19
Animation.....	20
Erasing images.....	0
What’s under there?.....	0
Smoother animation.....	21
Keeping the ball moving.....	0
Bouncing the ball.....	23
Bouncing in 2-D.....	0
Wrapping the ball.....	25
What did you learn?.....	0
Test your knowledge.....	26
Try it out.....	27

CHAPTER 16

Graphics

You have been learning about a lot of the basic elements of computer programming: input and output, variables, decisions, loops, lists, functions, objects, and modules. I hope you have enjoyed filling up your brain with all this stuff! Now it's time to start having a bit more fun with programming and Python.

In this chapter, you'll learn how to draw things on the screen, like lines, shapes, colors, and even a bit of animation. This will help us make some games and other programs in the next few chapters.

Getting some help—Pygame

Getting graphics (and sound) to work on your computer can be a little complicated. It involves the

operating system, your graphics card, and a lot of low-level code that we don't really want to worry about for now. So we're going to use a Python module called Pygame to help make things a bit simpler.



Pygame lets you create graphics and the other things you need to make games work on different computers and operating systems, without having to know all the messy details of each system. Pygame is free, and a version of Pygame comes with this book. It should be installed if you used the book's installer to install Python. If not, you'll have to install it separately. You can get it from the Pygame web site, www.pygame.org.

Pygame also needs some help from another module called Numeric. Numeric is also installed by the book's installer, and if you don't have it, you can get it at the Pygame web site.

Pygame and IDLE

Remember when we used EasyGui to make our first GUI programs, and I mentioned that some people have trouble using EasyGui with IDLE? Well, the same goes for Pygame and IDLE. On my system, I can't run some Pygame programs properly from IDLE. For the rest of this chapter, and for any other programs in the rest of the book that use Pygame, I recommend you use SPE instead of IDLE, just like we did with EasyGui back in chapter 6.

The only thing you might have to do differently is use the **Run in Terminal** option (or **Run in Terminal without arguments**), instead of the normal **Run** option. Play around with it, experiment, and I'm sure you'll figure it out. That's a big part of what programming is about—figuring things out for yourself!

A Pygame window

The first thing we need to do is make a window where we'll start drawing our graphics. Listing 16.1 shows a very simple program that just makes a Pygame window.

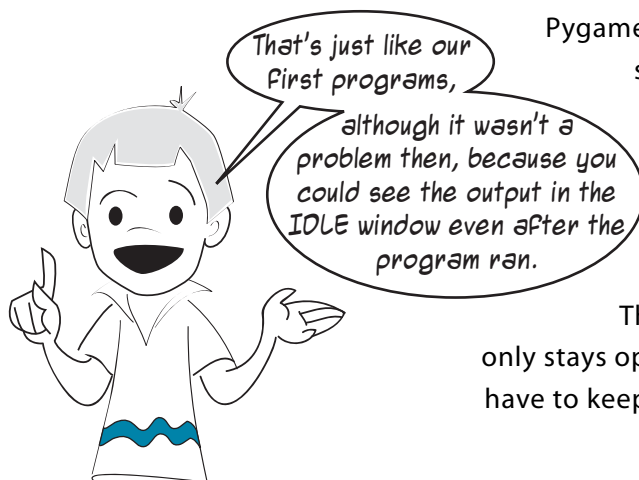
Listing 16.1 Making a Pygame window

```
import pygame
pygame.init()
screen = pygame.display.set_mode([640, 480])
```

Try running this program. What did you see? If you were looking closely, you might have seen a window (filled with black) pop on the screen very briefly. What's up with that?

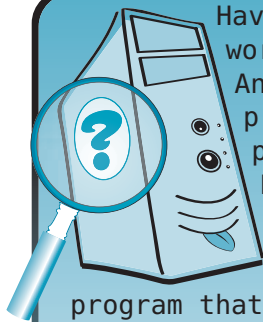
Well, Pygame is meant for making games. Games don't just do things on their own—they have to interact with the player. So Pygame has something called an *event loop* that constantly checks for the user doing something, like pressing keys or moving the mouse.

Pygame programs need to keep the event loop running all the time, and as soon as the event loop stops, the program stops. In our first Pygame program, we didn't start the event loop, so the program stopped very soon after it started.



That's right. But in Pygame, the window only stays open while the program is running. So we have to keep it running.

WHAT'S GOING ON IN THERE?



Have you been wondering why Pygame sometimes doesn't work with IDLE? It has to do with the *event loop*.

An *event loop* is a loop that runs constantly in a program, checking for *events* like a key being pressed or the mouse being clicked or moved.

Pygame programs need an event loop.

IDLE also has its own event loop, because it's a program too, and it happens to be a graphical program that needs to keep checking for user input. The two event loops don't always get along—they sometimes bump into each other and cause havoc.

The same is true for IDLE and EasyGui. It's like if someone is on the phone and you pick up an extension and try to make another call. You can't, because the phone is already busy. If you start talking or dialing, that will interfere with the conversation that's already going on.

SPE doesn't have this problem because it has a way to keep its own event loop separate from the event loop of the program that it's running (like your game).

One way to keep the Pygame event loop running is with a **while** loop, like the one in listing 16.2 (but *don't try it yet!*).

Listing 16.2 Keeping the Pygame window open

```
import pygame
pygame.init()
screen = pygame.display.set_mode([640, 480])
while True:
    pass
```

pass is a Python keyword that means “do nothing.” It's just a placeholder, because a **while** loop needs a block of code, and the block can't be empty. (Perhaps you remember that from chapter 8 when we talked about loops.) So we put something in the **while** block, but that “something” does nothing.

Remember that a **while** loop runs as long as the condition is **True**. So this really says, “While **True** is **True**, keep looping.” Because **True** is always **True**, that means forever (or as long as the program runs).

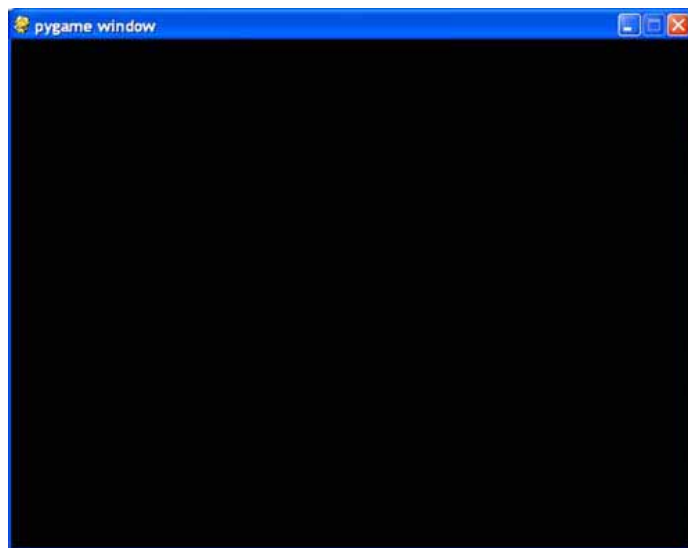
But if it'll keep going forever, how will we stop it? Do you recall that, back in chapter 8, Carter asked about stopping a program that had a runaway loop? We learned that you can

use **Ctrl-C** to do that. We can use the same method here. However, *when running programs in SPE on Windows, you need to use **Ctrl-Break** instead of **Ctrl-C***. There's only one trick to this: you need to *make the command shell the active window* before you type **Ctrl-Break**. If you try using **Ctrl-Break** in the Pygame window, nothing will happen.



