CHAPTER 29

Project 10: Do-It-Yourself Arcade Game

Welcome to the final project. Now that you've sampled several of Python's many capabilities, it's time to go out with a bang. In this chapter, you learn how to use Pygame, an extension that enables you to write full-fledged, full-screen arcade games in Python. Although easy to use, Pygame is quite powerful and consists of several components that are thoroughly documented in the Pygame documentation (available on the Pygame web site, `http://pygame.org`). This project introduces you to some of the main Pygame concepts, but because this chapter is only meant as a starting point, I've skipped several interesting features, such as sound and video handling. I recommend that you investigate the other features yourself, once you've familiarized yourself with the basics. You might also want to take a look at *Beginning Game Development with Python and Pygame* by Will McGugan (Apress, 2007).

What's the Problem?

So, how do you write a computer game? The basic design process is similar to the one you use when writing any other program, but before you can develop an object model, you need to design the game itself. What are its characters, its setting, and its objectives?

I'll keep things reasonably simple here, so as not to clutter the presentation of the basic Pygame concepts. Feel free to create a much more elaborate game if you like.

The game you'll create will be based on the well-known Monty Python sketch "Self-Defense Against Fresh Fruit." In this sketch, a Sergeant Major (John Cleese) is instructing his soldiers in self-defense techniques against attackers, wielding fresh fruit such as pomegranates, mangoes in syrup, greengages, and bananas. The defense techniques include using a gun, unleashing a tiger, and dropping a 16-ton weight on top of the attacker. In this game, you'll turn things around—the player controls a banana that desperately tries to survive a course in self-defense, avoiding a barrage of 16-ton weights dropping from above. I guess a fitting name for the game might be Squish.

Note If you would like to try your hand at a game of your own as you follow this chapter, feel free to do so. If you just want to change the look and feel of the game, simply replace the graphics (a couple of GIF or PNG images) and some of the descriptive text.

The specific goals of this project revolve around the game design. The game should behave as it was designed (the banana should be movable, and the 16-ton weight should drop from above). In addition, the code should be modular and easily extensible (as always). A useful requirement might be that game states (such as the game introduction, the various game levels, and the "game over" state) should be part of the design, and that new states should be easy to add.

Useful Tools

The only new tool you need in this project is Pygame, which you can download from the Pygame web site (`http://pygame.org`). To get Pygame to work in UNIX, you may need to install some extra software, but it's all documented in the Pygame installation instructions (also available from the Pygame web site). The Windows binary installer is very easy to use—simply execute the installer and follow the instructions.

■ **Note** The Pygame distribution does not include NumPy (`http://numpy.scipy.org`), which may be useful for manipulating sounds and images. Although it's not needed for this project, you might want to check it out. The Pygame documentation thoroughly describes how to use NumPy with Pygame.

The Pygame distribution consists of several modules, most of which you won't need in this project. The following sections describe the modules you do need. (Only the specific functions or classes you'll need are discussed here.) In addition to the functions described in the following sections, the various objects used (such as surfaces, groups, and sprites) have several useful methods, which I'll discuss as they are used in the implementation sections.

■ **Tip** You can find a nice introduction to Pygame in the "Line-by-Line Chimp" tutorial on the Pygame web site (`http://pygame.org/docs/tut/chimp/ChimpLineByLine.html`). It addresses a few issues not discussed here, such as playing sound clips.

---

pygame

The `pygame` module automatically imports all the other Pygame modules, so if you place `import pygame` at the top of your program, you can automatically access the other modules, such as `pygame.display` and `pygame.font`.

The `pygame` module contains (among other things) the `Surface` function, which returns a new surface object. Surface objects are simply blank images of a given size that you can use for

drawing and blitting. To blit (calling a surface object's `blit` method) simply means to transfer the contents of one surface to another. (The word *blit* is derived from the technical term *block transfer*, which is abbreviated BLT.)

The `init` function is central to any Pygame game. It must be called before your game enters its main event loop. This function automatically initializes all the other modules (such as `font` and `image`).

