Beginning\_Game\_Development\_Python\_Pygame\_c03

CHAPTER 3

■■■

Introducing Pygame

Have you ever opened up your computer and had a look inside the case? No need to do it

now, but you will find that it is built from a number of parts necessary to deliver your comput-

ing experience. The video card generates an image and sends a signal to your monitor. The

sound card mixes sound together and sends audio to your speakers. Then there are the input

devices, such as the keyboard, mouse, and joystick(s), and a variety of other electronic giz-

mos—all of which are essential in making a game.

In the early days of home computers, programmers with bad haircuts and thick-rimmed

glasses had to come to grips with each of the computer’s components. The game programmer

had to read the technical manual for each device in order to write the computer code to commu-

nicate with it—all before working on the actual game. The situation only got worse when the

manufacturers brought out different devices and versions of existing devices with new capabili-

ties. Programmers wanted to support as many devices as possible so there was a bigger market

for their games, but they found themselves bogged down in the details of working with these new

graphics and sound cards. It was also a pain for the game-buying public, who had to carefully

check the box to see if they had the right combination of devices to make the game work.

Things got a little easier with the introduction of graphical operating systems like Micro-

soft Windows. They gave the game programmer a single way of communicating with the

devices. It meant that the programmer could throw away the technical manuals because the

manufacturers supplied drivers, small programs that handle the communication between the

operating system and the hardware.

Fast-forward to more recent times, when programmers still have bad haircuts but thinner

rims on their glasses. The life of a game programmer is still not an easy one. Even though there

is a common way of communicating with graphics, audio, and input, it can still be tricky to

write games because of the variety of hardware on the market. The cheap family PC that Mom

bought at the local superstore is vastly different from the top-of-the-range machine purchased

by a company executive. It’s this variety that makes it such an effort to initialize the hardware

and ready it for use in the game. Fortunately, now that Pygame is here we have a way of creat-

ing games without having to worry about these details (and game programmers have time to go

out and get decent haircuts).

In this chapter we will introduce you to Pygame and explain how to use it to create a

graphical display and read the state of input devices.

History of Pygame

Pygame is built on another game creation library called Simple DirectMedia Layer (SDL). SDL

was written by Sam Lantinga while he was working for Loki Software (a now-defunct game

company) to simplify the task of porting games from one platform to another. It provided a

common way to create a display on multiple platforms as well as work with graphics and input

devices. Because it was so simple to work with, it became very popular with game developers

when it was released in 1998, and has since been used for many hobby and commercial games.

SDL was written in C, a language commonly used for games because of its speed and abil-

ity to work with the hardware at a low level. But developing in C, or its successor C++, can be

slow and error prone. So programmers produced bindings to their favorite languages, and SDL

can now be used from just about any language out there. One such binding is Pygame, which

lets Python programmers use the powerful SDL library.

Pygame and SDL have been in active development for many years, and because they are

both open source, a large number of programmers have worked to refine and enhance this

superb tool for creating games.

Installing Pygame

You can download Pygame for your operating system from www.pygame.org/. Click the Down-

loads link and it will take you to a page listing the installers for a variety of systems, including

Windows, Linux, and Mac. Many Unix/Linux distribution repositories already include Pygame;

on this page you can learn more about these distributions.

Once you have installed an appropriate package, you can test it by opening the Python

interpreter and entering the following two lines:

>>> import pygame

>>> print pygame.ver

If Pygame was installed successfully, then you should see the version displayed:

1.7.1release

At the time of this writing version 1.7.1 is the most recent, but you may find that a later version

is available when this book is released. The example code will still work, since newer versions

of Pygame tend to be backward compatible.

Using Pygame

The Pygame package contains a number of modules that can be used independently. There is

a module for each of the devices that you might use in a game, and many others to make game

creation a breeze. See Table 3-1 for all the Pygame modules. You access these modules through

the pygame namespace; for instance, pygame.display refers to the display module.

Some of the modules you will use in every game. You will always have some sort of display,

so the display module is essential, and you will definitely need some kind of input, whether it

is keyboard, joystick, or mouse. Other modules are less commonly used, but in combination

they give you one of the most powerful game creation tools around.

Table 3-1. Modules in the Pygame Package

Module NamePurpose

pygame.cdromAccesses and controls CD drives

pygame.cursorsLoads cursor images

pygame.displayAccesses the display

pygame.drawDraws shapes, lines, and points

pygame.eventManages external events

pygame.fontUses system fonts

pygame.imageLoads and saves an image

pygame.joystickUses joysticks and similar devices

pygame.keyReads key presses from the keyboard

pygame.mixerLoads and plays sounds

pygame.mouseManages the mouse

pygame.moviePlays movie files

pygame.musicWorks with music and streaming audio

pygame.overlayAccesses advanced video overlays

pygameContains high-level Pygame functions

pygame.rectManages rectangular areas

pygame.sndarrayManipulates sound data

pygame.spriteManages moving images

pygame.surfaceManages images and the screen

pygame.surfarrayManipulates image pixel data

pygame.timeManages timing and frame rate

pygame.transformResizes and moves images

\* For complete documentation on the Pygame modules, see

www.pygame.org/docs/.

Not all of the modules in Table 3-1 are guaranteed to be present on every platform. It is

possible that the hardware the game is running on does not have certain capabilities, or that

required drivers are not installed. If this is the case, Pygame will set the module to None, which

makes it easy to test for. The following snippet will detect whether the pygame.font module is

available and exit if it isn’t:

if pygame.font is None:

print "The font module is not available!"

exit()

Hello World Revisited

As I mentioned in Chapter 1, there is a tradition when learning new languages that the first

code you write displays the text “Hello, World!” on the screen. Technically we have already

done this with a print 'Hello, World!' statement—but it is a little disappointing because as

game programmers we are interested in creating appealing visuals and a line of text just does

not cut it! We are going to create a Hello World script with Pygame that opens a graphical win-

dow on your desktop and draws an image under the standard mouse cursor. When run, you’ll

see a window similar to what’s shown in Figure 3-1.

