

16

Create games with pygame



What you will learn

Writing games is great fun. Unlike “proper” programs, games are not always tied to a formal specification and don’t need to do anything useful. They just must be fun to play. Games are a great place to experiment with your software. You can write code just to see what happens when it runs, and see whether the result is interesting. Everyone should write at least one computer game in their lives. In this chapter, you’ll start creating games. You’ll learn how to make a complete game and finish with a framework you can use to create more games of your own design.

Getting started with pygame	594
Draw images with pygame.....	601
Get user input from pygame.....	606
Create game sprites	609
Complete the game	629
What you have learned	636



Getting started with pygame

In this section, we'll get started with pygame, and we'll create some shapes and display them on the screen. The free pygame library contains lots of Python classes you can use to create games. The snaps functions we used in the early chapters of this book were written using pygame, so you should have already loaded pygame onto your computer (see Chapter 3 for instructions).

Note that the pygame library makes use of tuples to create single-data items that contain colors and coordinates that describe items in the games. If you're not sure what a tuple is, read the description of tuples in Chapter 8 before you work through the following "Make Something Happen."



MAKE SOMETHING HAPPEN

Start pygame and draw some lines

The best way to understand how pygame works is to start it up and draw something. Open the Python Command Shell in IDLE to get started. Before we can use pygame in a program, we need to import it; enter the statement below and press **Enter**:

```
>>> import pygame
```

Once we've imported the pygame module, we can start using the functions and classes it contains. The pygame framework needs to be set up before you can use it to display the items in your game. A game program does this by calling the `init` function in the pygame module, as shown here:

```
>>> pygame.init()
```

When you press Enter, the `init` function sets up the different pygame elements, each of which performs a specific task when the game is running. Elements read user input, make sounds, and so on. The `init` function returns a tuple that tells you how many elements have been successfully initialized, and how many have failed to initialize. If an element fails to initialize, pygame might not have been installed correctly. However, most games ignore this value and assume that all is well.

```
>>> pygame.init()  
(6, 0)
```

The display above shows that six modules have been set up correctly and that none have failed to initialize. If you see any failures—in other words, if the second value in the tuple is any value other than zero—you should make sure that pygame has been properly installed.

Next, we need to create a drawing surface. A drawing surface has a specific size, which is set when we create it. The size is given in pixels (a pixel is the size of a dot on the display). The more pixels you have, the better quality the display. You also find pixel dimensions when talking about camera and video screen resolution. We'll use a screen size of 800 pixels wide and 600 pixels high. We can use a tuple to create a surface as follows:

```
>>> size = (800, 600)
```

Remember that a tuple is a way of grouping a number of items. You can find out more about them in Chapter 8. Once we have the tuple that describes the size of the game screen, we can use this value as an argument to the function that creates a pygame drawing surface.

```
>>> surface = pygame.display.set_mode(size)
```

This statement creates the drawing surface, sets the variable `surface` to refer to it, and then displays the surface on the screen. You should see the window below appear on your screen.



You can change the title of the drawing window using the following function:

```
>>> pygame.display.set_caption('An awesome game by Rob')
```

This function changes the title of the window as shown below.



Now we can draw things on the surface, so we'll start by drawing some lines. The line drawing function in pygame accepts four parameters:

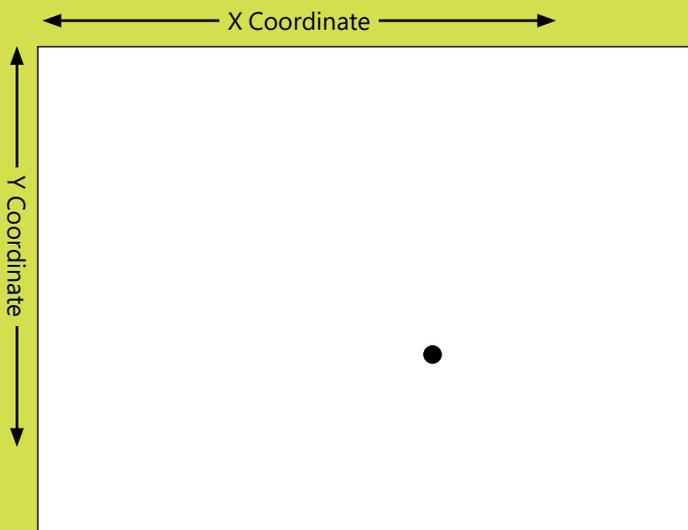
- The surface on which to draw
- The drawing color
- The start position of the line
- The end position of the line

Let's assemble these items. We've already created the surface, so we can just use that. The color of an item in pygame is expressed as a tuple containing three values. We first saw this mechanism for expressing color in Chapter 3 when we used the snaps framework to draw text. Each value in the tuple represents the amount of red, green, and blue, respectively. The lowest level is 0; the highest level is 255. If we want to draw a red line, we can create a tuple that contains all the red and none of the other two primary colors. Enter the following tuple:

```
>>> red = (255, 0, 0)
```

Now we can set the start position of the line. For a given position on the screen, the value of *x* specifies how far the position is from the left edge, and the value of *y* specifies how far down the screen from the top edge. A specific location is expressed as a tuple containing the values (*x*, *y*). The figure below shows how pygame coordinates work. The important thing to remember is that the *origin*, which is the point with the coordinate (0,0) is the top left corner of the display. Increasing the value of *x* moves you toward the right of the screen, and increasing the value of *y* will move you down the screen.

This might not be how you expect graphics to work. Most graphs that you draw have their origins in the bottom left, and increasing *y* moves up. However, placing the origin in the top left corner is standard practice when drawing graphics on a computer.