If you have a runaway loop on a Mac, you should be able to press **Ctrl-C** to stop it. If that doesn't work, you can try **Ctrl-** to send it a *quit* signal. Or you can start up the Activity Monitor (located in the Utilities folder in the Applications folder), find the Python or Pygame process, and quit it. If you are using Linux, the easiest way is to kill the process.

Okay, now that you know how to stop it, try running the program in listing 16.2. You can type it into whatever editor you're using, and save it as **pygame_1.py**. When you run it, you should see a new window pop up, with a black background. It should have **pygame** in the title bar. The window will stay there until you make the command shell the active window and end the program with **Ctrl-Break**.



If you're running Pygame from SPE, there will be a shell window opened for you. That window will have something like **SPE <filename> - Press Ctrl + Break to stop** in the title bar. Click in that window to make it active before trying to quit the application.

A better ending

There is a better way to stop our Pygame program. You probably noticed that the Pygame window has an "X" icon in the top-right corner in the title bar (as most windows do in Windows). You'd expect the "X" would close the window; it works in every other program. But this is our program. We're in control, and we haven't told the "X" what to do yet. We're going to make the "X" close our Pygame program.

In a Pygame program, the “X” should be connected to a built-in function called `sys.exit()`. This is a function in Python’s standard `sys` module that tells the program to exit, or stop. We just need to import the `sys` module and make one other change to our code, as shown in listing 16.3.

Listing 16.3 Making the Pygame window closeable

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640, 480])
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
```

Remove the pass and put this code in its place

We’ll learn more about what those last three lines mean soon. For now, we’ll just include them in all our Pygame programs.

Drawing in the window

Now we have a Pygame window that stays open until we close it using the “X” icon. The [640, 480] in the third line of listing 16.3 is the size of our window: 640 pixels wide by 480 pixels high. Let’s start drawing some graphics in there. Change your program so it looks like listing 16.4.

Listing 16.4 Drawing a circle

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640, 480])
screen.fill([255, 255, 255])
pygame.draw.circle(screen, [255, 0, 0], [100, 100], 30, 0)
pygame.display.flip()
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
```

Fills the window with a white background

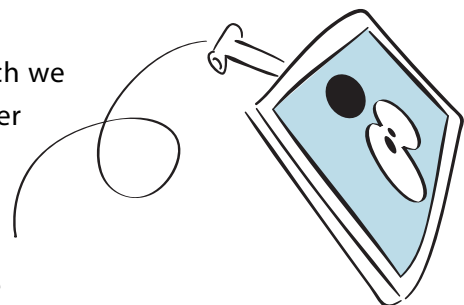
Flips your monitor over... Just kidding!

Add these three lines

Draws a circle

What’s the “flip”?

The display object in Pygame (ours is called `screen`, which we created in line 3 of listing 16.4) has two copies of whatever is displayed in the Pygame window. The reason for this is that, when we start doing animation, we want to make it as smooth and fast as possible. So instead of updating the display every time we make a small change



to our graphics, we can make a number of changes, then “flip” to the new version of the graphics. This makes the changes appear all at once, instead of one by one. This way we don’t get half-drawn circles (or aliens, or whatever) on our display.

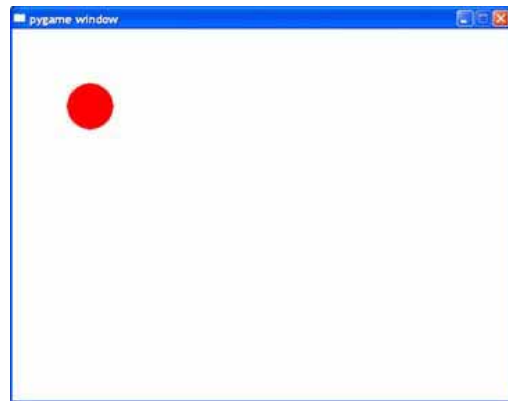
Think of the two copies as being a current screen and a next screen. The current screen is what we see right now. The “next” screen is what we’ll see when we do a “flip.” We make all our changes on the “next” screen and then flip to it so we can see them.

How to make a circle

When you run the program in listing 16.4, you should see a red circle near the upper-left corner of the window, like this:

Not surprisingly, the `pygame.draw.circle()` function draws a circle. You have to tell it five things:

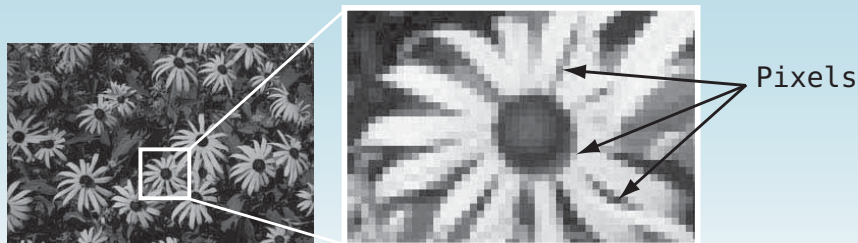
- On what *surface* to draw the circle. (In this case, it’s on the surface we defined in line 3, called `screen`, which is the display surface.)
- What *color* to draw it. (In this case, it’s red, which is represented by the `[255, 0, 0]`).
- At what *location* to draw it. (In this case, it’s at `[100, 100]`, which is 100 pixels down and 100 pixels over from the top-left corner.)
- What *size* to draw it. (In this case, it’s 30, which is the radius, in pixels—the distance from the center of the circle to its outer edge.)
- The *width of the line*. (If `width = 0`, the circle is filled in completely, as it is here.)

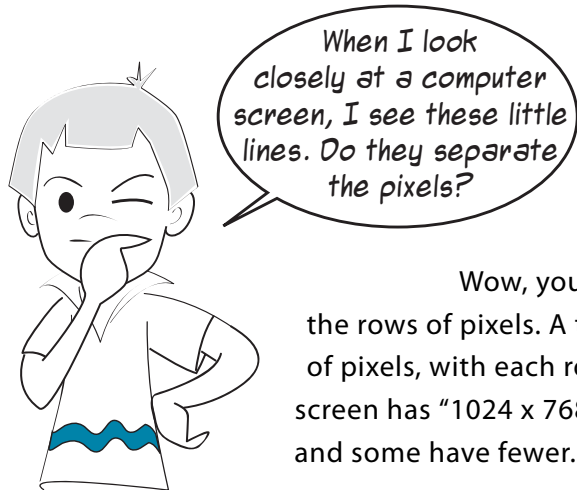


Now we’re going to look at these five things in more detail.

WORD BOX

The word *pixel* is short for “picture element.” This means one dot on your screen or in an image. If you look at any picture with an image viewer and zoom in (make the image really big), you can see the individual pixels. Here’s a regular view of a photo and a zoomed-in version where you can see the pixels.





Wow, you have good eyes! The little lines are actually the rows of pixels. A typical computer screen might have 768 rows of pixels, with each row having 1024 pixels in it. We'd say that screen has "1024 x 768 resolution." Some screens have more pixels, and some have fewer.

Pygame surfaces

If I asked you to draw a picture in real life, one of your first questions would be, "What should I draw it on?" In Pygame, a *surface* is what we draw on. The *display surface* is the one we see on the screen. That's the one we called `screen` in listing 16.4. But a Pygame program can have many surfaces, and you can copy images from one surface to another. You can also do things to surfaces, like rotate them and resize them (make them bigger or smaller).