Figure 3-1. Hello World in Pygame

See Listing 3-1 for the code. Run it now if you like; we will go through it step by step in this

chapter.

Listing 3-1. Hello World Redux (helloworld.py)

#!/usr/bin/env python

background\_image\_filename = 'sushiplate.jpg'

mouse\_image\_filename = 'fugu.png'

import pygame

from pygame.locals import \*

from sys import exit

pygame.init()

screen = pygame.display.set\_mode((640, 480), 0, 32)

pygame.display.set\_caption("Hello, World!")

background = pygame.image.load(background\_image\_filename).convert()

mouse\_cursor = pygame.image.load(mouse\_image\_filename).convert\_alpha()

while True:

for event in pygame.event.get():

if event.type == QUIT:

exit()

screen.blit(background, (0,0))

x, y = pygame.mouse.get\_pos()

x-= mouse\_cursor.get\_width() / 2

y-= mouse\_cursor.get\_height() / 2

screen.blit(mouse\_cursor, (x, y))

pygame.display.update()

We need two images for Listing 3-1: one to use as a background and another to draw as our

mouse cursor. You can download the files for this and the other samples from the Source Code/

Download section at the Apress web site. If you don’t have Internet access at the moment, you

can use image files you have on your hard drive, or make them with any graphics- or photo-

editing software. Any image is fine for the background, as long as it is at least 640 by 480 in size

(any larger and the excess will be clipped). For the mouse cursor, you will need a smaller image

that fits comfortably inside the background; a good size is 80 by 80. To continue with the fugu

theme of the first chapter, the official background will be a picture of bowls and chopsticks, and

a picture of a very raw fugu for the mouse cursor. The first two lines set the file names of the

images; if you are using different images, you should replace the file names with the location of

your images. Let’s break this script into bite-sized chunks. At the top of the script we import the

external modules, classes, functions, and so forth we will need when running the example:

import pygame

from pygame.locals import \*

from sys import exit

The first line imports the pygame package, which gives us access to all of its submodules, such

as pygame.image and pygame.sound. The second line imports a number of functions and con-

stants (values that don’t change) into the top-level namespace. It isn’t essential to do this in

order to use Pygame, but it is convenient because we don’t have to precede frequently used

values with the pygame namespace. The last import statement imports a single function from

sys (a module in the standard library). As you may have guessed, the purpose of exit is to

immediately finish with the script. Calling it will cause the Pygame window to disappear and

Python to close. The script will call exit when the user clicks the close button; otherwise, the

user would have no way of closing the window!

■Tip If you get into a situation where you can’t close the Pygame window, you may be able to stop Python

in its tracks by pressing Ctrl+C.

This rather simple line of Python code actually does a lot of work:

pygame.init()

It initializes each of the submodules in the pygame package, which may load drivers and query

hardware so that Pygame is ready to use all the devices on your computer. You can initialize

only the modules you intend to use by calling the init function in each submodule individu-

ally; for example, pygame.sound.init() will initialize the sound module. This can make the

script start a little quicker because only the modules you actually use will be initialized. For

games you will require most, if not all, of the modules—so we will stick with this catchall initial-

ize function. After we call it, we have the full power of Pygame at our disposal!

After initializing Pygame we need to create a display surface:

screen = pygame.display.set\_mode((640, 480), 0, 32)

pygame.display.set\_caption("Hello, World!")

The display could be a window on your desktop or it could be the entire screen, but you always

access it via a Pygame Surface object. The call to pygame.display.set\_mode in our script returns

the Surface object representing the window on your desktop. It takes three parameters;

only the first is required, which should be a tuple containing the width and height of the dis-

play we want to create. Our window will be 640 ×480 pixels, which is large enough so we can see

what is happening, but not so large that it obscures too much of the desktop. The next param-

eter we give to set\_mode is a value containing flags used in the display creation. A flag is a

feature that can be switched on or off; you can combine several flags together with the bitwise

OR operator (|). For instance, to create a double-buffered hardware surface, set the flags

parameter to DOUBLEBUF|HWSURFACE. See Table 3-2 for the flags you can use. I will cover them in more detail in the “Opening a Display” section later in this chapter. We won’t be enabling any

of these flags for this first Pygame script, so the value we give for flags is just 0, which is also the

default.

Table 3-2. Flags for pygame.display.set\_mode

FlagPurpose

FULLSCREENCreates a display that fills the entire screen.

DOUBLEBUFCreates a “double-buffered” display. Recommended for HWSURFACE or OPENGL.

HWSURFACECreates a hardware-accelerated display (must be combined with the

FULLSCREEN flag).

OPENGLCreates an OpenGL renderable display.

RESIZABLECreates a resizable display.

NOFRAMERemoves the border and title bar from the display.

The next parameter specifies the depth of the display surface, which is the amount of bits

used to store colors in the display. A bit, or binary digit, is the most fundamental unit of storage

in a computer. Bits have exactly two potential values, 1 or 0, and are arranged in memory as

groups of 8. A group of 8 bits is called a byte. Don’t worry if this sounds like techno-babble to

you; Python tends to hide this kind of thing from the programmer. We will use the value 32 for

our bit depth because it gives us the most colors; see Table 3-3 for other potential bit-depth val-

ues. If you don’t supply a value for the depth or set it to 0, Pygame will use the depth of your

desktop.