Bearing this in mind, let's draw a line from the origin on the screen to the position (500,300). We can create some tuples that hold these values. Type in these two statements to set the start and end position of the line.

```
>>> start = (0,0)
>>> end = (500, 300)
```

Now we can issue our drawing instruction. Type in the following call to the `line` function in the `pygame.draw` module:

```
>>> pygame.draw.line(surface, red, start, end)
```

When you press **Enter**, the line is drawn, and the line function returns a rectangle object that encloses this line:

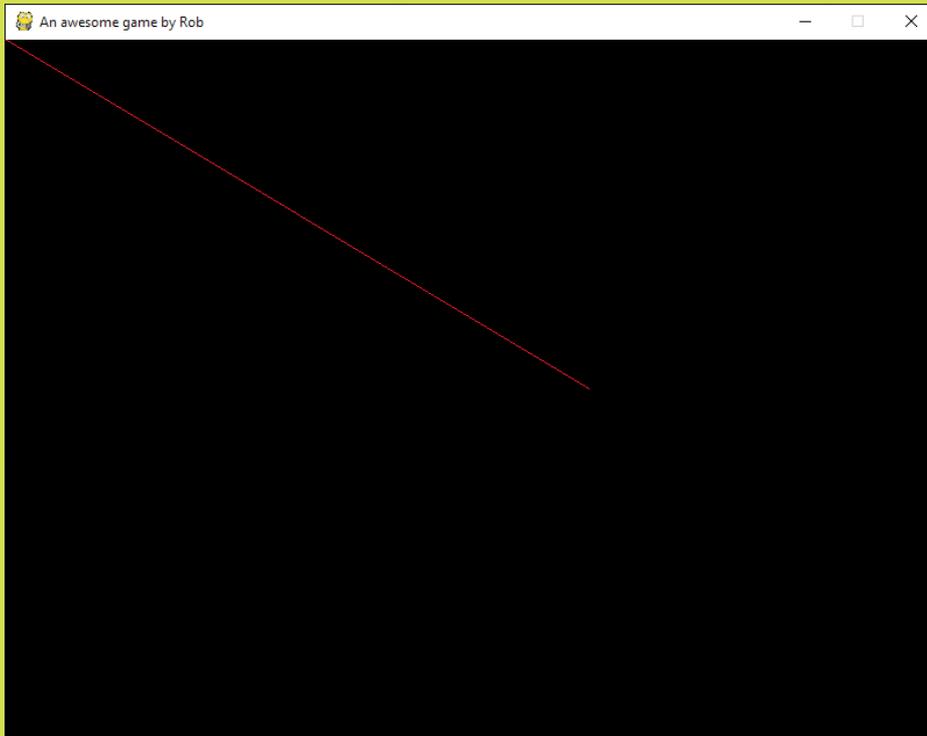
```
>>> pygame.draw.line(surface, red, start, end)
<rect(0, 0, 501, 301)>
```

We'll ignore the values returned from the drawing methods. Unfortunately, if you look at the game window, you won't see any lines on the screen. Draw operations take place on the *back buffer* managed by `pygame`. We don't draw directly on the screen because we don't want the player to see each individual draw action. Instead, we perform all our drawing operations on a piece of memory in the computer (called the back buffer). When the drawing is finished, we copy this piece of memory onto the display memory. The memory that used to be displayed becomes the new back buffer, and the process starts again.

In `pygame`, the `flip` function swaps the display memory and the back-buffer memory. We need to call `flip` to make a line appear on the screen, so type the call below and press **Enter**.

```
>>> pygame.display.flip()
```

This call will cause a red line to appear on the game display, as shown on the next page.



If you don't want a black background, you can use the `fill` function to fill the screen with a chosen color. These three statements create a tuple that describes the color white, fills the back buffer with white, and then flips the back buffer to display the white screen.

```
>>> white = (255, 255, 255)
>>> surface.fill(white)
>>> pygame.display.flip()
```

If you do this, you'll notice that the red line we created has been erased.

We can use these functions to create some nice-looking images. The program below draws 100 colored lines and 100 colored dots. The program uses functions that create random colors and positions on the display area.

```
#EG 16.01 pygame drawing functions
```

```
import random
```

The demo uses random numbers

```

import pygame The demo uses pygame

class DrawDemo: Class to contain our demo program

    @staticmethod Make the method static since we should need to create a demo class
    def do_draw_demo(): Method to demonstrate pygame drawing
        init_result = pygame.init() Initialize pygame
        if init_result[1] != 0: If the number of failures is not zero, we have a problem
            print('pygame not installed properly') Display a message
            return Abandon the demonstration

        width = 800 Set the width of the screen
        height = 600 Set the height of the screen
        size = (width, height) Set the size of the game display

        def get_random_coordinate(): Function to get a random coordinate
            X = random.randint(0, width-1) Get a random X value
            Y = random.randint(0, height-1) Get a random Y value
            return (X, Y) Return a tuple made from X and Y

        def get_random_color(): Function to get a random color
            red = random.randint(0, 255) Get a random red value
            green = random.randint(0, 255) Get a random green value
            blue = random.randint(0, 255) Get a random blue value
            return (red, green, blue) Return a tuple made from red, green, and blue

        surface = pygame.display.set_mode(size) Create the game surface
        pygame.display.set_caption('Drawing example') Set the window caption

        red = (255, 0, 0)
        green = (0, 255, 0)
        blue = (0, 0, 255)
        black = (0, 0, 0)
        yellow = (255, 255, 0)
        magenta = (255, 0, 255)
        cyan = (0, 255, 255)
        white = (255, 255, 255)
        gray = (128, 128, 128) Create some color tuples

        # Fill the screen with white
        surface.fill(white)

        # Draw 100 random lines
        for count in range(100):

```

```
start = get_random_coordinate()
end = get_random_coordinate()
color = get_random_color()
pygame.draw.line(surface, color, start, end)

# Draw 100 dots
dot_radius = 10
for count in range(100):
    pos = get_random_coordinate()
    color = get_random_color()
    radius = random.randint(5, 50)
    pygame.draw.circle(surface, color, pos, radius)

pygame.display.flip()
```

Flip the drawn elements to the display memory

```
DrawDemo.do_draw_demo()
```

Call the `do_draw_demo` method in the `DrawDemo` object

When I ran the above program, the display appeared as shown in **Figure 16-1**:

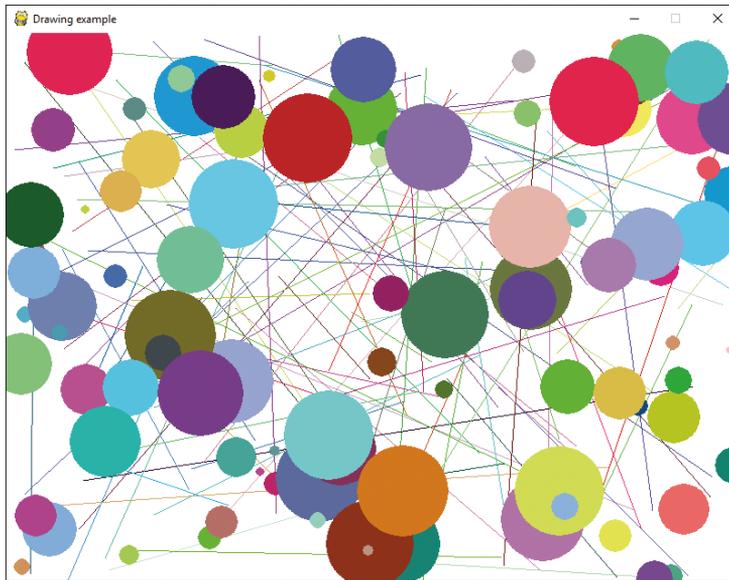


Figure 16-1 Drawing dots and lines

When you run the program, you'll get an image that looks similar but will have a completely different arrangement of lines and circles because your program will get a different sequence of random numbers from the ones produced when I ran the program.



Making art

You could create a program that displays a different pattern every now and then. You could use the time of day and the current weather conditions to determine what colors to use in the pattern and create a display that changes throughout the day (perhaps with bright primary colors in the morning and more mellow and darker colors in the evening). If the weather is warm, the colors could have a red tinge, and if it's colder, you could create colors with more blues. Remember that you can create any color you like for your graphics by choosing the amount of red, green, and blue it should contain.

Draw images with pygame

Pygame can also draw images on the screen. The images are loaded from files stored on your computer. You've already used the `display_image` function from the `snaps` library to draw images; now you'll discover how to use `pygame` to load and display images.

Image file types

There are a number of different formats for storing pictures on computers. When working with Pygame, your pictures should be in one of these two formats:

- PNG—The PNG format is *lossless*, meaning it always stores an exact version of the image. PNG files can also have transparent regions, which is important when you want to draw one image on top of another.
- JPEG—The JPEG format is *lossy*, meaning the image is compressed in a way that makes it much smaller, but at the expense of precise detail.

The games you create should use JPEG images for the large backgrounds and PNG images for smaller objects drawn on top of them.

If you have no usable pictures of your own, you can use the ones I've provided with the sample files for this chapter, but the games will work best if you use your own pictures.

Figure 16-2 shows my picture of the cheese we'll be using in the game that we will create. In the game, the player will control the cheese and use it to catch crackers around the screen. You can use another picture if you wish. In fact, I strongly advise that you do. I've saved the image in the PNG file format with a width of 50 pixels, which will work with the size of the screen we're using.



Figure 16-2 The cheese

If you need to convert images into the PNG format, you can load an image using the Microsoft Paint program and then save it in this format. With Paint, you can also scale and crop images if you want to reduce the number of pixels in the image. For more advanced image manipulation, I recommend the program Paint.Net, which is free here: www.getpaint.net. Another great image manipulation program is Gimp, which is available for most machines. You can download Gimp from www.gimp.org.

Load an image into a game

The pygame library contains a function called `load` that loads an image. The image to be loaded is identified by its file name. The `load` function searches the local folder for the file. In other words, it looks in the folder from which the program is running. We saw this behavior in Chapter 8 when we wrote programs to store and load data using files. The statement below loads an image from a file. The variable `cheeseImage` is set to refer to the image that's been loaded.

```
cheeseImage = pygame.image.load('cheese.png')
```

Now that we have an image loaded, we can draw it on the display. When an image is drawn, the data that describes the image is copied into the memory used for the display. Game developers call this *blitting* the graphics data onto the screen. The pygame library contains a method called `blit` that's used to copy an image into display memory. The `blit` method requires two pieces of information to work:

- The image to be drawn
- The coordinates on the screen where the image is to be blitted

Let's put our cheese image at the top left corner of the display. The statement below creates a tuple that describes this position. The values of the x and y coordinates are both zero.

```
cheesePos = (0,0)
```

We can now call the `blit` method to actually draw the cheese. The `blit` method is provided by the display surface that we created when our game program started.

```
surface.blit(cheeseImage, cheesePos)
```

The complete program that draws the cheese on the screen can be found below:

```
# EG16-02 Image Drawing

import pygame

class ImageDemo:

    @staticmethod
    def do_image_demo():
        init_result = pygame.init()
        if init_result[1] != 0:
            print('pygame not installed properly')
            return

        width = 800
        height = 600
        size = (width, height)

        surface = pygame.display.set_mode(size)

        pygame.display.set_caption('Image example')

        white = (255, 255, 255)
        surface.fill(white)

        cheeseImage = pygame.image.load('cheese.png')
        cheesePos = (0,0)
        surface.blit(cheeseImage, cheesePos)
        pygame.display.flip()

ImageDemo.do_image_demo()
```

Initialize pygame

End the method if pygame fails to start

Set the size of the display

Get the pygame drawing surface

Sets up the pygame display

Clear the screen to white

Load the cheese image

Set the cheese position to the top left corner of the screen

Draw the cheese

Flip the display memory so that the cheese is displayed

When we run this program, it draws some cheese on the screen as shown in **Figure 16-3**. Note that the drawing position for an image when we blit it onto the screen is the top left corner of that image.