You need the `error` class when you want to catch Pygame-specific errors.

pygame.locals

The `pygame.locals` module contains names (variables) you might want in your own module's scope. It contains names for event types, keys, video modes, and more. It is designed to be safe to use when you import everything (`from pygame.locals import *`), although if you know what you need, you may want to be more specific (for example, `from pygame.locals import FULLSCREEN`).

pygame.display

The `pygame.display` module contains functions for dealing with the Pygame display, which either may be contained in a normal window or occupy the entire screen. In this project, you need the following functions:

`flip`: Updates the display. In general, when you modify the current screen, you do that in two steps. First, you perform all the necessary modifications to the surface object returned from the `get_surface` function, and then you call `pygame.display.flip` to update the display to reflect your changes.

`update`: Used instead of `flip` when you want to update only a part of the screen. It can be used with the list of rectangles returned from the `draw` method of the `RenderUpdates` class (described in the upcoming discussion of the `pygame.sprite` module) as its only parameter.

`set_mode`: Sets the display size and the type of display. Several variations are possible, but here you'll restrict yourself to the `FULLSCREEN` version, and the default "display in a window" version.

`set_caption`: Sets a caption for the Pygame program. The `set_caption` function is primarily useful when you run your game in a window (as opposed to full screen) because the caption is used as the window title.

`get_surface`: Returns a surface object on which you can draw your graphics before calling `pygame.display.flip` or `pygame.display.blit`. The only surface method used for drawing in this project is `blit`, which transfers the graphics found in one surface object onto another one, at a given location. (In addition, the `draw` method of a `Group` object will be used to draw `Sprite` objects onto the display surface.)

pygame.font

The `pygame.font` module contains the `Font` function. Font objects are used to represent different typefaces. They can be used to render text as images that may then be used as normal graphics in Pygame.

pygame.sprite

The `pygame.sprite` module contains two very important classes: `Sprite` and `Group`.

The `Sprite` class is the base class for all visible game objects—in the case of this project, the banana and the 16-ton weight. To implement your own game objects, you subclass `Sprite`, override its constructor to set its `image` and `rect` properties (which determine how the `Sprite` looks and where it is placed), and override its `update` method, which is called whenever the sprite might need updating.

Instances of the `Group` class (and its subclasses) are used as containers for `Sprites`. In general, using groups is A Good Thing. In simple games (such as in this project), just create a group called `sprites` or `allsprites` or something similar, and add all your `Sprites` to it. When you call the `Group` object's `update` method, the `update` methods of all your `Sprite` objects will then be called automatically. Also, the `Group` object's `clear` method is used to erase all the `Sprite` objects it contains (using a callback to do the erasing), and the `draw` method can be used to draw all the `Sprites`.

In this project, you'll use the `RenderUpdates` subclass of `Group`, whose `draw` method returns a list of rectangles that have been affected. These may then be passed to `pygame.display.update` to update only the parts of the display that need to be updated. This can potentially improve the performance of the game quite a bit.

pygame.mouse

In Squish, you'll use the `pygame.mouse` module for just two things: hiding the mouse cursor and getting the mouse position. You hide the mouse with `pygame.mouse.set_visible(False)`, and you get the position with `pygame.mouse.get_pos()`.

pygame.event

The `pygame.event` module keeps track of various events such as mouse clicks, mouse motion, keys that are pressed or released, and so on. To get a list of the most recent events, use the function `pygame.event.get`.

■ **Note** If you rely only on state information such as the mouse position returned by `pygame.mouse.get_pos`, you don't need to use `pygame.event.get`. However, you need to keep the Pygame updated ("in sync"), which you can do by calling the function `pygame.event.pump` regularly.

---

pygame.image

The `pygame.image` module is used to deal with images such as those stored in GIF, PNG, JPEG, and several other file formats. In this project, you need only the `load` function, which reads an image file and creates a surface object containing the image.

Preparations

Now that you know a bit about what some of the different Pygame modules do, it's almost time to start hacking away at the first prototype game. There are, however, a couple of preparations you need to make before you can get the prototype up and running. First of all, you should make sure that you have Pygame installed, including the `image` and `font` modules. (You might want to import both of these in an interactive Python interpreter to make sure they are available.)