Table 3-3. Bit-Depth Values

Bit DepthNumber of Colors

8 bits256 colors, chosen from a larger palette of colors

15 bits32,768 colors, with a spare bit

16 bits65,536 colors

24 bits16.7 million colors

32 bits16.7 million colors, with a spare 8 bits

\* It is possible to have other bit depths, but these are the most

common.

■Note Sometimes Pygame is unable to give us the exact display we ask for. It may be that the graphics

card doesn’t support the features we are requesting. Fortunately, Pygame will choose a display that is com-

patible with the hardware and emulates the display we actually asked for. Thank you, Pygame!

If all goes well, the call to set\_mode will display a Pygame window on your desktop and

return a Surface object, which is then stored in the variable screen. The first thing we do with

our newly created surface is call set\_caption in the display module to set the title bar of the

Pygame window. We set the title to “Hello, World!”—just to make it a valid Hello World script!

Next up we use the load function in pygame.image to load the two images for the background

and mouse cursor. We pass in the file names of the images stored at the start of the script:

background = pygame.image.load(background\_image\_filename).convert()

mouse\_cursor = pygame.image.load(mouse\_image\_filename).convert\_alpha()

The load function reads a file from your hard drive and returns a surface containing the image

data. These are the same type of objects as our display, but they represent images stored in mem-

ory and aren’t visible until we draw them to the main display. The first call to pygame.image.load

reads in the background image and then immediately calls convert, which is a member function

for Surface objects. This function converts the image to the same format as our display, because

it is faster to draw images if the display is of the same depth. The mouse cursor is loaded in a sim-

ilar way, but we call convert\_alpha rather than convert. This is because our mouse cursor image

contains alpha information, which means that portions of the image could be translucent or

completely invisible. Without alpha information in our mouse image, we are limited to an

unsightly square or rectangle as our mouse cursor! The next chapter will cover alpha and image

formats in more detail.

The next line in the script jumps straight into the main game loop:

while True:

This while loop has True as the condition, which means it will loop continually until we break

out of it, or force it to exit in some other way. All games will have a loop similar to this, which

typically repeats once per screen refresh.

Inside the main game loop we have another loop—the event loop, which most games will

also have in one form or another:

for event in pygame.event.get():

if event.type == QUIT:

exit()

An event is how Pygame informs you that something has happened outside your code. Events are

created for many things, from key presses to receiving information from the Internet, and are

queued up for you until you handle them. The function get in the pygame.event module returns

any events waiting for us, which we then loop through in a for loop. For this script, we are only

interested in the QUIT event, which is generated by Pygame when the user clicks the close button

in the Pygame window. So if the event type is QUIT we call exit to shut down, and all other events

are ignored. In a game, of course, we would have to handle a greater number of events.

The next line blits the background image to the screen (blitting means copying from one

image to another):

screen.blit(background, (0,0))

This line uses the blit member function of the screen Surface object, which takes a source

image—in this case, our 640 ×480 background—and a tuple containing the destination posi-

tion. The background will never move; we just want it to cover the entire Pygame window, so

we blit to the coordinate (0, 0), which is the top left of the screen.

■Tip It is important that you blit to every portion of the screen. If you don’t, strange visual effects may occur

when you animate things, and your game may look different on each computer it is run on. Try commenting

out the call to screen.blit to see what happens.

After we draw the background, we want to draw mouse\_cursor underneath the usual

mouse pointer:

x, y = pygame.mouse.get\_pos()

x -= mouse\_cursor.get\_width()/2

y -= mouse\_cursor.get\_height()/2

screen.blit(mouse\_cursor, (x, y))

Getting the position of the mouse is nice and simple; the pygame.mouse module contains all we

need to work with the mouse, including get\_pos, which returns a tuple containing the mouse

coordinates. The first line unpacks this tuple into two values for convenience: x and y. We

could use these two values as coordinates when we blit the mouse cursor, but that would place

the top-left corner of the image under the mouse, and we want the center of the image to be

under the mouse. So we do a little math (fear not!) to adjust x and y so that the mouse image is

moved up by half its height and left by half its width. Using these coordinates places the center

of the image right under the mouse pointer, which looks better. At least it does for an image of

a fish—if you want to use a more typical pointer image, adjust the coordinates so that the tip

lies underneath the real mouse coordinates.

Blitting the mouse image is done in the same way as blitting the background, but we use

the coordinates we calculated rather than (0, 0). This is enough to create the effect we are look-

ing for, but there is one more thing we have to do before we can see anything:

pygame.display.update()

When you build an image through blits to the screen surface, you won’t see them right away.

This is because Pygame first builds up an image to a back buffer, which is an invisible display

in memory, before it is displayed. If we didn’t have this step, the user would see individual

blits as they happen, which would flicker most unpleasantly. For games programmers, flicker

is the enemy! We want to see silky-smooth, convincing animation. Fortunately a call to

pygame.display.update() is all we need to ensure that the image we have created in memory

is shown to the user without flicker.

When you run this script, you should see something like Figure 3-1. If you are using the

“official” images, then an odd-looking fish will dutifully follow the mouse cursor.

Understanding Events

In Hello World we only handled the QUIT event, which is essential unless you want to have

immortal Pygame windows! Pygame creates other events to inform you of things such as

mouse movement and key presses.

Events can be generated at any time, no matter what your program is currently doing. For

example, your code could be drawing a tank on the screen when the user presses the fire but-

ton on the joypad. Because you can’t react to events the instant they happen, Pygame stores

them in a queue until you are ready to handle them (typically at the beginning of the main

game loop). You can think of the event queue as a line of people waiting to get into a building,

each carrying specific information about an event. When the player presses the fire button, the

joystick event arrives, carrying information about which key was pressed. Similarly, when

the player releases the fire button, a clone of the same joystick event arrives with information

about the button that was released. They could be followed by a mouse event and key event.