Figure 16-3 Cheese on the screen

Make an image move

The `blit` function is given the draw position for an image. We can make an image appear to move by repeatedly drawing the image at different positions.

```
# EG16-03 Moving cheese

cheeseX = 40
cheeseY = 60

clock = pygame.time.Clock()

for i in range(1,100):
    clock.tick(30)
    surface.fill((255,255,255))
    cheeseX = cheeseX + 1
    cheeseY = cheeseY + 1
    cheesePos = (cheeseX,cheeseY)
    surface.blit(cheeseImage, cheesePos)
pygame.display.flip()
```

- Set the start position for the cheese
- Create a pygame clock instance
- Move the cheese 100 times
- Pause the game so that we have 30 frames per second
- Fill the screen with white
- Increase the x position of the cheese
- Increase the y position of the cheese
- Create a cheese position tuple
- Blit the cheese onto the screen
- Flip to the back buffer to update the display



Move an image

We can investigate the way that games make objects appear to move by using the **EG16-03 Moving cheese** program. When you use IDLE to run it, you should find that the cheese moves majestically down the screen for a while and then stops. The speed of the movement is controlled by the *frame rate* of the game. The frame rate is the rate at which the screen is redrawn, expressed as the number of frames per second (fps). The pygame **CLock** class provides a tick method that is given the number of frames per second required by the game. The program creates a new clock before it starts moving the cheese around.

```
clock = pygame.time.Clock()
```

The **CLock** class provides a set of time management methods that games can use. We'll use the **tick** method that allows us to make the game run at a constant speed. Without the clock, our game would run as fast as Python can execute the program, which would be impossible to play.

```
clock().tick(30)
```

The **tick** method will pause the game until the start of the next frame “slot.” Find the above statement in the program and change the value from 30 to 60. The program will now update the screen 60 times per second. Run the program, and you'll find that the cheese moves twice as fast as it did before because the tick method is now allowing 60 frames per second.

If you change the frame rate to 5 (5 frames per second), you'll find that the cheese moves slowly and you'll be able to see each movement.

A player will get a good game experience if the game updates at 60 frames per second. Games on smaller devices—for example, mobile phones and tablets—might use lower frame rates to save battery power.

Get user input from pygame

Now that we can move items around the screen under program control, the next thing we need is a way that a player can interact with the game. A game receives input from the user by means of pygame *events*. An event is a user action—for example, pressing a keyboard key or moving the mouse. We first saw these kinds of events when we created a graphical user interface using Tkinter in Chapter 13. When we wanted to receive events in Tkinter, we bound a method to an event. When the event occurred, the method was called.

In pygame, events are managed differently. While a pygame program is running, the pygame system captures input events and places them in a queue. The game program must check the event queue regularly to see if there are any actions to which the program must respond. The events we're interested in are keyboard events generated when a key is pressed or released.



MAKE SOMETHING HAPPEN

Investigate events in pygame

We can look at how events work in pygame by creating some events and seeing the results. Open the Python IDLE Command Shell and type in the following statements to create a pygame window:

```
>>> import pygame
>>> pygame.init()
(6, 0)
>>> size = (800, 600)
>>> surface = pygame.display.set_mode(size)
```

Now use your mouse to click in the window that pygame has opened and press a few keys. Each key press will generate an event that will be captured by pygame. Now we can create a loop to look at the events that have been stored. Go back to IDLE and enter the following:

```
>>> for e in pygame.event.get():
    print(e)
```

The `get` method returns a collection of events. This loop will print all the events in the pygame event queue. When you enter an empty line after the `print` statement, you'll see all the event information:

```

>>> for e in pygame.event.get():
        print(e)

<Event(17-VideoExpose {})>
<Event(16-VideoResize {'size': (800, 600), 'w': 800, 'h': 600})>
<Event(1-ActiveEvent {'gain': 0, 'state': 1})>
<Event(2-KeyDown {'unicode': 'r', 'key': 114, 'mod': 0, 'scancode': 19})>
<Event(3-KeyUp {'key': 114, 'mod': 0, 'scancode': 19})>
<Event(2-KeyDown {'unicode': 'o', 'key': 111, 'mod': 0, 'scancode': 24})>
<Event(3-KeyUp {'key': 111, 'mod': 0, 'scancode': 24})>
<Event(2-KeyDown {'unicode': 'b', 'key': 98, 'mod': 0, 'scancode': 48})>
<Event(3-KeyUp {'key': 98, 'mod': 0, 'scancode': 48})>
<Event(1-ActiveEvent {'gain': 1, 'state': 1})>
>>>

```

Each event is described by a dictionary that holds information about the event. If you look through the events above, you'll see that the R, O, and B keys have been pressed and released in turn.

As the game runs, the event queue must be checked to see if any commands have been entered that should cause objects on the screen to move. We want the cheese to move while an arrow key is held down and then stop moving when the key is released. The code below does this. Also, this code contains a test that causes the game to end when the player presses the Escape (Esc) key.

```

# EG16-04 Steerable cheese

cheeseX = 40
cheeseY = 60
cheeseYSpeed = 2
cheeseMovingUp = False
cheeseMovingDown = False
clock = pygame.time.Clock()
while True:
    clock().tick(60)
    for e in pygame.event.get():
        if e.type == pygame.KEYDOWN:
            if e.key == pygame.K_ESCAPE:
                pygame.quit()
                return

```

	Set the cheese's initial position
	Set the speed of the cheese movement
	Cheese is not moving up
	Cheese is not moving down
	Create a clock
	Repeatedly perform the game loop
	Wait for the next frame start
	Work through the events
	Does the event describe a key-down event?
	Is the key the Escape key?
	Shut down pygame
	If Escape has been pressed, exit the game loop

<code>elif e.key == pygame.K_UP:</code>	Is the key the Up arrow?
<code> cheeseMovingUp = True</code>	Set the flag that indicates the cheese is moving up
<code>elif e.key == pygame.K_DOWN:</code>	Is the key the Down arrow?
<code> cheeseMovingDown = True</code>	Set the flag that indicates the cheese is moving down
<code>elif e.type == pygame.KEYUP:</code>	Does the event describe a key up event?
<code> if e.key == pygame.K_UP:</code>	Is the key the Up arrow?
<code> cheeseMovingUp = False</code>	Clear the flag that indicates the cheese is moving up
<code> elif e.key == pygame.K_DOWN:</code>	Is the key the Down arrow?
<code> cheeseMovingDown = False</code>	
<code>if cheeseMovingDown:</code>	Is the cheese moving down?
<code> cheeseY = cheeseY+cheeseYSpeed</code>	Move the cheese down the screen
<code>if cheeseMovingUp:</code>	Is the cheese moving up?
<code> cheeseY = cheeseY-cheeseYSpeed</code>	Move the cheese up
	Clear the flag that indicates the cheese is moving down



CODE ANALYSIS

Game loops

The code above is an example of a “game loop.” You may have some questions about it.