As I mentioned before, there are two copies of the display surface. In software lingo, we say the display surface is *double-buffered*. This is so we don't get half-completed shapes and images drawn on the screen. We draw our circles, aliens, or whatever in the buffer, and then "flip" the display surface to show us the completely drawn images.

Colors in Pygame

The color system used in Pygame is a common one used in many computer languages and programs. It's called *RGB*. The R, G, and B stand for red, green, and blue.

You might have learned in science class that you can make any color by combining or mixing the three *primary colors* of light: red, green, and blue. That's the same way it works on computers. Each color gets a number from 0 to 255. If all the numbers are 0, there is none of any color, which is completely dark, so you get the color black. If they're all 255, you get the brightest of all three colors mixed together, which is white. If you have something like [255, 0, 0], that would be pure red with no green or blue. Pure green would be [0, 255, 0]. Pure blue would be [0, 0, 255]. If all three numbers are the same, like [150, 150, 150], you get some shade of grey. The lower the numbers, the darker the shade; the higher the numbers, the brighter the shade.

Colors are given as a list of three integers, each one ranging from 0 to 255.

Color names

Pygame has a list of named colors you can use if you don't want to use the [R, G, B] notation. There are over 600 color names defined. I won't list them all here, but if you want to see what they are, search your hard drive for a file called **colordict.py**, and open it in a text editor.

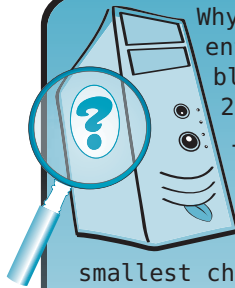
If you want to use the color names, you have to add this line at the start of your program:

```
from pygame.color import THECOLORS
```

Then, when you want to use one of the named colors, you'll do it like this (in our circle example):

```
pygame.draw.circle(screen, THECOLORS["red"], [100, 100], 30, 0)
```

WHAT'S GOING ON IN THERE?



Why 255? The range from 0 to 255 gives us 256 different values for each primary color (red, green, and blue). So, what's special about that number? Why not 200 or 300 or 500?

Two hundred and fifty-six is the number of different values you can make with 8 bits. That's all the possible combinations of eight 1s and 0s. Eight bits is also called a byte, and a byte is the smallest chunk of memory that has its own address. An address is the computer's way of finding particular pieces of memory.

It's like on your street. Your house or apartment has an address, but your room doesn't have its own address. A house is the smallest "addressable unit" on the street. A byte is the smallest "addressable unit" in your computer's memory.

They could have used more than 8 bits for each color, but the next amount that makes sense would be 16 bits (2 bytes), because it's not very convenient to use only part of a byte. And it turns out that, because of the way the human eye sees color, 8 bits is enough to make realistic-looking colors.

Because there are three values (red, green, blue), each with 8 bits, that's 24 bits in total, so this way of representing color is also known as "24-bit color." It uses 24 bits for each pixel, 8 for each primary color.

If you want to play around and experiment with how the red, green, and blue combine to make different colors, you can try out the **colormixer.py** program that was put in the **examples** folder when you ran this book's installer. This will let you try any combination of red, green, and blue to see what color you get.

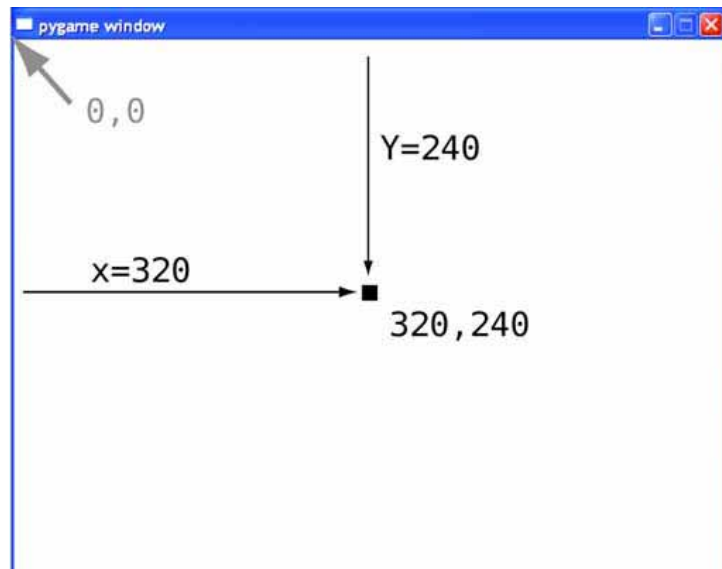
Locations—screen coordinates

If we want to draw or place something on the screen, we need to specify where on the screen it should go. There are two numbers: one for the x-axis (horizontal direction) and one for the y-axis (vertical direction). In Pygame, the numbers start at [0, 0] in the upper-left corner of the window.



When you see a pair of numbers like [320, 240], the first number is horizontal, or the distance from the left side. The second number is vertical, or the distance down from the top. In math and programming, the letter x is often used for horizontal distance, and y is often used for vertical distance.

We made our window 640 pixels wide by 480 pixels high. If we wanted to put the circle in the middle of the window, we'd need to draw it at [320, 240]. That's 320 pixels over from the left-hand edge, and 240 pixels down from the top edge.



Let's try drawing the circle in the middle of the window. Try the program in listing 16.5.

Listing 16.5 Putting the circle in the middle of the window

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
pygame.draw.circle(screen, [255,0,0],[320,240], 30, 0)
pygame.display.flip()
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
```

Change this from [100, 100] to [320, 240]

The location [320, 240] is used as the center of the circle. Compare the results of running listing 16.5 to the results you saw when you ran listing 16.4 to see the difference.

Size of shapes

When you use Pygame's **draw** functions to draw shapes, you have to specify what size to make the shape. For a circle, there is only one size: the radius. For something like a rectangle, you'd have to specify the length and width.

Pygame has a special kind of object called a **rect** (short for "rectangle") that is used for defining rectangular areas. You define a **rect** using the coordinates of its top-left corner and its width and height:

```
Rect(left, top, width, height)
```

This defines both the location and the size. Here's an example:

```
my_rect = Rect(250, 150, 300, 200)
```

This would create a rectangle where the top-left corner is 250 pixels from the left side of the window and 150 pixels down from the top of the window. The rectangle would be 300 pixels wide and 200 pixels high. Let's try it and see.

Substitute this line for line 5 in listing 16.5 and see what it looks like:

```
pygame.draw.rect(screen, [255,0,0], [250, 150, 300, 200], 0)
```

Color of the
rectangle

Location and size
of the rectangle

Line width
(or filled)

The location and size of the rectangle can be a simple list (or tuple) of numbers or a Pygame **Rect** object. So you could also substitute the preceding line with two lines like this:

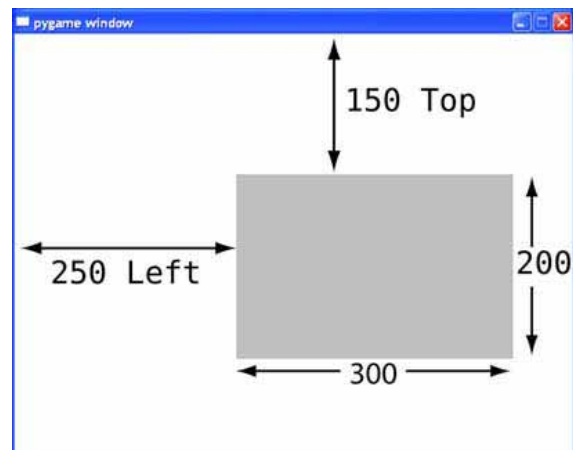
```
my_list = [250, 150, 300, 200]
pygame.draw.rect(screen, [255,0,0], my_list, 0)
```

or