Retrieving Events

In the earlier example, we called pygame.event.get() to retrieve all the events and remove

them from the queue, which is like opening the door and letting everyone in. This is probably

the best way to deal with events, as it ensures we have handled everything before we go on to

draw something to the screen—but there are other ways to work with the event queue. If you

call pygame.event.wait(), Pygame will wait for an event to occur before it returns, which is like

waiting by the door until someone arrives. This function isn’t often used for games because it

suspends the script until something happens, but it can be useful for Pygame applications that

cooperate more with other programs on your system, such as media players. An alternative is

pygame.event.poll(), which returns a single event if there is one waiting, or a dummy event of

type NOEVENT if there are no events in the queue. Whatever method you use, it is important to

not allow them to build up, because the event queue is limited in size and events will be lost if

the queue overflows.

It is necessary to call at least one of the event-handling functions at regular intervals so

that Pygame can process events internally. If you don’t use any of the event-handling func-

tions, you can call pygame.event.pump() in place of an event loop.

Event objects contain a few member variables that describe the event that occurred. The

information they contain varies depending on the event. The only thing common to all event

objects is type, which is a value that indicates the type of the event. It is this value that you first

query so you can decide what to do with it. Table 3-4 lists the standard events that you may

receive; we will go over a few of them in this chapter.

Table 3-4. Standard Events

EventPurposeParameters

QUITUser has clicked the close button.none

ACTIVEEVENTPygame has been activated or hidden.gain, state

KEYDOWNKey has been pressed.unicode, key, mod

KEYUPKey has been released.key, mod

MOUSEMOTIONMouse has been moved.pos, rel, buttons

MOUSEBUTTONDOWNMouse button was pressed.pos, button

MOUSEBUTTONUPMouse button was released.pos, button

JOYAXISMOTIONJoystick or pad was moved.joy, axis, value

JOYBALLMOTIONJoy ball was moved.joy, ball, rel

JOYHATMOTIONJoystick hat was moved.joy, hat, value

JOYBUTTONDOWNJoystick or pad button was pressed.joy, button

JOYBUTTONUPJoystick or pad button was released.joy, button

VIDEORESIZEPygame window was resized.size, w, h

VIDEOEXPOSEPart or all of the Pygame window was exposed.none

USEREVENTA user event has occurred.code

Let’s write a simple Pygame script to display all the events that are generated. Listing 3-2

uses pygame.event.wait() to wait for a single event. As soon as it gets one, it turns it into a

string with str and adds it to a list. The rest of the code displays the new event along with as

many previous events as it can fit on the screen. It uses the font module to display text (which

we will discuss later).

■Tip If you change the fill color in Listing 3-2 to (0,0,0) and the color of the font to (0, 255, 0), it will look a

little like Matrix-style code. You may have to use your imagination a little!

Listing 3-2. Displaying the Message Queue

import pygame

from pygame.locals import \*

from sys import exit

pygame.init()

SCREEN\_SIZE = (800, 600)

screen = pygame.display.set\_mode(SCREEN\_SIZE, 0, 32)

font = pygame.font.SysFont("arial", 16);

font\_height = font.get\_linesize()

event\_text = []

while True:

event = pygame.event.wait()

event\_text.append(str(event))

event\_text = event\_text[-SCREEN\_SIZE[1]/font\_height:]

if event.type == QUIT:

exit()

screen.fill((255, 255, 255))

y = SCREEN\_SIZE[1]-font\_height

for text in reversed(event\_text):

screen.blit( font.render(text, True, (0, 0, 0)), (0, y) )

y-=font\_height

pygame.display.update()

If you run Listing 3-2, you will see a simple white window. Move the mouse over it and it will

start to stream MOUSEMOTION events, which are created whenever the mouse changes position

(see Figure 3-2). These events specify the current position of the mouse, how far the mouse has

moved since the last motion event, and which buttons are currently pressed. You can get the cur-

rent position of the mouse with the pygame.mouse module, as we did in the Hello World example,

but you risk losing information about what the player has been doing. This is a particular prob-

lem on desktop computers that do a lot of work in the background, and may occasionally pause

your game for a brief amount of time. For a mouse cursor, you only need to know where the

mouse is at the beginning of every frame, so it is reasonable to use pygame.mouse.get\_pos(). If

you were using mouse movement to drive a tank and the buttons to fire, it would be better to

work with events so that the game can more closely monitor what the player has been doing.

Figure 3-2. Output from the events script

Handling Mouse Motion Events

As you have seen, MOUSEMOTION events are issued whenever you move the mouse over the

Pygame window. They contain these three values:

• buttons—A tuple of three numbers that correspond to the buttons on the mouse.

So buttons[0] is the left mouse button, buttons[1] is the middle button, and buttons[2]

is the right button. If the button is pressed, then its value is set to 1; if it is not pressed, the

value will be 0. Multiple buttons can be pressed at once.

• pos—A tuple containing the position of the mouse when the event was generated.

• rel—A tuple containing the distance the mouse has moved since the last mouse motion

event (sometimes called the mouse mickies).

Handling Mouse Button Events

In addition to motion events, the mouse generates MOUSEBUTTONDOWN and MOUSEBUTTONUP events.

If you click the mouse on the message queue script, you will first see the down event, followed

by an up event when you take your finger off the button. So why have the two events? If you are

using the mouse button as a trigger to fire a rocket, you would only need one of the events, but

you may have a different type of weapon, such as a chain gun that fires continuously while the

button is held down. In this case you would start the chain gun speeding up on the down event

and have it fire until you get the corresponding up event. Both types of mouse button events

contain the following two values:

• button—The number of the button that was pressed. A value of 1 indicates that the left

mouse button was pressed, 2 indicates that the middle button was pressed, and 3 indi-

cates the right button was pressed.

• pos—A tuple containing the position of the mouse when the event was generated.