Question: What is the variable `e` used for in the program?

Answer: The variable `e` contains each event that the game loop is checking. The game is interested only in events generated when a key is pressed or released. When a key press is detected, the program checks to see which key was pressed. If the key is the Up Arrow, the code sets the flag to indicate that the cheese should move up; if the key is the Down Arrow, the code sets the flag to indicate that the cheese should move down. The game loop also contains tests that will clear the flag if a key is released.

Question: Why does the cheese move when I hold a key down?

Answer: Remember that the statements in the game loop are being repeated 60 times a second. So, every sixtieth of a second, the program is updating the position of the cheese. If a key is down, the cheese will be moved each time around the game loop. Currently, the `cheeseYspeed` is 2, which means that in a second the cheese will move 120 pixels.

Question: How do we change the speed of the cheese?

Answer: The variable `cheeseYspeed` gives the speed of the cheese in the y direction (up and down the screen). If we want to make the cheese move faster, we can increase the value of this variable.

Question: Why do we increase the value of `y` to move the cheese down the screen?

Answer: This is because the coordinate system used by `pygame` places the origin (the point where the values of `x` and `Y` are zero) at the top of the screen. Increasing the value of `y` will move the cheese down the screen.

Question: What would happen if the player pressed both the Up and the Down Arrow keys at the same time?

Answer: The cheese would be moved both up and then down again when it was updated. The result of this would be that the cheese would not appear to move, which is what we want the game to do.

Question: What would happen if the player moved the cheese right off the screen?

Answer: You can run the sample program to find out what happens. Drawing an image off the screen will not cause the game program to fail, but the object will not be visible. If we want to stop the cheese from moving off the screen, we will need to add code to make sure that the cheese is never positioned off the screen.

Question: What does the `pygame.quit()` method do?

Answer: The `pygame.quit()` method is called when the user presses the Escape key to finish a game; it closes `pygame` and causes the game window to be closed.

Create game sprites

The game we'll create will display three different object types on the screen:

- **Cheese**—The player will steer the cheese around the screen.
- **Crackers**—The player will try to capture the cheese on the cracker.
- **Killer tomato**—The tomato will chase the cheese.

Each of these screen objects is called a *sprite*. You can think of a sprite as an image that is part of the game display. We will create a `Sprite` class that has an image drawn on the screen, a position on the screen, and a set of behaviors. Each sprite will do the following things:

- Draw itself on the screen.
- Update itself. If the sprite is the cheese, it will move in response to player input; if the sprite is the killer tomato, it will chase the cheese.
- Reset itself. When we start a new game, we must put the sprite in its starting position.

Sprites might have other behaviors, too, but these are the fundamental things that a sprite must do. We can put these behaviors into a class:

```
class Sprite:
    """
    A sprite in the game. Can be subclassed
    to create sprites with particular behaviors
    """
    def __init__(self, image, game):
        """
        Initialize a sprite
        image is the image to use to draw the sprite
        default position is origin (0,0)
        game is the game that contains this sprite
        """
        self.image = image
        self.position = [0, 0]
        self.game = game
        self.reset()

    def update(self):
        """
        Called in the game loop to update
        the status of the sprite.
        Does nothing in the superclass
        """
        pass

    def draw(self):
        """
        Draws the sprite on the screen at its
        current position
        """
        self.game.surface.blit(self.image, self.position)

    def reset(self):
        """
        Called at the start of a new game to
        reset the sprite
        """
        pass
```

This will be the superclass for all sprites in the game

Called to set up the values in a sprite

Store the image in the sprite

Set the position in the sprite to the top left corner

Store the game reference in the sprite

Reset the sprite

Called when a sprite is to be updated

Called to ask a sprite to draw itself

Called to ask a sprite to reset itself



Sprite superclass

The code above defines the superclass for all the sprites in the game. You may have some questions about it.

Question: What is the `game` parameter used for in the initializer?

Answer: When the game creates a new sprite, it must tell the sprite which game it is part of because some sprites will need to use information stored in the game object. For example, if the cheese manages to capture a cracker, the score value will need to be updated.

Programmers say that the sprite class and the game class will be tightly *coupled*. Changes to the code in the `CrackerChaseGame` class might affect the behavior of sprites in the game. If the programmer of the `CrackerChaseGame` class changes the name of the variable that keeps the score from `score` to `game_score`, the `Update` method in the `Cheese` class will fail when the player captures a cracker. A lot of coupling between classes in a large system is a bad idea, but in the case of our game it makes the development much easier, so I think it's reasonable to make the program work in this way.

Question: Why are the `update` and `reset` methods empty?

Answer: You can think of the `Sprite` class as a template for subclasses. Some of the game elements will need methods to implement `update` and `reset` behaviors. The `cheese` will need a `reset` method that places it in the middle of the screen at the start of the game. The `cheese` will need an `update` method that moves it around the screen. The `cheese` class will be a subclass of `Sprite`, and adds its own version of these methods.

Question: How does the `draw` method work?

Answer: The `draw` method is called to ask the sprite to draw itself on the screen.

```
def draw(self):
    """
    Draws the sprite on the screen at its
    current position
    """
    self.game.surface.blit(self.image, self.position)
```

The game that the sprite is part of contains an attribute called `surface`, which is the pygame drawing surface for this game. The above method finds the game attribute from the sprite that's drawing itself. The game attribute was set when the sprite was created; the game attribute uses the game's `surface` property to blit the sprite image onto the screen.