You also need a couple of images (for example, from a web site like `http://www.openclipart.org` or found through Google's image search). If you want to stick to the theme of the game as presented in this chapter, you need one image depicting a 16-ton weight and one depicting a banana, both of which are shown in Figure 29-1. Their exact sizes aren't all that important, but you might want to keep them in the range of 100 × 100 through 200 × 200 pixels. You should have these two images available in a common image file format such as GIF, PNG, or JPEG.

■ **Note** You might also want a separate image for the *splash screen,* the first screen that greets the user of your game. In this project, I simply used the weight symbol for that as well.

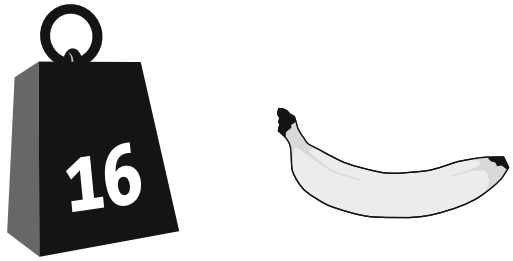

Figure 29-1. *The weight and banana graphics used in my version of the game*

First Implementation

When you use a new tool such as Pygame, it often pays off to keep the first prototype as simple as possible and to focus on learning the basics of the new tool, rather than the intricacies of the

program itself. Let's restrict the first version of Squish to an animation of 16-ton weights falling from above. The steps needed for this are as follows:

1. Initialize Pygame, using `pygame.init`, `pygame.display.set_mode`, and `pygame.mouse.set_visible`. Get the screen surface with `pygame.display.get_surface`. Fill the screen surface with a solid white color (with the `fill` method) and call `pygame.display.flip` to display this change.
2. Load the weight image.
3. Create an instance of a custom `Weight` class (a subclass of `Sprite`) using the image. Add this object to a `RenderUpdates` group called (for example) `sprites`. (This will be particularly useful when dealing with multiple sprites.)
4. Get all recent events with `pygame.event.get`. Check all the events in turn. If an event of type `QUIT` is found, or if an event of type `KEYDOWN` representing the escape key (`K_ESCAPE`) is found, exit the program. (The event types and keys are kept in the attributes `type` and `key` in the event object. Constants such as `QUIT`, `KEYDOWN`, and `K_ESCAPE` can be imported from the module `pygame.locals`.)
5. Call the `clear` and `update` methods of the `sprites` group. The `clear` method uses the callback to clear all the sprites (in this case, the weight), and the `update` method calls the `update` method of the `Weight` instance. (You must implement the latter method yourself.)
6. Call `sprites.draw` with the screen surface as the argument to draw the `Weight` sprite at its current position. (This position changes each time `update` is called.)
7. Call `pygame.display.update` with the rectangle list returned from `sprites.draw` to update the display only in the right places. (If you don't need the performance, you can use `pygame.display.flip` here to update the entire display.)
8. Repeat steps 4 through 7.

See Listing 29-1 for code that implements these steps. The `QUIT` event would occur if the user quit the game—for example, by closing the window.