Handling Keyboard Events

The keyboard and joystick have similar up and down events; KEYDOWN is issued when a key is

pressed, and KEYUP is issued when the key is released. Listing 3-3 demonstrates how you might

respond to KEYUP and KEYDOWN events to move something on screen with the cursor keys. If you

run this listing, you will see a window containing a simple background image. Press up, down,

left, or right and the background will slide in that direction. Take your finger off the cursor key

and the background will stop moving.

Listing 3-3. Using Keyboard Events to Move a Background

background\_image\_filename = 'sushiplate.jpg'

import pygame

from pygame.locals import \*

from sys import exit

pygame.init()

screen = pygame.display.set\_mode((640, 480), 0, 32)

background = pygame.image.load(background\_image\_filename).convert()

x, y = 0, 0

move\_x, move\_y = 0, 0

while True:

for event in pygame.event.get():

if event.type == QUIT:

exit()

if event.type == KEYDOWN:

if event.key == K\_LEFT:

move\_x = -1

elif event.key == K\_RIGHT:

move\_x = +1

elif event.key == K\_UP:

move\_y = -1

elif event.key == K\_DOWN:

move\_y = +1CHAPTER 3 ■ INTRODUCING PYGAME

elif event.type == KEYUP:

if event.key == K\_LEFT:

move\_x = 0

elif event.key == K\_RIGHT:

move\_x = 0

elif event.key == K\_UP:

move\_y = 0

elif event.key == K\_DOWN:

move\_y = 0

x+= move\_x

y+= move\_y

screen.fill((0, 0, 0))

screen.blit(background, (x, y))

pygame.display.update()

Listing 3-3 begins just like Hello World; it imports and initializes Pygame, then loads a

background image. The event loop in this script is different, because it handles KEYDOWN and

KEYUP. These key events both contain the same three values:

• key—This is a number representing the key that was pressed or released. Each physical

key on the keyboard has a constant that begins with K\_. The alphabet keys are K\_a

through K\_z, but there are also constants for all the other keys, such as K\_SPACE and

K\_RETURN. For a complete list of the key constants you can use, see www.pygame.org/docs/

ref/key.html.

• mod—This value represents keys that are used in combination with other keys, such as

Shift, Alt, and Ctrl. Each of these modifier keys are represented by a constant that begins

with KMOD\_, such as KMOD\_SHIFT, KMOD\_ALT, and KMOD\_CTRL. Check for these values by using

the bitwise AND operator. For example, mod & KMOD\_CTRL will evaluate to True if the Ctrl key

is pressed. www.pygame.org/docs/ref/key.html provides a full list of the modifier keys.

• unicode—This is the Unicode value of the key that was pressed. It is produced by com-

bining the pressed key with any of the modifier keys that was pressed. There is a Unicode

value for every symbol in the English alphabet and other languages. You won’t often use

this value in a game because keys tend to be used more like switches than for entering

text. An exception would be for entering a high score table, where you would want the

player to be able to type non-English letters as well as mix upper- and lowercase.

Inside the handler for KEYDOWN we check for the four key constants that correspond to the

cursor keys. If K\_LEFT is pressed, then the value of move\_x is set to –1; if K\_RIGHT is pressed, it is

set to +1. This value is later added to the x coordinate of the background in order to move it left

or right. There is also a move\_y value, which is set if K\_UP or K\_DOWN is pressed, which will move

the background vertically.

We also handle the KEYUP event, because we want the background to stop moving when

the user releases the cursor key. The code inside the handler for KEYUP events is similar to the

down event, but it sets move\_x or move\_y back to zero to stop the background from moving.

After the event loop, all we have to do is add the values move\_x and move\_y to x and y, then

draw the background at (x, y). The only thing you haven’t seen before is screen.fill((0, 0, 0)),

which is used to clear the display to black (colors are explained in Chapter 4). This line is neces-

sary because if we move the background image it no longer covers the whole display— which I

guess would technically mean it is no longer a background!

Filtering Events

Not all events need to be handled in every game, and there are often alternative ways of getting the

information that events might give you. For example, if you are using pygame.mouse.get\_pos() you

will not need to respond to the MOUSEMOTION event.

Occasionally you also need to suspend the handling of certain events. If you were to play a

cut scene movie between levels, you would probably want to ignore input events until it is fin-

ished. The Pygame event module has a number of functions to help you just do that.

You can block events from the event queue with the set\_block function. For example, the

following line will disable mouse movement:

pygame.event.set\_blocked(MOUSEMOTION)

If you pass in a list of event types, all those events will be blocked. For example, the follow-

ing line will disable all keyboard input by blocking both KEYDOWN and KEYUP events:

pygame.event.set\_blocked([KEYDOWN, KEYUP])

If you want to unblock all events, pass the value of None to set\_blocked. This line will allow

all events to occur in the event queue:

pygame.event.set\_blocked(None)

The opposite of set\_blocked is set\_allowed, which selects the events that should be

allowed (unblocked). It also takes a single event type, or a list of event types. But if you pass in

the value of None, it effectively blocks all events. You can ask Pygame if an event is currently

blocked with pygame.event.get\_block, which takes a single event type.

Posting Events

Generally it is Pygame that creates all the events for you, but you can create your own. You

could use this ability to play back demos (by replicating the player’s input), or simulate the

effects of a cat walking across the keyboard (I like to make my games cat proof).