The `Sprite` class doesn't do much, but it can be used to manage the background image for this game. The game will take place on a "tablecloth" background. We can think of this as a very large sprite that fills the screen. We can now make our first version of the game that contains a game loop that just displays the background sprite.

```

class CrackerChase:
    '''
    Plays the amazing cracker chase game
    '''

    def play_game(self):
        '''
        Starts the game playing
        Will return when the player exits
        the game.
        '''
        init_result = pygame.init()
        if init_result[1] != 0:
            print('pygame not installed properly')
            return

        self.width = 800
        self.height = 600
        self.size = (self.width, self.height)

        self.surface = pygame.display.set_mode(self.size)
        pygame.display.set_caption('Cracker Chase')
        background_image = pygame.image.load('background.png')
        self.background_sprite = Sprite(image=background_image,
                                       game=self)

        clock = pygame.time.Clock()
        while True:
            clock.tick(60)
            for e in pygame.event.get():
                if e.type == pygame.KEYDOWN:
                    if e.key == pygame.K_ESCAPE:
                        pygame.quit()
                        return
            self.background_sprite.draw()
            pygame.display.flip()

```

The diagram shows the following callouts for the code:

- `class CrackerChase:` — Class that contains the entire game
- `'''` —
- `Plays the amazing cracker chase game` —
- `'''` —
- `def play_game(self):` — Called to play the game
- `'''` —
- `Starts the game playing` —
- `Will return when the player exits` —
- `the game.` —
- `'''` —
- `init_result = pygame.init()` — Initialize pygame
- `if init_result[1] != 0:` —
- `print('pygame not installed properly')` —
- `return` — Quit if pygame is not installed on this machine
- `self.width = 800` —
- `self.height = 600` — Set the width and height of the game display
- `self.size = (self.width, self.height)` — Create a tuple that defines the screen size
- `self.surface = pygame.display.set_mode(self.size)` — Create the drawing surface
- `pygame.display.set_caption('Cracker Chase')` — Set the caption for the game screen
- `background_image = pygame.image.load('background.png')` —
- `self.background_sprite = Sprite(image=background_image,` —
- `game=self)` — Tell the sprite the game it is part of
- `clock = pygame.time.Clock()` — Create the game for the clock
- `while True:` — Game loop that runs forever
- `clock.tick(60)` — Ensure the game updates 60 times per second
- `for e in pygame.event.get():` — Get the events from pygame
- `if e.type == pygame.KEYDOWN:` — Is the event a key press?
- `if e.key == pygame.K_ESCAPE:` —
- `pygame.quit()` — Close the game screen
- `return` — Return from the game method
- `self.background_sprite.draw()` — Ask the background to draw itself
- `pygame.display.flip()` — Flip the back buffer to the front
- `if e.key == pygame.K_ESCAPE:` — If the key pressed is Escape, return from the game method
- `background_image = pygame.image.load('background.png')` — Load the background image
- `self.background_sprite = Sprite(image=background_image,` — Create the background sprite



Game class

The code above defines the class that will implement our game. You might have some questions about it.

Question: How does the game pass a reference to itself to the sprite constructor?

Answer: We know that when a method in a class is called, the `self` parameter is called to reference the object within which the method is running. We can pass `self` into other parts of the game that need it:

```
self.background_sprite = Sprite(image=background_image, game=self)
```

The code above makes a new `Sprite` instance and sets the value of the `game` argument to `self` so that the sprite now knows which game it is part of.

Question: Why does the game call the `draw` method on the sprite to draw it? Can't the game just draw the image held inside the sprite?

Answer: This is a very important question, and it comes down to responsibility. Should the sprite be responsible for drawing on the screen, or should the game do the drawing? I think drawing should be the sprite's job because it gives the developer a lot more flexibility.

For instance, adding smoke trails to some of the sprites in this game by drawing "smoke" images behind the sprite would be much easier to do if I could just add the code into the "smoky" sprites rather than the game having to work out which sprites needed smoke trails and draw them differently.

Question: Does this mean that when the game runs the entire screen will be redrawn each time, even if nothing on the screen has changed?

Answer: Yes. You might think that this is wasteful of computer power, but this is how most games work. It is much easier to draw everything from scratch than it is to keep track of changes to the display and only redraw parts that have changed.

The code below shows how we would start a game running:

```
# EG16-05 background sprite
```

```
game = CrackerChase()
```

Create a game instance

```
game.play_game()
```

Start the game running

Add a player sprite

The player sprite will be a piece of cheese that is steered around the screen. We've seen how a game can respond to keyboard events; now we'll create a player sprite and get the game to control it. The `Cheese` class below implements the player object in our game.

```
class Cheese(Sprite):
```

```
    """
```

```
    Player-controlled cheese object that can be steered
    around the screen by the player
    """
```

```
    """
```

```
    def reset(self):
```

Override the `reset` method in the sprite superclass

```
        """
```

```
        Reset the cheese position and stop any movement
```

```
        """
```

Center the cheese across the screen

```
        self.movingUp = False
```

Stop the cheese moving up

```
        self.movingDown = False
```

Stop the cheese moving down

```
        self.position[0] = (self.game.width - self.image.get_width())/2
```

```
        self.position[1] = (self.game.height - self.image.get_height())/2
```

```
        self.movement_speed=[5,5]
```

Set the initial move speed for the cheese

Center the cheese down the screen

Center the cheese across the screen

```
    def update(self):
```

```
        """
```

```
        Update the cheese position and then stop it moving off
        the screen.
        """
```

```
        """
```

```
        if self.movingUp:
```

If we are moving up, move the cheese up

```
            self.position[1] = self.position[1] - (self.movement_speed[1])
```

```
        if self.movingDown:
```

If we are moving down, move the cheese down

```
            self.position[1] = self.position[1] + (self.movement_speed[1])
```

```

if self.position[0] < 0:
    self.position[0]=0
if self.position[1] < 0:
    self.position[1]=0
if self.position[0] + self.image.get_width() > self.game.width:
    self.position[0] = self.game.width - self.image.get_width()
if self.position[1] + self.image.get_height() > self.game.height:
    self.position[1] = self.game.height - self.image.get_height()

def StartMoveUp(self):
    'Start the cheese moving up'
    self.movingUp = True

def StopMoveUp(self):
    'Stop the cheese moving up'
    self.movingUp = False

    'Other cheese movement methods go here...'