```
my_rect = pygame.Rect(250, 150, 300, 200)
pygame.draw.rect(screen, [255,0,0], my_rect, 0)
```

Here's what the rectangle should look like.

I added some dimensions to show you which numbers mean what:



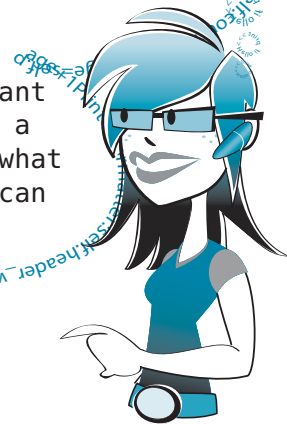
Notice that we only pass four arguments to `pygame.draw.rect`. That's because the `rect` has both location and size in a single argument. In `pygame.draw.circle`, the location and size are two different arguments, so we pass it five arguments.

Thinking like a (Pygame) programmer

Once you create a rectangle with `Rect(left, top, width, height)`, there are several other attributes that you can use to move and align the `Rect`:

- the four edges: `top`, `left`, `bottom`, `right`
- the four corners: `topleft`, `bottomleft`, `topright`, `bottomright`
- the middle of each side: `midtop`, `midleft`, `midbottom`, `midright`
- the center: `center`, `centerx`, `centery`
- dimensions: `size`, `width`, `height`

These are just for convenience. So, if you want to move a rectangle so that its center is at a certain point, you don't have to figure out what the top and left coordinates should be; you can access the center location directly.



Line width

The last thing we need to specify when drawing shapes is how thick to make the line. In the examples so far, we used the line width of 0, which fills in the whole shape. If we used a different line width, we'd see an outline of the shape.

Try changing the line width to 2:

```
pygame.draw.rect(screen, [255,0,0], [250, 150, 300, 200], 2)
```

Make this 2

Try it and see how it looks. Try other line widths too.

Modern art?

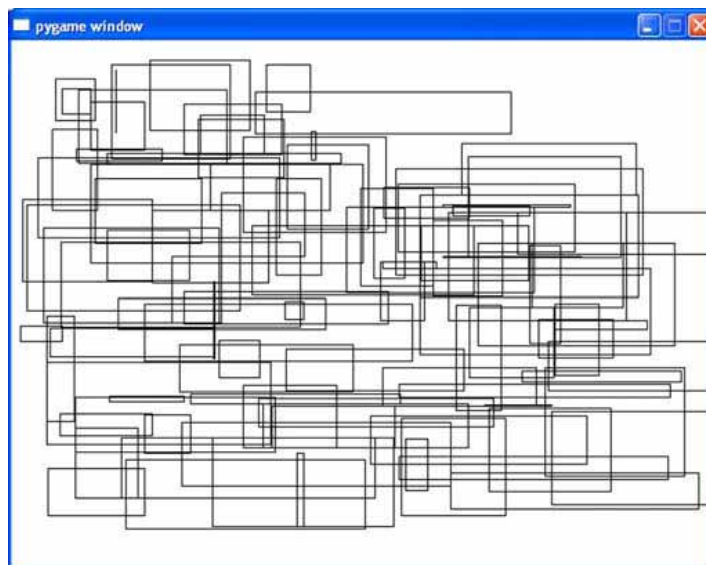
Want to try making some computer-generated modern art? Just for fun, try the code in listing 16.6. You can start with what you had from listing 16.5 and modify it, or just start from scratch.

Listing 16.6 Using `draw.rect` to make art

```
import pygame, sys, random
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
for i in range (100):
    width = random.randint(0, 250)
    height = random.randint(0, 100)
    top = random.randint(0, 400)
    left = random.randint(0, 500)
    pygame.draw.rect(screen, [0,0,0], [left, top, width, height], 1)
pygame.display.flip()

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
```

Run this and see what you get. It should look something like this:



Do you understand how the program works? It draws one hundred rectangles with random sizes and positions. To make it even more “artsy,” add some color and make the line width random too, as in listing 16.7.

Listing 16.7 Modern art with color

```
import pygame, sys, random
from pygame.color import THECOLORS
pygame.init()
screen = pygame.display.set_mode([640,480])
```

```

screen.fill([255, 255, 255])
for i in range(100):
    width = random.randint(0, 250)
    height = random.randint(0, 100)
    top = random.randint(0, 400)
    left = random.randint(0, 500)
    color_name = random.choice(THECOLORS.keys())
    color = THECOLORS[color_name]
    line_width = random.randint(1, 3)
    pygame.draw.rect(screen, color, [left, top, width, height], line_width)
pygame.display.flip()
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

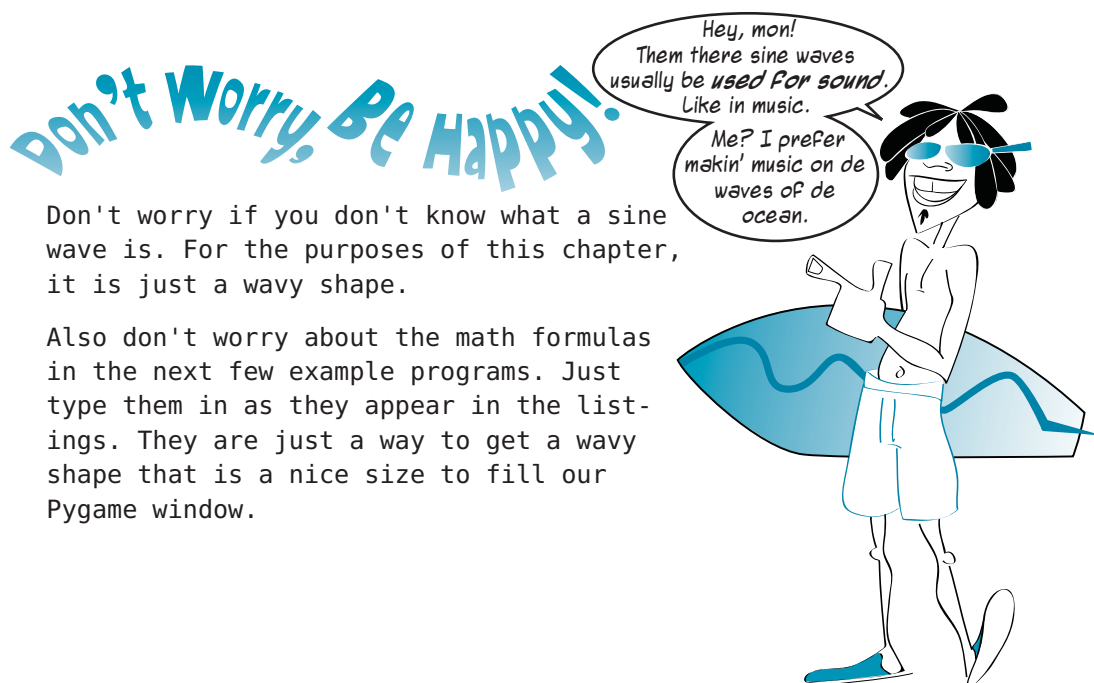
```

Don't worry about
how this line works
for now

When you run this, you'll get something that looks different every time. If you get one that looks really nice, give it a fancy title like "Voice of the Machine" and see if you can sell it to your local art gallery!

Individual pixels

Sometimes we don't want to draw a circle or rectangle, but we want to draw individual dots or pixels. Maybe we're creating a math program and want to draw a sine wave, for example.



Don't worry if you don't know what a sine wave is. For the purposes of this chapter, it is just a wavy shape.

Also don't worry about the math formulas in the next few example programs. Just type them in as they appear in the listings. They are just a way to get a wavy shape that is a nice size to fill our Pygame window.

Because there is no `pygame.draw.sinewave()` method, we have to draw it ourselves from individual points. One way to do this is to draw tiny circles or rectangles, with a size of just one or two pixels. Listing 16.8 shows how that would look using rectangles.

Listing 16.8 Drawing curves using a lot of small rectangles