To send an event, you first construct an event object with pygame.event.Event and then

post it with pygame.event.post. The event will be placed on the end of the queue, ready for

retrieval in the event loop. Here’s how to simulate the player pressing the spacebar:

my\_event = pygame.event.Event(KEYDOWN, key=K\_SPACE, mod=0, unicode=u' ')

[pgame.event.post](http://pgame.event.post)(my\_event)

The Event constructor takes the type of the event, such as one of the events in Table 3-4,

followed by the values the event should contain. Since we are simulating the KEYDOWN event, we

need to supply all the values that the event handler would expect to be there. If you prefer, you

can supply these values as a dictionary. This line will create the same event object:

my\_event = pygame.event.Event(KEYDOWN, {"key":K\_SPACE, "mod":0, "unicode":u' '})

In addition to simulating Pygame-generated events, you can create completely new

events. All you have to do is use a value for the event that is above USEREVENT, which is the max-

imum value that Pygame will use for its own event IDs. This can sometimes be useful if you

want to do something in the event loop before you go on to draw to the screen. Here’s an exam-

ple of a user event to respond to a cat walking over the keyboard:

CATONKEYBOARD = USEREVENT+1

my\_event = pygame.event.Event(CATONKEYBOARD, message="Bad cat!")

pgame.event.post(my\_event)

Handling user events is done in the same way as the usual events that Pygame generates—

just check the event type to see if it matches your custom event. Here’s how you might handle

a CATONKEYBOARD event:

for event in pygame.event.get():

if event.type == CATONKEYBOARD:

print event.message

Opening a Display

I deliberately glossed over opening a display in the Hello World example because we only

needed a simple display, but Pygame has a variety of options for displays. The type of display

you create depends on the game. It is generally easier to used fixed resolution (display size)

because it can simplify your code. Your decision also depends on how much action you will

have in the game—the more things you have moving on screen at one time, the slower the

game will run. You may have to compensate by selecting a lower resolution (which will speed

things up again).

The best solution is usually to let the player decide what resolution they want to run in so

that they can adjust the display until they have a good compromise between visual quality and

how smoothly the game runs. If you go this route, you will have to make sure that your game

looks OK in all potential resolutions!

Don’t worry about this until it comes time to write your game. Just select a resolution

that works for you while you are experimenting with Pygame scripts, but feel free to experi-

ment a little.

Full-Screen Displays

In Hello World we used the following line to create a Pygame window:

screen = pygame.display.set\_mode((640, 480), 0, 32)

The first parameter is the size of the window we want to create. A size of (640, 480) creates a

small window that will fit comfortably on most desktops, but you can select a different size if

you wish. Running in a window is great for debugging, but most games fill the entire screen

with the action and don’t have the usual borders and title bar. Full-screen mode is usually

faster because your Pygame script doesn’t have to cooperate with other windows on your desk-

top. To set full-screen mode, use the FULLSCREEN flag for the second parameter of set\_mode:

screen = pygame.display.set\_mode((640, 480), FULLSCREEN, 32)

■Caution If something goes wrong with your script in full-screen mode, it can sometimes be difficult to

get back to your desktop. Therefore, it’s best to test it in windowed mode first. You should also provide an

alternative way to exit the script because the close button is not visible in full-screen mode.

When you go full screen, your video card will probably switch to a different video mode,

which will change the width and height of the display, and potentially how many colors it can

show at one time. Video cards only support a few combinations of size and number of colors,

but Pygame will help you if you try to select a video mode that the card does not support

directly. If the size of display you ask for isn’t supported, Pygame will select the next size up and

copy your display to the center of it, which may lead to black borders at the top and bottom of

your display. To avoid these borders, select one of the standard resolutions that virtually all

video cards support: (640, 480), (800, 600), or (1024, 768). To see exactly what resolutions your

display supports, you can use pygame.display.list\_modes(), which returns a list of tuples con-

taining supported resolutions. Let’s try this from the interactive interpreter:

>>> import pygame

>>> pygame.init()

>>> pygame.display.list\_modes()

[(800, 600), (1280, 1024), (1280, 960), (1280, 800), (1280, 768), (1280, 720),

(1152, 864), (1088, 612), (1024, 768), (960, 600), (848, 480), (800, 600),

(720, 576), (720, 480), (640, 480), (640, 400), (512, 384), (480, 360), (400, 300),

(320, 240), (320, 200), (640, 480)]

If the video card can’t give you the number of colors you asked for, Pygame will convert

colors in the display surface automatically to fit (which may result in a slight drop in image

quality).

Listing 3-4 is a short script that demonstrates going from windowed mode to full-screen

mode. If you press the F key, the display will fill the entire screen (there may be a delay of a few

seconds while this happens). Press F a second time, and the display will return to a window.

Listing 3-4. Full-Screen Example

background\_image\_filename = 'sushiplate.jpg'

import pygame

from pygame.locals import \*

from sys import exit

pygame.init()

screen = pygame.display.set\_mode((640, 480), 0, 32)

background = pygame.image.load(background\_image\_filename).convert()

Fullscreen = False

while True:

for event in pygame.event.get():

if event.type == QUIT:

exit()

if event.type == KEYDOWN:

if event.key == K\_f:

Fullscreen = not Fullscreen

if Fullscreen:

screen = pygame.display.set\_mode((640, 480), FULLSCREEN, 32)

else:

screen = pygame.display.set\_mode((640, 480), 0, 32)

screen.blit(background, (0,0))

pygame.display.update()

Resizable Pygame Windows

Occasionally you may want the user to be able to resize a Pygame window, which you typically do

by clicking on the corner of the window and dragging with the mouse. It’s easy enough to do this by

using the RESIZABLE flag when you call set\_mode. Pygame informs your code if the user has changed

the window size by sending a VIDEORESIZE event that contains the new width and height of the win-

dow. When you get one of these events, you should call pygame.display.set\_mode again to set the

display to the new dimensions. Listing 3-5 demonstrates how to respond to VIDEORESIZE events.