Listing 29-1. *A Simple "Falling Weights" Animation (weights.py)*

```
import sys, pygame
from pygame.locals import *
from random import randrange

class Weight(pygame.sprite.Sprite):

    def __init__(self):
        pygame.sprite.Sprite.__init__(self)
        # image and rect used when drawing sprite:
        self.image = weight_image
        self.rect = self.image.get_rect()
        self.reset()
```

```
    def reset(self):
        """
        Move the weight to a random position at the top of the screen.
        """
        self.rect.top = -self.rect.height
        self.rect.centerx = randrange(screen_size[0])

    def update(self):
        """
        Update the weight for display in the next frame.
        """
        self.rect.top += 1

        if self.rect.top > screen_size[1]:
            self.reset()

# Initialize things
pygame.init()
screen_size = 800, 600
pygame.display.set_mode(screen_size, FULLSCREEN)
pygame.mouse.set_visible(0)

# Load the weight image
weight_image = pygame.image.load('weight.png')
weight_image = weight_image.convert() # ... to match the display

# Create a sprite group and add a Weight
sprites = pygame.sprite.RenderUpdates()
sprites.add(Weight())

# Get the screen surface and fill it
screen = pygame.display.get_surface()
bg = (255, 255, 255) # White
screen.fill(bg)
pygame.display.flip()

# Used to erase the sprites:
def clear_callback(surf, rect):
    surf.fill(bg, rect)

while True:
    # Check for quit events:
    for event in pygame.event.get():
        if event.type == QUIT:
            sys.exit()
        if event.type == KEYDOWN and event.key == K_ESCAPE:
            sys.exit()
```

```
    # Erase previous positions:
    sprites.clear(screen, clear_callback)
    # Update all sprites:
    sprites.update()
    # Draw all sprites:
    updates = sprites.draw(screen)
    # Update the necessary parts of the display:
    pygame.display.update(updates)
```

You can run this program with the following command:

```
$ python weights.py
```

You should make sure that both `weights.py` and `weight.png` (the weight image) are in the current directory when you execute this command.

■ **Note** I have used a PNG image with transparency here, but a GIF image might work just as well. JPEG images aren't really well suited for transparency.

Figure 29-2 shows a screenshot of the program created in Listing 29-1.
Most of the code should speak for itself. However, a few points need some explanation:

- All sprite objects should have two attributes called `image` and `rect`. The former should contain a surface object (an image), and the latter should contain a rectangle object (just use `self.image.get_rect()` to initialize it). These two attributes will be used when drawing the sprites. By modifying `self.rect`, you can move the sprite around.
- Surface objects have a method called `convert`, which can be used to create a copy with a different color model. You don't need to worry about the details, but using `convert` without any arguments creates a surface that is tailored for the current display, and displaying it will be as fast as possible.
- Colors are specified through RGB triples (red-green-blue, with each value being 0–255), so the tuple `(255, 255, 255)` represents white.