```

import pygame, sys
import math
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
for x in range(0, 640):
    y = int(math.sin(x/640.0 * 4 * math.pi) * 200 + 240)
    pygame.draw.rect(screen, [0,0,0],[x, y, 1, 1], 1)
pygame.display.flip()
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

```

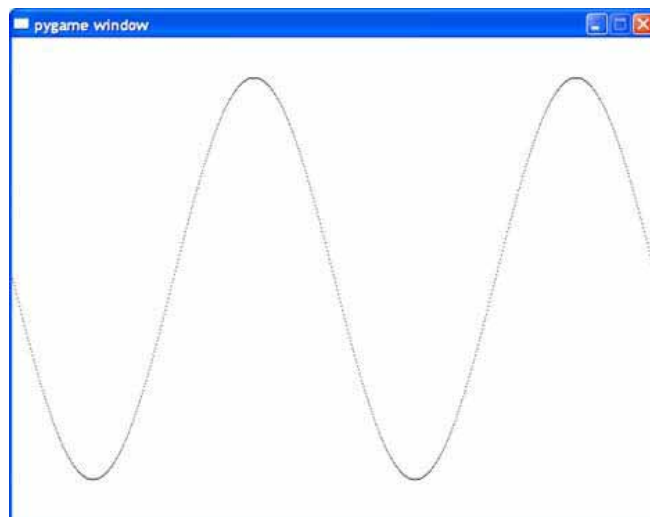
Imports the math functions, including `sin()`

Loops from left to right, `x = 0 to 639`

Calculates the y-position (vertical) of each point

Draws the point using a small rectangle

And here's what it looks like when it runs:



To draw each point, we used a rectangle 1 pixel wide by 1 pixel high. Note that we also used a line width of 1, not 0. If we used a line width of 0, nothing would show up, because there's no "middle" to fill in.

Connect the dots

If you look really closely, you might notice that the sine wave isn't continuous—there are spaces between the points in the middle. That's because, at the steep part of the sine wave, we have to move up (or down) by 3 pixels when we move one pixel to the right. And because we're drawing individual points, not lines, there's nothing to fill the space in between.

Let's try the same thing using a short line to join each plot point. Pygame has a method to draw a single line, but it also has a method that will draw lines between a series of points (like "connect the dots"). That method is `pygame.draw.lines()`, and it needs five parameters:

- the **surface** to draw on.
- a **color**.

- whether the shape will be **closed** by drawing a line joining the last point back to the first one. We don't want to enclose our sine wave, so this will be **False** for us.
- a **list** of points to connect.
- the **width** of the line.

So in our sine wave example, the `pygame.draw.lines()` method would look like this:

```
pygame.draw.lines(screen, [0,0,0],False, plotPoints, 1)
```

In the **for** loop, instead of drawing each point, we'll just create the list of points that `draw.lines()` will connect. Then we have a single call to `draw.lines()`, which is outside the **for** loop. The whole program is shown in listing 16.9.

Listing 16.9 A well-connected sine wave

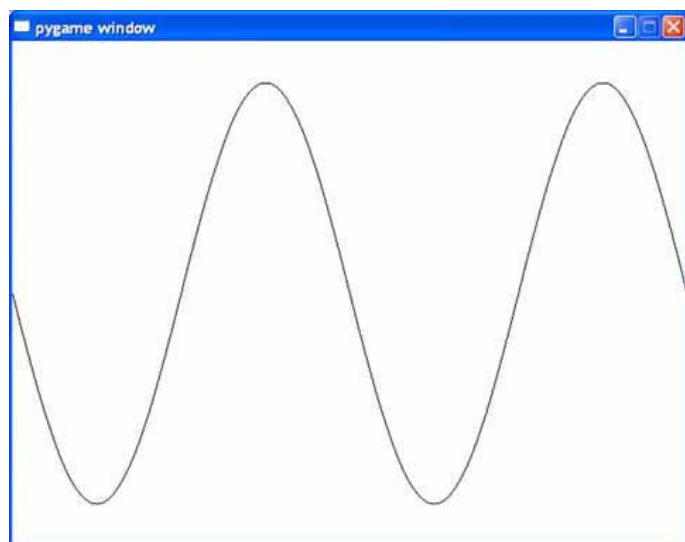
```
import pygame, sys
import math
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
plotPoints = []
for x in range(0, 640):
    y = int(math.sin(x/640.0 * 4 * math.pi) * 200 + 240)
    plotPoints.append([x, y])
pygame.draw.lines(screen, [0,0,0],False, plotPoints, 1)
pygame.display.flip()
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
```

Calculates y-position for each point

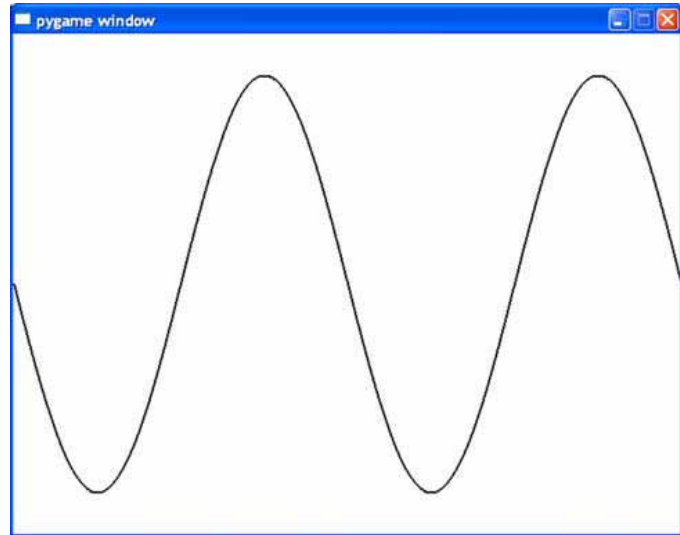
Adds each point to the list

Draws the whole curve with the draw.lines() function

Now when we run it, it looks like this:



That's better—no gaps between the points. If we increase the line width to 2, it looks even better:



Connect the dots, again

Remember those connect-the-dots puzzles you did when you were young? Here's a Pygame version.

The program in listing 16.10 creates a shape using the `draw.lines()` function and a list of points. To reveal the secret picture, type in the program in listing 16.10. There's no cheating this time! This one isn't in the `\examples` folder—you have to type it in if you want to see the mystery picture. But typing in all the numbers can be a bit tedious, so you can find the `dots` list in a text file in the `\examples` folder, or on the web site.

Listing 16.10 *Connect-the-dots mystery picture*