Listing 3-5. Using a Resizable Window

background\_image\_filename = 'sushiplate.jpg'

import pygame

from pygame.locals import \*

from sys import exit

5960

CHAPTER 3 ■ INTRODUCING PYGAME

SCREEN\_SIZE = (640, 480)

pygame.init()

screen = pygame.display.set\_mode(SCREEN\_SIZE, RESIZABLE, 32)

background = pygame.image.load(background\_image\_filename).convert()

while True:

event = pygame.event.wait()

if event.type == QUIT:

exit()

if event.type == VIDEORESIZE:

SCREEN\_SIZE = event.size

screen = pygame.display.set\_mode(SCREEN\_SIZE, RESIZABLE, 32)

pygame.display.set\_caption("Window resized to "+str(event.size))

screen\_width, screen\_height = SCREEN\_SIZE

for y in range(0, screen\_height, background.get\_height()):

for x in range(0, screen\_width, background.get\_width()):

screen.blit(background, (x, y))

pygame.display.update()

When you run this script, it will display a simple Pygame window with a background

image. If you click on the corner or edge of the window and drag with the mouse, the script will

get a VIDEORESIZE event. In the handler to that message is another call to set\_mode, which cre-

ates a new screen surface that matches the new dimensions. The resize message contains the

following values:

• size—This is a tuple containing the new dimensions of the window; size[0] is the width

and size[1] is the height.

• w—This value contains the new width of the window. It is the same value as size[0], but

may be more convenient.

• h—This value contains the new height of the window. It is the same value as size[1], but

may be more convenient.

Because the display size can vary with this script, we draw the background slightly differ-

ently by blitting the background image as many times as necessary to cover the display. The

two calls to range produce the coordinates needed to place these multiple background images.

Most games run in full screen so resizable displays are perhaps not a feature you will use

very often. But it is there in your toolbox if you need it!

Windows with No Borders

Generally when you create a Pygame window you will want a standard window with title bars

and border. It is possible, though, to create a window that doesn’t have these features so that

the user will not be able to move or resize the window, or close it via the close button. One

instance of such a use is the window used for splash screens. Some games can take a while to

load because they contain many image and sound files. If there is nothing visible on the screen

while this is happening, the player may feel that the game is not working and try to launch it

again. To set a display with no borders, use the NOFRAME flag when calling set\_mode. For exam-

ple, the following line will create a “naked” window:

screen = pygame.display.set\_mode(SCREEN\_SIZE, RESIZABLE, 32)

Additional Display Flags

There are a few more flags you can use in a call to set\_mode. I consider them advanced, because

they can hurt performance if used incorrectly or cause compatibility problems on some plat-

forms. It is usually best to use the value 0 for windowed displays and FULLSCREEN for full-screen

displays to ensure your game will work well on all platforms. That said, if you know what you

are doing you can set a few advanced flags for extra performance. There is also no harm in

experimenting (it won’t hurt your computer).

If you set the HWSURFACE flag, it will create what is called a hardware surface. This is a special

kind of display surface that is stored in the memory of your graphics card. It can only be used

in combination with the FULLSCREEN flag, like this:

screen = pygame.display.set\_mode(SCREEN\_SIZE, HWSURFACE | FULLSCREEN, 32)

Hardware surfaces can be faster than surfaces created in system (regular) memory,

because they can take advantage of more features of your graphics card to speed up blitting.

The disadvantage of hardware surfaces is that they are not that well supported on all platforms.

They tend to work on Windows platforms but not so well on others. Hardware surfaces will also

benefit from the DOUBLEBUF flag. This effectively creates two hardware surfaces, but only one is

visible at any one time. The following line will create a double-buffered hardware surface:

screen = pygame.display.set\_mode(SCREEN\_SIZE, DOUBLEBUF | HWSURFACE | FULLSCREEN, ➥

32)

Normally when you call pygame.display.update() an entire screen is copied from memory

to the display—which takes a little time. Double-buffered surfaces allow you to switch to the

new screen instantly and thus makes your game run a little faster.

The last display flag you can use is OPENGL. OpenGL (www.opengl.org/) is a graphics library

that uses the 3D graphics accelerator found on just about every graphics card. The downside of

using this flag is that you will no longer be able to use Pygame’s 2D graphics functions. We will

cover using OpenGL to create 3D in Chapter 9.

■Note If you use a double-buffered display, you should call pygame.display.flip() rather than

pygame.display.update(). This does the instant display switch rather than copying screen data.

Using the Font Module

I promised to cover the font module that we used in the event queue script. The ability to draw

text on the screen can really help with testing scripts; you may also need it to display game

instructions, menu options, and so forth. The font module uses TrueType fonts (TTFs), which

are used on most systems to render high-quality, smooth text. There will be many such fonts

installed on your computer that can be used by the font module.

To use a font, you must first create a Font object. The easiest way to do this is with

pygame.font.SysFont, which uses one of the fonts you have installed on your computer. The

following line creates a Font object for the Arial font (a common font that is easy to read):

my\_font = pygame.font.SysFont("arial", 16)

The first parameter is the name of font you want to create, and the next parameter specifies the

font size in pixels. Pygame will look for a font with the name “arial” in your installed fonts; if it

doesn’t find it, a default font will be returned. You can get a list of the fonts installed on your sys-

tem by calling pygame.font.get\_fonts(). Fonts can also be created directly from .ttf files by

calling pygame.font.Font, which takes a file name. The following line loads the file my\_font.ttf

and returns a Font object:

my\_font = pygame.font.Font("my\_font.ttf", 16)

Once you have created a Font object, you can use it to render text to a new surface. To

render text, use the render member function of Font objects. It creates a new surface contain-

ing the text, which you can then blit to the display. The following line renders a piece of text and

returns a new surface:

text\_surface = my\_font.render("Pygame is cool!", True, (0,0,0), (255, 255, 255))

The first parameter of render is the text you want to render. It has to be a single line; if you

want multiple lines, you will have to break the string and use multiple render calls. The second

parameter is a boolean (True or False), used to enable antialiased text. If it is set to True, the

text will have a modern, smooth look; otherwise, it will appear more pixelated. The next two

parameters of render are the text color, followed by the background color. The background is

optional, if you leave it out (or set it to None), the background will be transparent.