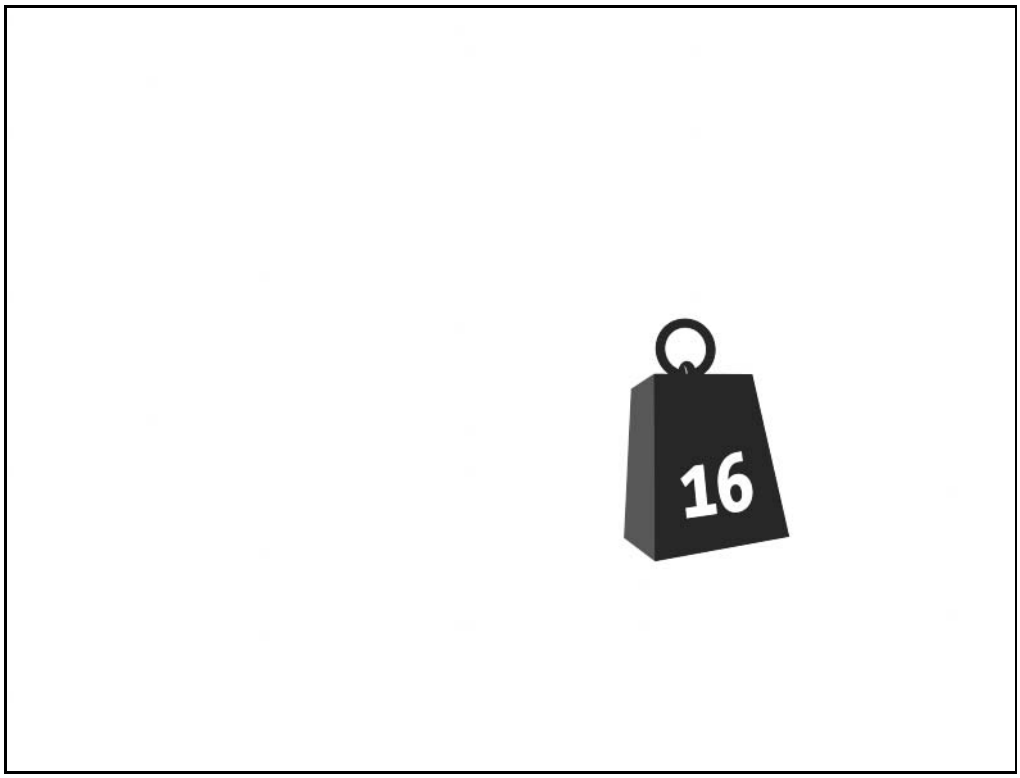

Figure 29-2. *A simple animation of falling weights*

You modify a rectangle (such as `self.rect` in this case) by assigning to its attributes (`top`, `bottom`, `left`, `right`, `topleft`, `topright`, `bottomleft`, `bottomright`, `size`, `width`, `height`, `center`, `centerx`, `centery`, `midleft`, `midright`, `midtop`, and `midbottom`) or calling methods such as `inflate` or `move`. (These are all described in the Pygame documentation at `http://pygame.org/docs/ref/rect.html`.)

Now that the Pygame technicalities are in place, it's time to extend and refactor your game logic a bit.

Second Implementation

In this section, instead of walking you through the design and implementation step by step, I have added copious comments and docstrings to the source code, shown in Listings 29-2 through 29-4. You can examine the source ("use the source," remember?) to see how it works, but here is a short rundown of the essentials (and some not-quite-intuitive particulars):

- The game consists of five files: `config.py`, which contains various configuration variables; `objects.py`, which contains the implementations of the game objects; `squish.py`, which contains the main `Game` class and the various game state classes; and `weight.png` and `banana.png`, the two images used in the game.
- The rectangle method `clamp` ensures that a rectangle is placed within another rectangle, moving it if necessary. This is used to ensure that the banana doesn't move off-screen.
- The rectangle method `inflate` resizes (inflates) a rectangle by a given number of pixels in the horizontal and vertical direction. This is used to shrink the banana boundary, to allow some overlap between the banana and the weight before a hit (or "squish") is registered.
- The game itself consists of a game object and various game states. The game object only has one state at a time, and the state is responsible for handling events and displaying itself on the screen. A state may also tell the game to switch to another state. (A `Level` state may, for example, tell the game to switch to a `GameOver` state.)

That's it. You can run the game by executing the `squish.py` file, as follows:

```
$ python squish.py
```

You should make sure that the other files are in the same directory. In Windows, you can simply double-click the `squish.py` file.

■ **Tip** If you rename `squish.py` to `squish.pyw`, double-clicking it in Windows won't pop up a gratuitous terminal window. If you want to put the game on your desktop (or somewhere else) without moving all the modules and image files along with it, simply create a shortcut to the `squish.pyw` file. See Chapter 18 for details on packaging your game.

Listing 29-2. *The Squish Configuration File (config.py)*

```
# Configuration file for Squish
# -----------------------------

# Feel free to modify the configuration variables below to taste.
# If the game is too fast or too slow, try to modify the speed
# variables.
```

```
# Change these to use other images in the game:
banana_image = 'banana.png'
weight_image = 'weight.png'
splash_image = 'weight.png'

# Change these to affect the general appearance:
screen_size = 800, 600
background_color = 255, 255, 255
margin = 30
full_screen = 1
font_size = 48

# These affect the behavior of the game:
drop_speed = 5
banana_speed = 10
speed_increase = 1
weights_per_level = 10
banana_pad_top = 40
banana_pad_side = 20
```