```



CODE ANALYSIS

Player sprite

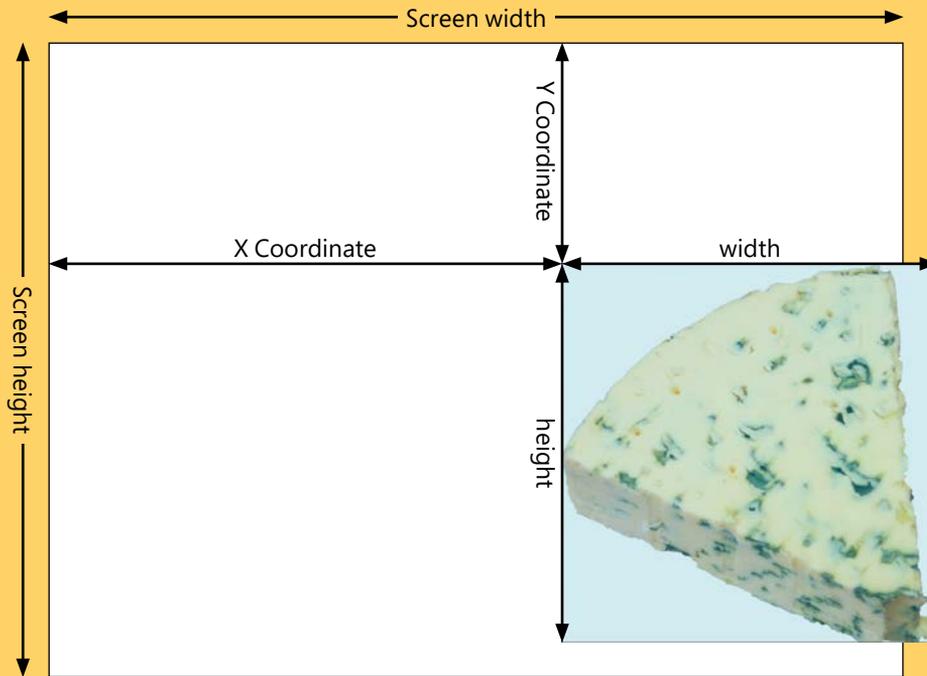
The code above defines the `Cheese` sprite. I've left off some of the movement methods to save space in the book, but you can find them all in the example program **EG16-06 Cheese Player** in the sample code for this chapter. You might have some questions about it.

Question: Why does the `Cheese` class not have an `__init__` or `draw` method?

Answer: The `Cheese` class is a subclass of the `Sprite` class we created earlier, which means the `Cheese` class inherits those two methods from the `Sprite` class.

Question: What do the `get_width` and `get_height` methods do?

Answer: These methods are provided by the pygame image class to allow a game to determine the dimensions of an image. We use them to make sure that the player cannot move the cheese off the screen.



The image above shows how this works. The program knows the position of the cheese and the width and height of the screen. If the x position plus the width of the cheese is greater than the width of the screen (as it is in the image above), the `update` method for the cheese will put the cheese back on the right edge:

```
if self.position[0] + self.image.get_width() > self.game.width:  
    self.position[0] = self.game.width - self.image.get_width()
```

The position of a sprite is held in a list, with the element at location 0 holding the x position of the sprite. The sprite can use its reference to the game to get the width of the screen and the `get_width` method to obtain the width of the sprite image. Note that in the above image, the cheese is not moving off the bottom of the screen. Forcing a sprite to stay on the screen in this way is called *clamping* the sprite.

The `Cheese` class also uses the width and the height of the sprite image to position the cheese in the center of the screen when the cheese is reset.

```
self.position[0] = (self.game.width - self.image.get_width())/2  
self.position[1] = (self.game.height - self.image.get_height())/2
```

Control the player sprite

The `game` class creates an instance of the cheese sprite and uses keyboard events to trigger message to the sprite to control its movement. Below is the `game` class code that does this. If you run the example program **EG16-06 Cheese Player**, you can see this in action. The player can move the cheese around the screen, but the cheese will not move off the edge of the screen.

```
cheese_image = pygame.image.load('cheese.png')
self.cheese_sprite = Cheese(image=cheese_image, game=self)

clock = pygame.time.Clock()

while True:
    clock.tick(60)
    for e in pygame.event.get():
        if e.type == pygame.KEYDOWN:
            if e.key == pygame.K_ESCAPE:
                pygame.quit()
                return
            elif e.key == pygame.K_UP:
                self.cheese_sprite.StartMoveUp()
            elif e.key == pygame.K_DOWN:
                self.cheese_sprite.StartMoveDown()
            'Other cheese movement key handlers go here...'

    self.background_sprite.draw()
    self.background_sprite.update()
    self.cheese_sprite.draw()
    self.cheese_sprite.update()
    pygame.display.flip()
```

Add a Cracker sprite

Moving the cheese around the screen is fun for a while, but we need to add some targets for the player. The targets are crackers the player must use to capture the cheese. When a cracker is captured, the game score is increased, and the cracker moves to another random position on the screen. The `Cracker` sprite is a subclass of the `Sprite` class:

```

class Cracker(Sprite):
    """
    The cracker provides a target for the cheese
    When reset, it moves to a new random place
    on the screen
    """
    def reset(self):
        self.position[0] = random.randint(0,
            self.game.width-self.image.get_width())
        self.position[1] = random.randint(0,
            self.game.height-self.image.get_height())

```

The `Cracker` class is very small because it gets most of its behavior from its superclass, the `Sprite` class. It just contains one method, `reset`, which uses the Python random number generator to pick a random position for the cracker. We can add it to our game by creating it and then drawing it in the game loop. The sample program **EG16-07 Cheese and cracker** shows how this works.