To finish this introduction to the font module, let’s write a small script to render my name

to a surface and save it as an image. Feel free to change the name that is drawn to your own. If

you modify the first line of Listing 3-6, it will do just that.

Listing 3-6. Writing Your Name to an Image File

my\_name = "Will McGugan"

import pygame

pygame.init()

my\_font = pygame.font.SysFont("arial", 64)

name\_surface = my\_font.render(my\_name, True, (0, 0, 0), (255, 255, 255))

pygame.image.save(name\_surface, "name.png")

This script is so simple that we don’t even need to create a display! When you run

Listing 3-6, you won’t see much happen on the screen, but the code will have created an

image file called name.png in the same location as the script. You can open the file with

any image viewer software. Saving the surface to a file is done with the pygame.image module,

which we will cover in the next chapter.

The font module provides other functions as well as Font objects, which you may occa-

sionally need to use. They are mostly informational, designed to retrieve various pieces of

information regarding the fonts. There are functions that will simulate bold and italic text, but

it is better to use a dedicated bold or italic font. For complete details on the font module, see

the documentation at www.pygame.org/docs/ref/font.html.

■Caution Installed fonts vary from computer to computer, and you cannot always rely on a particular font

being present. If Pygame doesn’t find the font you are asking for, it will use a default font that may not look

the same. The solution is to distribute the .ttf files with your game, but make sure you have permission

from the font author to do this! For a free-to-distribute font, you could use something from the BitStream Vera

family (http://en.wikipedia.org/wiki/Bitstream\_Vera).

When Pygame Goes Wrong

Sometimes even with Pygame’s best efforts it will be unable to give you what you ask for. For

example, if you have run out of memory you will not be able to load any more images and

pygame.image.load will have no space to read an image to. In this case, Pygame will throw a

pygame.error exception. Other situations will also produce errors. Here’s what happens if you

try to set a display mode with a height of 0 pixels:

>>> import pygame

>>> screen = pygame.display.set\_mode((640, 0))

Traceback (most recent call last):

File "<interactive input>", line 1, in ?

pygame.error: Cannot set 0 sized display mode

Generally speaking, there is not a great deal you can do when you get pygame.error excep-

tions that aren’t programming errors. If you can’t set the desired video mode, the game simply

can’t run. Similarly, if you can only load half the image you will not be able to properly display

the visuals in the game. Often the best you can do is apologize to the player and perhaps direct

them somewhere for help.

I have omitted catching Pygame exceptions in the samples to simplify the code a little, but

if you ever produce a game that you would like to distribute, it is good practice to check for

errors. The following snippet will display a message and exit if set\_mode throws an exception:

try:

screen = pygame.display.set\_mode(SCREEN\_SIZE)

except pygame.error, e:

print "Can't create the display :-("

print e

exit()

If you have chosen sensible values for the display, you are unlikely to get these exceptions. A

more likely place they will occur is when reading images. If you try to load an image that you

don’t have on your hard drive, or there is a typo in the file name, you will also get a pygame.error

exception. It is good practice to catch all exceptions that could potentially occur when your game

is distributed. Nobody likes games or applications that fail unexpectedly without any informa-

tion! Some exceptions indicate a bug (programmer error) in the code. It is reasonable to leave

these uncaught, so that you can find problems—and hopefully fix them before the game is

released.

Pygame in Action

Back when I was a youngling, the “scrolly message” was a very popular effect among hobbyist

graphics programmers. A scrolly message, or marquee as it is now known, is simply text sliding

across the screen from right to left. Listing 3-7 is a Pygame implementation of a scrolly mes-

sage. It’s not without its faults, the most major of which is that it will move at an inconsistent

speed, and may be faster or slower on different computers. This is a problem that you will learn

how to solve in the next chapter.

Most of this script should be familiar by now, so I won’t break it down. Try tweaking the

code to produce different results. You may also want to insert text of your own choice, which

you can do by modifying the message string at the start of the script.

Listing 3-7. Scrolly Message Script

background\_image\_filename = 'sushiplate.jpg'

SCREEN\_SIZE = (640, 480)

message="

This is a demonstration of the scrolly message script.

import pygame

from pygame.locals import \*

from sys import exit

pygame.init()

screen = pygame.display.set\_mode(SCREEN\_SIZE)

font = pygame.font.SysFont("arial", 80);

text\_surface = font.render(message, True, (0, 0, 255))

x = 0

y = ( SCREEN\_SIZE[1] - text\_surface.get\_height() ) / 2

background = pygame.image.load(background\_image\_filename).convert()

while True:

for event in pygame.event.get():

if event.type == QUIT:

exit()

screen.blit(background, (0,0))

x-= 2

if x < -text\_surface.get\_width():

x = 0

screen.blit(text\_surface, (x, y))

screen.blit(text\_surface, (x+text\_surface.get\_width(), y))

pygame.display.update()

Summary

Pygame is a powerful platform for building games. It consists of many submodules for a variety

of game-related tasks. Pygame works equally well on a large number of platforms. Ports are

available for all the major desktop systems and even some consoles—so you can develop a

game on your favorite platform and play it on another.

We’ve produced a Hello World script that demonstrates the basics of starting Pygame, cre-

ating a display, receiving events, and then drawing to the screen—steps that you will use when

creating more sophisticated games and demos. If you have done any game programming in C

or C++, you will appreciate the simplicity of the code, especially the one-liner to create a

display.