Listing 29-3. *The Squish Game Objects (objects.py)*

```
import pygame, config, os
from random import randrange

"This module contains the game objects of the Squish game."

class SquishSprite(pygame.sprite.Sprite):

    """
    Generic superclass for all sprites in Squish.  The constructor
    takes care of loading an image, setting up the sprite rect, and
    the area within which it is allowed to move. That area is governed
    by the screen size and the margin.
    """

    def __init__(self, image):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.image.load(image).convert()
        self.rect = self.image.get_rect()
        screen = pygame.display.get_surface()
        shrink = -config.margin * 2
        self.area = screen.get_rect().inflate(shrink, shrink)
```

```
class Weight(SquishSprite):

    """
    A falling weight. It uses the SquishSprite constructor to set up
    its weight image, and will fall with a speed given as a parameter
    to its constructor.
    """

    def __init__(self, speed):
        SquishSprite.__init__(self, config.weight_image)
        self.speed = speed
        self.reset()

    def reset(self):
        """
        Move the weight to the top of the screen (just out of sight)
        and place it at a random horizontal position.
        """
        x = randrange(self.area.left, self.area.right)
        self.rect.midbottom = x, 0

    def update(self):
        """
        Move the weight vertically (downwards) a distance
        corresponding to its speed. Also set the landed attribute
        according to whether it has reached the bottom of the screen.
        """
        self.rect.top += self.speed
        self.landed = self.rect.top >= self.area.bottom


class Banana(SquishSprite):

    """
    A desperate banana. It uses the SquishSprite constructor to set up
    its banana image, and will stay near the bottom of the screen,
    with its horizontal position governed by the current mouse
    position (within certain limits).
    """

    def __init__(self):
        SquishSprite.__init__(self, config.banana_image)
        self.rect.bottom = self.area.bottom
```

```
        # These paddings represent parts of the image where there is
        # no banana. If a weight moves into these areas, it doesn't
        # constitute a hit (or, rather, a squish):
        self.pad_top = config.banana_pad_top
        self.pad_side = config.banana_pad_side

    def update(self):
        """
        Set the Banana's center x-coordinate to the current mouse
        x-coordinate, and then use the rect method clamp to ensure
        that the Banana stays within its allowed range of motion.
        """
        self.rect.centerx = pygame.mouse.get_pos()[0]
        self.rect = self.rect.clamp(self.area)

    def touches(self, other):
        """
        Determines whether the banana touches another sprite (e.g., a
        Weight). Instead of just using the rect method colliderect, a
        new rectangle is first calculated (using the rect method
        inflate with the side and top paddings) that does not include
        the 'empty' areas on the top and sides of the banana.
        """
        # Deflate the bounds with the proper padding:
        bounds = self.rect.inflate(-self.pad_side, -self.pad_top)
        # Move the bounds so they are placed at the bottom of the Banana:
        bounds.bottom = self.rect.bottom
        # Check whether the bounds intersect with the other object's rect:
        return bounds.colliderect(other.rect)
```