```
import pygame, sys
pygame.init()

dots = [[221, 432], [225, 331], [133, 342], [141, 310],
        [51, 230], [74, 217], [58, 153], [114, 164],
        [123, 135], [176, 190], [159, 77], [193, 93],
        [230, 28], [267, 93], [301, 77], [284, 190],
        [327, 135], [336, 164], [402, 153], [386, 217],
        [409, 230], [319, 310], [327, 342], [233, 331],
        [237, 432]]

screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
pygame.draw.lines(screen, [255,0,0],True, dots, 2)
pygame.display.flip()
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
```

This time
closed=True

Drawing point-by-point

Let's go back to drawing point-by-point for a moment. It seems kind of silly to draw a tiny circle or rectangle, when all we want to do is change the color of one pixel. Instead of using the `draw` functions, you can access each individual pixel on a surface with the `Surface.set_at()` method. You tell it what pixel you want to set, and what color to set it:

```
screen.set_at([x, y], [0, 0, 0])
```

If we use this line of code in our sine wave example (in place of line 8 in listing 16.8), it looks the same as when we used one-pixel-wide rectangles.

You can also check what color a pixel is already set to with the `Surface.get_at()` method. You just pass it the coordinates of the pixel you want to check, like this: `pixel_color = screen.get_at([320, 240])`. In this example, `screen` was the name of the surface.

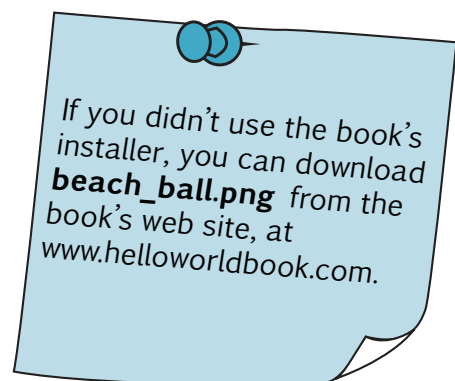
Images

Drawing shapes, lines, and individual pixels on the screen is one way to do graphics. But sometimes we want to use pictures that we get from somewhere else—maybe from a digital photo, something we downloaded from the Web, or something created in an image-editing program. In Pygame, the simplest way to use images is with the `image` functions.

Let's look at an example. We're going to display an image that is already on your hard drive if you installed Python from the book's installer. The installer created an `images` subfolder in the `examples` folder, and the file we're going to use for this example is `beach_ball.png`. So, for example, in Windows, you'd find it at

c:\Program Files\helloworld\examples\images\beach_ball.png.

You should copy the `beach_ball.png` file to wherever you're saving your Python programs as you work through these examples. That way Python can easily find it when the program runs. Once you have the `beach_ball.png` file in the correct location, type in the program in listing 16.11 and try it.



Listing 16.11 Displaying a beach ball image in a Pygame window

```
import pygame, sys
pygame.init()
```

```

screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load("beach_ball.png")
screen.blit(my_ball, [50, 50])
pygame.display.flip()
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT: sys.exit()

```

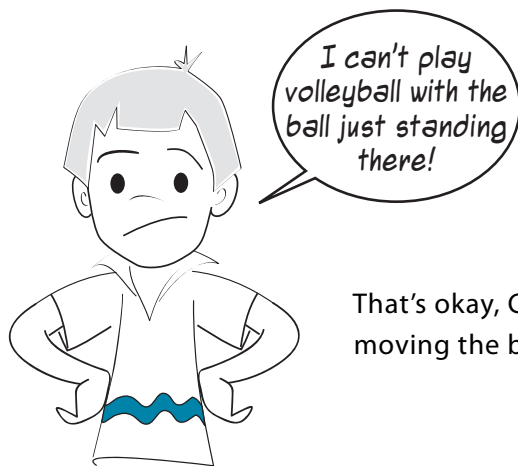
These are the only lines that are new

When you run this program, you should see the image of a beach ball displayed near the top-left corner of the Pygame window, like this:

In listing 16.11, the only lines that are new are lines 5 and 6. Everything else you have seen before in listings 16.4 to 16.10. We replaced the `draw` code from our previous examples with code that loads an image from disk and displays it.



In line 5, the `pygame.image.load()` function loads the image from disk and creates an object called `my_ball`. The `my_ball` object is a surface. (We talked about surfaces a few pages ago.) But we can't see this surface. It's only in memory. The only surface we can see is the `display` surface, which is called `screen`. (We created it in line 3.) Line 6 copies the `my_ball` surface onto the `screen` surface. Then `display.flip()` makes it visible, just like we did before.



That's okay, Carter. Pretty soon we'll start moving the ball around!

You might have noticed a funny-looking thing in line 6 of listing 16.11: `screen.blit()`. What does `blit` mean? See the "WORD BOX" to find out.

WORD BOX

When doing graphics programming, copying pixels from one place to another is something we do quite a lot (like copying from a variable to the screen, or from one surface to another). Pixel-copying has a special name in programming. It's called *blitting*. We say that we *blit* an image (or part of an image, or just a bunch of pixels) from one place to another. It's just a fancy way of saying "copy," but when you see "blit," you know it refers to copying pixels, not copying some other kind of thing.

In Pygame, we copy or *blit* pixels from one *surface* to another. Here we copied the pixels from the `my_ball` surface to the `screen` surface.

In line 6 of listing 16.11, we *blitted* the beach ball image to the location `50, 50`. That means 50 pixels from the left edge and 50 pixels from the top of the window. When you're working with a `surface` or `rect`, this sets the location of the top-left corner of the image. So the left edge of the beach ball is 50 pixels from the left edge of the window, and the top edge of the beach ball is 50 pixels from the top of the window.

Let's get moving!

Now that we can get graphics onto our Pygame window, let's start moving them around. That's right, we're going to do some animation! Computer animation is really just about moving images (groups of pixels) from one place to another. Let's try moving our beach ball.

To move it, we need to change its location. First, let's try moving it sideways. To make sure we can see the motion, let's move it 100 pixels to the right. The left-right direction (horizontal) is the first number in the pair of numbers that specify location. So to move something to the right by 100 pixels, we need to increase the first number by 100. We'll also put in a delay so we can see the animation happen.

Change the program from listing 16.11 to look like the one in listing 16.12. (You'll need to add lines 8, 9, and 10 before the `while` loop.)

Listing 16.12 Trying to move a beach ball

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load('beach_ball.png')
```

```

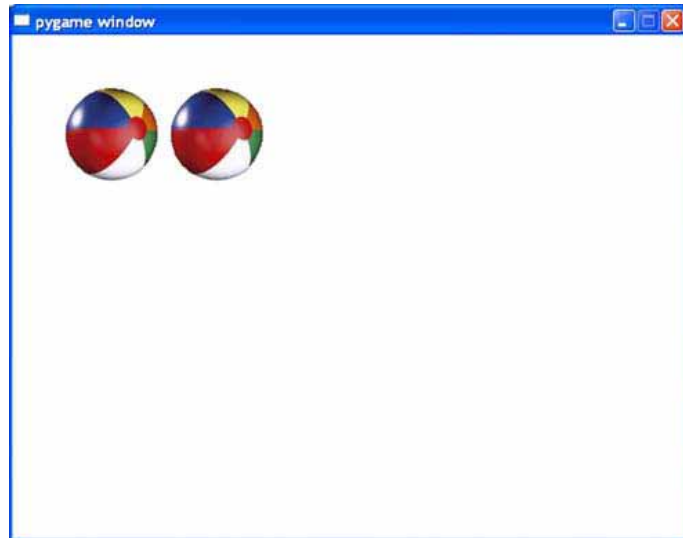
screen.blit(my_ball,[50, 50])
pygame.display.flip()
pygame.time.delay(2000)
screen.blit(my_ball,[150, 50])
pygame.display.flip()
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

```

These are the three new lines

Run the program and see what happens. Did the ball move? Well, sort of. You should have seen two beach balls:

The first one showed up in the original position, and then the second one appeared to the right of it a couple of seconds later. So we *did* move the beach ball to the right, but we forgot one thing. We need to erase the first ball!