Listing 29-4. *The Main Game Module (squish.py)*

```
import os, sys, pygame
from pygame.locals import *
import objects, config

"This module contains the main game logic of the Squish game."

class State:

    """
    A generic game state class that can handle events and display
    itself on a given surface.
    """
```

```
    def handle(self, event):
        """
        Default event handling only deals with quitting.
        """
        if event.type == QUIT:
            sys.exit()
        if event.type == KEYDOWN and event.key == K_ESCAPE:
            sys.exit()

    def firstDisplay(self, screen):
        """
        Used to display the State for the first time. Fills the screen
        with the background color.
        """
        screen.fill(config.background_color)
        # Remember to call flip, to make the changes visible:
        pygame.display.flip()

    def display(self, screen):
        """
        Used to display the State after it has already been displayed
        once. The default behavior is to do nothing.
        """
        pass


class Level(State):

    """
    A game level. Takes care of counting how many weights have been
    dropped, moving the sprites around, and other tasks relating to
    game logic.
    """

    def __init__(self, number=1):
        self.number = number
        # How many weights remain to dodge in this level?
        self.remaining = config.weights_per_level

        speed = config.drop_speed
        # One speed_increase added for each level above 1:
        speed += (self.number-1) * config.speed_increase
```

```
        # Create the weight and banana:
        self.weight = objects.Weight(speed)
        self.banana = objects.Banana()
        both = self.weight, self.banana # This could contain more sprites...
        self.sprites = pygame.sprite.RenderUpdates(both)

    def update(self, game):
        "Updates the game state from the previous frame."
        # Update all sprites:
        self.sprites.update()
        # If the banana touches the weight, tell the game to switch to
        # a GameOver state:
        if self.banana.touches(self.weight):
            game.nextState = GameOver()
        # Otherwise, if the weight has landed, reset it. If all the
        # weights of this level have been dodged, tell the game to
        # switch to a LevelCleared state:
        elif self.weight.landed:
            self.weight.reset()
            self.remaining -= 1
            if self.remaining == 0:
                game.nextState = LevelCleared(self.number)

    def display(self, screen):
        """
        Displays the state after the first display (which simply wipes
        the screen). As opposed to firstDisplay, this method uses
        pygame.display.update with a list of rectangles that need to
        be updated, supplied from self.sprites.draw.
        """
        screen.fill(config.background_color)
        updates = self.sprites.draw(screen)
        pygame.display.update(updates)


class Paused(State):

    """
    A simple, paused game state, which may be broken out of by pressing
    either a keyboard key or the mouse button.
    """

    finished = 0  # Has the user ended the pause?
    image = None  # Set this to a file name if you want an image
    text = ''     # Set this to some informative text
```

```
    def handle(self, event):
        """
        Handles events by delegating to State (which handles quitting
        in general) and by reacting to key presses and mouse
        clicks. If a key is pressed or the mouse is clicked,
        self.finished is set to true.
        """
        State.handle(self, event)
        if event.type in [MOUSEBUTTONDOWN, KEYDOWN]:
            self.finished = 1

    def update(self, game):
        """
        Update the level. If a key has been pressed or the mouse has
        been clicked (i.e., self.finished is true), tell the game to
        move to the state represented by self.nextState() (should be
        implemented by subclasses).
        """
        if self.finished:
            game.nextState = self.nextState()

    def firstDisplay(self, screen):
        """
        The first time the Paused state is displayed, draw the image
        (if any) and render the text.
        """
        # First, clear the screen by filling it with the background color:
        screen.fill(config.background_color)

        # Create a Font object with the default appearance, and specified size:
        font = pygame.font.Font(None, config.font_size)

        # Get the lines of text in self.text, ignoring empty lines at
        # the top or bottom:
        lines = self.text.strip().splitlines()

        # Calculate the height of the text (using font.get_linesize()
        # to get the height of each line of text):
        height = len(lines) * font.get_linesize()

        # Calculate the placement of the text (centered on the screen):
        center, top = screen.get_rect().center
        top -= height // 2
```

```
        # If there is an image to display...
        if self.image:
            # load it:
            image = pygame.image.load(self.image).convert()
            # get its rect:
            r = image.get_rect()
            # move the text down by half the image height:
            top += r.height // 2
            # place the image 20 pixels above the text:
            r.midbottom = center, top - 20
            # blit the image to the screen:
            screen.blit(image, r)

        antialias = 1    # Smooth the text
        black = 0, 0, 0  # Render it as black

        # Render all the lines, starting at the calculated top, and
        # move down font.get_linesize() pixels for each line:
        for line in lines:
            text = font.render(line.strip(), antialias, black)
            r = text.get_rect()
            r.midtop = center, top
            screen.blit(text, r)
            top += font.get_linesize()

        # Display all the changes:
        pygame.display.flip()


class Info(Paused):

    """
    A simple paused state that displays some information about the
    game. It is followed by a Level state (the first level).
    """

    nextState = Level
    text = '''
    In this game you are a banana,
    trying to survive a course in
    self-defense against fruit, where the
    participants will "defend" themselves
    against you with a 16 ton weight.'''
```

```
class StartUp(Paused):

    """
    A paused state that displays a splash image and a welcome
    message. It is followed by an Info state.
    """

    nextState = Info
    image = config.splash_image
    text = '''
    Welcome to Squish,
    the game of Fruit Self-Defense'''


class LevelCleared(Paused):

    """
    A paused state that informs the user that he or she has cleared a
    given level.  It is followed by the next level state.
    """

    def __init__(self, number):
        self.number = number
        self.text = '''Level %i cleared
        Click to start next level''' % self.number


    def nextState(self):
        return Level(self.number+1)


class GameOver(Paused):

    """
    A state that informs the user that he or she has lost the
    game. It is followed by the first level.
    """

    nextState = Level
    text = '''
    Game Over
    Click to Restart, Esc to Quit'''
```

```
class Game:

    """
    A game object that takes care of the main event loop, including
    changing between the different game states.
    """

    def __init__(self, *args):
        # Get the directory where the game and the images are located:
        path = os.path.abspath(args[0])
        dir = os.path.split(path)[0]
        # Move to that directory (so that the image files may be
        # opened later on):
        os.chdir(dir)
        # Start with no state:
        self.state = None
        # Move to StartUp in the first event loop iteration:
        self.nextState = StartUp()

    def run(self):
        """
        This method sets things in motion. It performs some vital
        initialization tasks, and enters the main event loop.
        """
        pygame.init() # This is needed to initialize all the pygame modules

        # Decide whether to display the game in a window or to use the
        # full screen:
        flag = 0                  # Default (window) mode

        if config.full_screen:
            flag = FULLSCREEN      # Full screen mode
        screen_size = config.screen_size
        screen = pygame.display.set_mode(screen_size, flag)

        pygame.display.set_caption('Fruit Self Defense')
        pygame.mouse.set_visible(False)

        # The main loop:
        while True:
            # (1) If nextState has been changed, move to the new state, and
            #     display it (for the first time):
            if self.state != self.nextState:
                self.state = self.nextState
                self.state.firstDisplay(screen)
```

```
            # (2) Delegate the event handling to the current state:
            for event in pygame.event.get():
                self.state.handle(event)
            # (3) Update the current state:
            self.state.update(self)
            # (4) Display the current state:
            self.state.display(screen)


if __name__ == '__main__':
    game = Game(*sys.argv)
    game.run()
```