Animation

When doing animation with computer graphics, there are two steps to moving something:

- 1 We draw the thing in its new position.
- 2 We erase the thing from its old position.

We already saw the first part. We drew the ball in a new position. Now we have to erase the ball from where it was before. But what does “erasing” really mean?

Erasing images

When you draw something on paper or on a blackboard, it’s easy to erase it. You just use an eraser, right? But what if you made a painting? Let’s say you made a painting of blue sky, and then you painted a bird in the sky. How would you “erase” the bird? You can’t erase paint. You’d have to paint some new blue sky over where the bird was.

Computer graphics are like paint, not like pencil or chalk. In order to “erase” something, what you really have to do is “paint over” it. But what do you paint over with? In the case of

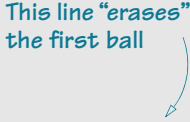
your sky painting, the sky is blue, so you'd paint over the bird with blue. Our background is white, so we have to paint over the beach ball's original image with white.

Let's try that. Modify your program in listing 16.12 to match listing 16.13. There's only one new line to add.

Listing 16.13 Trying to move a beach ball again

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load('beach_ball.png')
screen.blit(my_ball,[50, 50])
pygame.display.flip()
pygame.time.delay(2000)
screen.blit(my_ball, [150, 50])
pygame.draw.rect(screen, [255,255,255], [50, 50, 90, 90], 0)
pygame.display.flip()
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
```

This line "erases" the first ball



We added line 10 to draw a white rectangle over the first beach ball. The beach ball image is about 90 pixels wide by 90 pixels high, so that's how big we made the white rectangle. If you run the program in listing 16.13, it should look like the beach ball moves from its original location to the new location.

What's under there?

Painting over our white background (or the blue sky in your painting) is fairly easy. But what if you painted the bird on a cloudy sky? Or on a background of trees? Then you'd have to paint over the bird with clouds or trees to erase it. The important idea here is that you have to keep track of what's in the background, "underneath" your images, because when you move them, you have to put back or repaint what was there before.

This is pretty easy for our beach ball example, because the background is just white. But if the background was a scene of a beach, it would be trickier. Instead of painting just white, we'd have to paint the correct portion of the background image. Another option would be to repaint the whole scene and then place the beach ball in its new location.

Smoother animation

So far, we have made our ball move once! Let's see if we can get it moving in a more realistic way. When animating things on the screen, it's usually good to move them in small steps, so the motion appears smooth. Let's try moving our ball in smaller steps.

We're not just going to make the steps smaller—we're going to add a loop to move the ball (because we want to make many small steps). Starting with listing 16.13, edit the code so it looks like listing 16.14.

Listing 16.14 Moving a beach ball image smoothly

```
import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load('beach_ball.png')
x = 50
y = 50
screen.blit(my_ball,[x, y])
pygame.display.flip()
for loop in range (1, 100):
    pygame.time.delay(20)
    pygame.draw.rect(screen, [255,255,255], [x, y, 90, 90], 0)
    x = x + 5
    screen.blit(my_ball, [x, y])
    pygame.display.flip()

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
```

Annotations for Listing 16.14:

- Add these lines**: Points to the `pygame.time.delay(20)` line.
- Uses x and y (instead of numbers)**: Points to the `screen.blit(my_ball,[x, y])` line.
- Starts a for loop**: Points to the `for loop in range (1, 100):` line.
- Changes time.delay value from 2000 to 20**: Points to the `pygame.time.delay(20)` line.

If you run this program, you should see the ball moving from its original position over to the right side of the window.

Keeping the ball moving

In the previous program, the ball moved over to the right side of the window, then stopped. Now we'll try to keep the ball moving.

If we just keep increasing **x**, what will happen? The ball will keep moving to the right as its x-value increases. But our window (the display surface) stops at **x = 640**. So the ball will just disappear. Try changing the **for**

loop in line 10 of listing 16.14 to this:

```
for loop in range (1, 200):
```

Now that the loop runs twice as long, the ball disappears off the edge! If we want to continue seeing the ball, we have two choices:

- We make the ball *bounce* off the side of the window.
- We make the ball *wrap around* to the other side of the window.

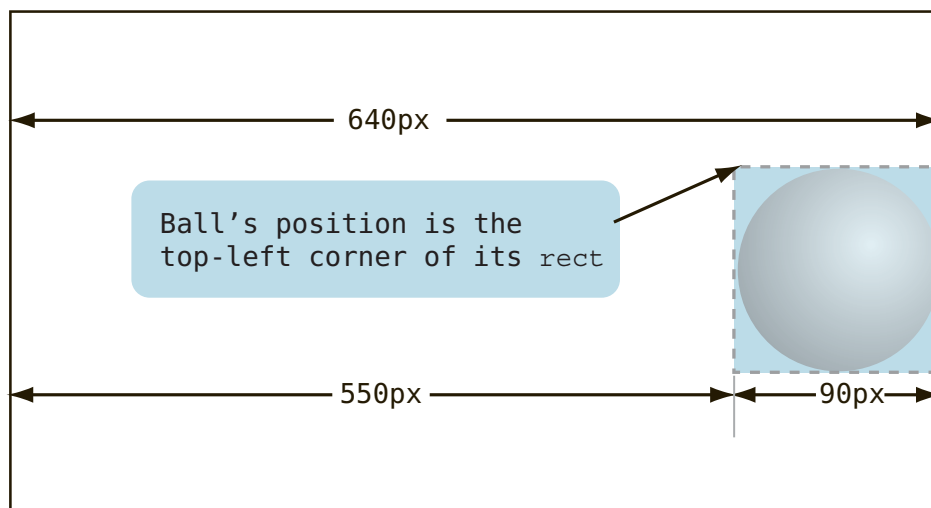
Let's try both to see how to do them.

Bouncing the ball

If we want to make the ball appear to *bounce* off the side of the window, we need to know when it “hits” the edge of the window, and then we need to reverse its direction. If we want to keep the ball moving back and forth, we need to do this at both the left and right edges of the window.

At the left edge, it’s easy, because we just check for the ball’s position to be 0 (or some small number).

At the right side, we need to check to see if the right side of the ball is at the right side of the window. But the ball’s position is set from its left side (the top-left corner), not its right side. So we have to subtract the width of the ball:



When the ball is moving toward the right edge of the window, we need to bounce it (reverse its direction) when its position is 550.

To make things easier, we’re going to make some changes to our code:

- We’re going to have the ball bouncing around forever (or until we close the Pygame window). Because we already have a `while` loop that runs as long as the window is open, we’ll move our ball-display code inside that loop. (That’s the `while` loop that is in the last part of the program.)
- Instead of always adding 5 to the ball’s position, we’ll make a new variable, `speed`, to determine how fast to move the ball on each iteration. I’m also going to speed the ball up a bit by setting this value at 10.

The new code is in listing 16.15.

Listing 16.15 Bouncing a beach ball

```

import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load('beach_ball.png')
x = 50
y = 50
x_speed = 10

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    pygame.time.delay(20)
    pygame.draw.rect(screen, [255,255,255], [x, y, 90, 90], 0)
    x = x + x_speed
    if x > screen.get_width() - 90 or x < 0:
        x_speed = - x_speed
    screen.blit(my_ball, [x, y])
    pygame.display.flip()

```

Here's the speed variable

Put the ball-display code here, inside the while loop

When ball hits either edge of the window ...

... reverse direction, by making speed the opposite sign

The key to bouncing the ball off the sides of the window is lines 18 and 19. In line 18 (**if `x > screen.get_width() - 90 or x < 0`:**), we detect whether the ball is at the edge of the window, and if it is, we reverse its direction in line 19 (**`x_speed = - x_speed`**).

Try this and see how it works.

Bouncing in 2-D

So far, we only have the ball moving back and forth, or one-dimensional motion. Now, let's get it moving up and down at the same time. To do this, we only need a few changes, as shown in listing 16.16.

Listing 16.16 Bouncing a beach ball in 2-D

```

import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load('beach_ball.png')
x = 50
y = 50
x_speed = 10
y_speed = 10

while True:
    for event in pygame.event.get():

```

Add code for y-speed (vertical motion)

```

        if event.type == pygame.QUIT:
            sys.exit()
    pygame.time.delay(20)
    pygame.draw.rect(screen, [255,255,255], [x, y, 90, 90], 0)
    x = x + x_speed
    y = y + y_speed
    if x > screen.get_width() - 90 or x < 0:
        x_speed = - x_speed
    if y > screen.get_height() - 90 or y < 0:
        y_speed = -y_speed
    screen.blit(my_ball, [x, y])
    pygame.display.flip()

```

← Add code for y-speed (vertical motion)

Bounces ball off top or bottom of window

We added lines 9 (`y_speed = 10`), 17 (`y = y + y_speed`), 20 (`if y > screen.get_height() - 90 or y < 0:`), and 21 (`y_speed = -y_speed`) to the previous program. Try it now and see how it works!

If you want to slow down the ball, there are a couple of ways to do it:

- You can reduce the speed variables (`x_speed` and `y_speed`). This reduces how far the ball moves on each animation step, so the motion will also be smoother.
- You could also increase the delay setting. In listing 16.16, it's 20. That is measured in milliseconds, which is thousandths of a second. So each time through the loop, the program waits for 0.02 seconds. If you increase this number, the motion will slow down. If you decrease it, the motion will speed up.

Try playing around with the speed and delay to see the effects.

Wrapping the ball

Now let's look at the second option for keeping the ball moving. Instead of bouncing it off the side of the screen, we're going to *wrap* it around. That means, when the ball disappears off the right side of the screen, it'll reappear on the left side.

To make things simpler, we'll go back to just moving the ball horizontally. The program is in listing 16.17.

Listing 16.17 Moving a beach ball image with wrapping

```

import pygame, sys
pygame.init()
screen = pygame.display.set_mode([640,480])
screen.fill([255, 255, 255])
my_ball = pygame.image.load('beach_ball.png')
x = 50
y = 50
x_speed = 5

```

```

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
    pygame.time.delay(20)
    pygame.draw.rect(screen, [255,255,255], [x, y, 90, 90], 0)
    x = x + x_speed
    if x > screen.get_width():
        x = 0
    screen.blit(my_ball, [x, y])
    pygame.display.flip()

```

If the ball is at the far right ...

... start over at the left side

In lines 17 (`if x > screen.get_width():`) and 18 (`x = 0`), we detected when the ball reached the right edge of the window, and we moved it back, or wrapped it back, to the left side.

You might have noticed that, when the ball appears on the right, it “pops in” at [0, 50]. It would look more natural if it “slid in” from off screen. Change line 18 (`x = 0`) to `x = -90` and see if you notice the difference.

```
0011000111100111000011011010001101101011100110001100110011010011000111001000110
```

What did you learn?

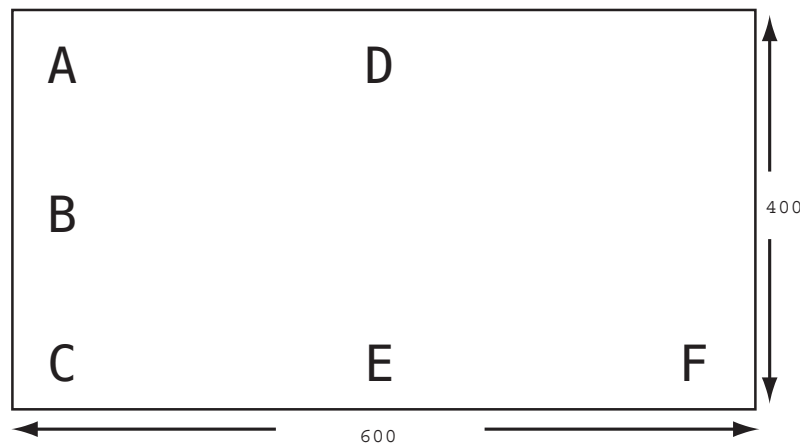
Whew! That was a busy chapter! In it, you learned

- how to use Pygame.
- how to run programs from SPE.
- how to create a graphics window and draw some shapes in it.
- how to set colors in computer graphics.
- how to copy images to a graphics window.
- how to animate images, including “erasing” them when you move them to a new place.
- how to make a beach ball “bounce” around the window.
- how to make a beach ball “wrap” around the window.

Test your knowledge

- 1 What color does the RGB value [255, 255, 255] make?
- 2 What color does the RGB value [0, 255, 0] make?
- 3 What Pygame method can you use to draw rectangles?
- 4 What Pygame method can you use to draw lines joining a number of points together?
- 5 What does the term “pixel” mean?
- 6 In a Pygame window, where is the location [0, 0]?

- 7 If a Pygame window is 600 pixels wide by 400 pixels high, what letter in the diagram below is at [50, 200]?



- 8 What letter in the diagram is at location [300, 50]?
 9 What Pygame method is used to copy images to a surface (like the display surface)?
 10 What are the two main steps when you're "moving" or animating an image?

Try it out

- 1 We talked about drawing circles and rectangles. Pygame also has methods to draw lines, arcs, ellipses, and polygons. Try using these to draw some other shapes in a program.

You can find out more about these methods in the Pygame documentation, at www.pygame.org/docs/ref/draw.html. If you don't have Internet access, you can also find it on your hard drive (it's installed with Pygame), but it can be hard to find. Search your hard drive for a file called **pygame_draw.html**.

You can also use Python's help system (which we talked about at the end of chapter 6). One thing SPE doesn't have is an interactive shell that works, so start IDLE and type the following:

```
>>> import pygame
>>> help()
help> pygame.draw
```

You'll get a list of the different draw methods and some explanation for each one.

- 2 Try changing one of the sample programs that uses the beach ball image to use a different image. You can find some sample images in the **examples\images** folder, or you can download or draw one of your own. You could also use a piece of a digital photo.

- 3 Try changing the `x_speed` and `y_speed` values in listing 16.16 or 16.17 to make the ball move faster or slower and in different directions.
- 4 Try to change listing 16.16 to make the ball “bounce” off an invisible wall or floor that isn’t the edge of the window.
- 5 In listings 16.6 to 16.10 (the modern art, sine wave, and mystery picture programs), try moving the line `pygame.display.flip` inside the `while` loop. To do that, just indent it four spaces. After that line, and also inside the `while` loop, add a delay with this line and see what happens:

```
pygame.time.delay(30)
```