Some screenshots of the game are shown in Figures 29-3 through 29-6.

Figure 29-3. *The Squish opening screen*

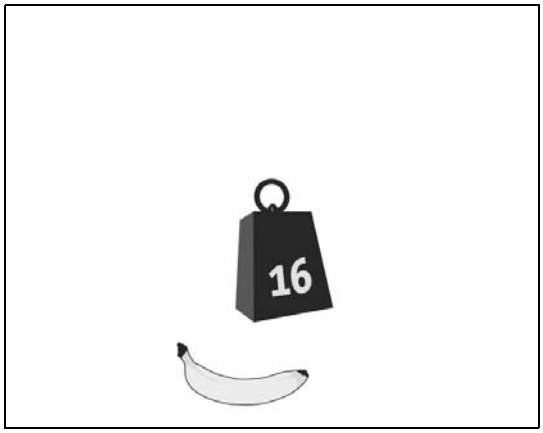

Figure 29-4. *A banana about to be squished*

Figure 29-5. *The "level cleared" screen*

Figure 29-6. *The "game over" screen*

Further Exploration

Here are some ideas for how you can improve the game:

- Add sounds to it.
- Keep track of the score. Each weight dodged could be worth 16 points, for example. How about keeping a high-score file? Or even an online high-score server (using `asyncore` or XML-RPC, as discussed in Chapters 24 and 27, respectively)?
- Make more objects fall simultaneously.
- Give the player more than one "life."
- Create a stand-alone executable of your game (using `py2exe`, for example) and package it with an installer. (See Chapter 18 for details.)

For a much more elaborate (and extremely entertaining) example of Pygame programming, check out the SolarWolf game by Pete Shinners, the Pygame maintainer (`http://www.pygame.org/shredwheat/solarwolf`). You can find plenty of information and several other games at the Pygame web site. If playing with Pygame gets you hooked on game development, you might want to check out web sites like `http://www.gamedev.net` and `http://www.flipcode.com`. A web search should give you plenty of other similar sites.

What Now?

Well, this is it. You have finished the last project. If you take stock of what you have accomplished (assuming that you have followed all the projects), you should be rightfully impressed with yourself. The breadth of the topics presented has given you a taste of the possibilities that await you in the world of Python programming. I hope you have enjoyed the trip this far, and I wish you good luck on your continued journey as a Python programmer.