Figure 16-4 shows the game in action. The figure shows that there are at least two problems with this game. First, the cracker seems to be on top of the cheese. If the cheese is going to “capture” the cracker, it would look better if the cheese appeared to be “on top” of the cracker. We can fix this by changing the order in which the game elements are drawn. The pygame framework places images on the screen in the order they are drawn. The second problem with this game is that it looks a bit boring. I think we need more crackers to serve as additional targets.

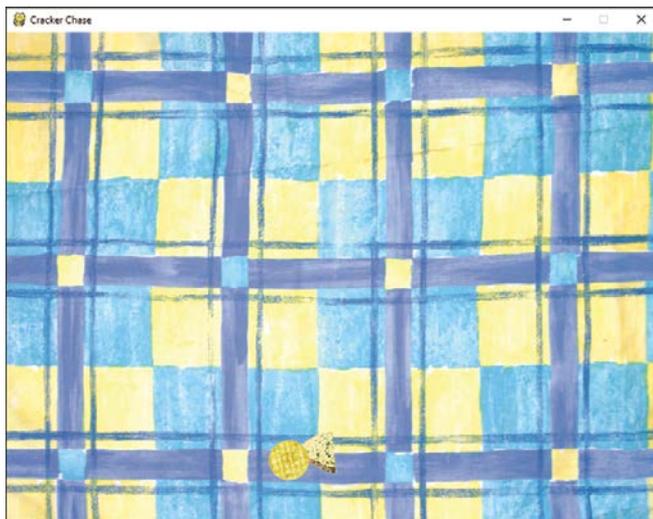


Figure 16-4 Cheese and cracker

Add lots of sprite instances

We could increase the number of crackers by creating more individual cracker instances:

```
cracker_image = pygame.image.load('cracker.png')
self.cracker1 = Cracker(image=cracker_image, game=self)
self.cracker2 = Cracker(image=cracker_image, game=self)
self.cracker3 = Cracker(image=cracker_image, game=self)
```

The code above would create three crackers called `cracker1`, `cracker2`, and `cracker3`. This would work, but it would be hard to manage because the game would have to update and draw each of these sprites individually. It would turn into a real problem when game players request 50 crackers on the screen. Whenever we've had this problem in the past, we have used a collection of some kind (usually a list) to solve it. We can do this here, too.

```
self.sprites = [] Create a list to hold all the sprites in the game

cracker_image = pygame.image.load('cracker.png') Load the cracker image

for i in range(20): Create a for loop that goes around 20 times
    cracker_sprite = Cracker(image=cracker_image, game=self) Create a Cracker sprite
    self.sprites.append(cracker_sprite) Add the sprite to the list of sprites
```

The statements above create 20 cracker sprites. The game now contains a list, called `sprites`, which holds all the sprites in the game.

```
for sprite in self.sprites:
    sprite.update()

for sprite in self.sprites:
    sprite.draw()
```

Above are the statements that we can use in the game loop to update and draw the cracker sprites. In the sample game **EG16-08 Cheese and crackers**, you can see how this works. This version of the game also adds the background and the cheese objects to the `sprites` list so that everything in the game is drawn and updated by the above two loops. **Figure 16-5** shows the game now. If we want to have even more crackers, we just need to change the limit of the range in the `for` loop that creates them.

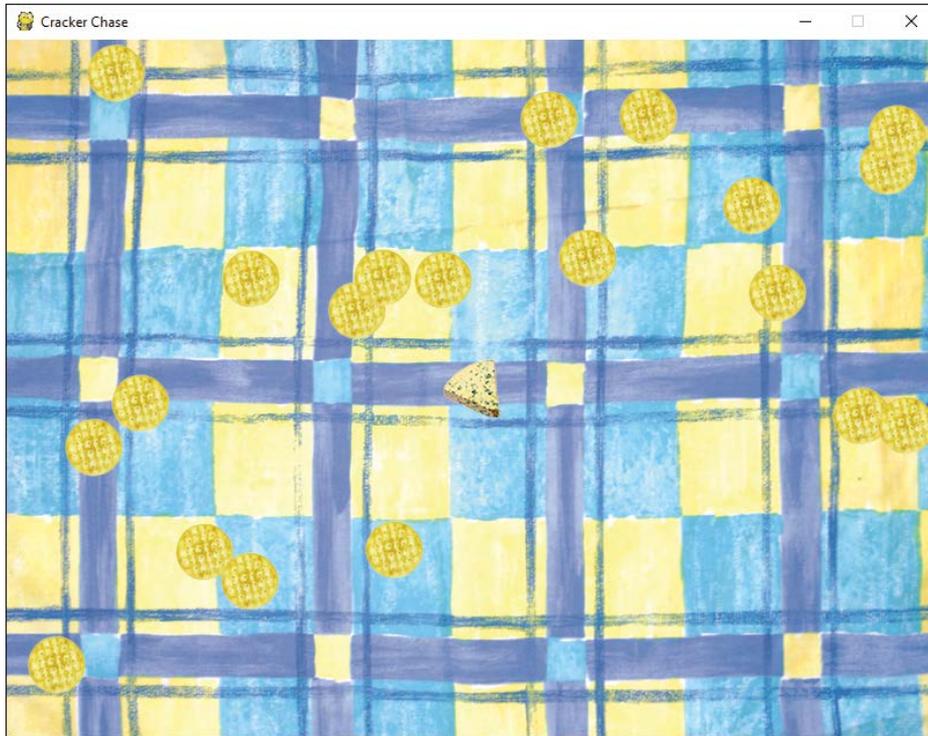


Figure 16.5 Cheese and multiple crackers

Catch the crackers

The game now has lots of crackers and a piece of cheese that can chase them. But nothing happens when the cheese “catches” a cracker. We need to add a behavior to the `Cracker` that detects when the cracker has been “caught” by the cheese. A cracker is caught by the cheese when the cheese moves “on top” of it. The game can detect when this happens by testing that rectangles enclosing the two sprites *intersect*.

Figure 16-6 shows the cheese in the process of catching a cracker. The rectangles around the cheese and cracker images are called *bounding boxes*. When one bounding box moves “inside” another, we say that the two are intersecting. When the cracker updates, it will test to see whether it intersects with the cheese.

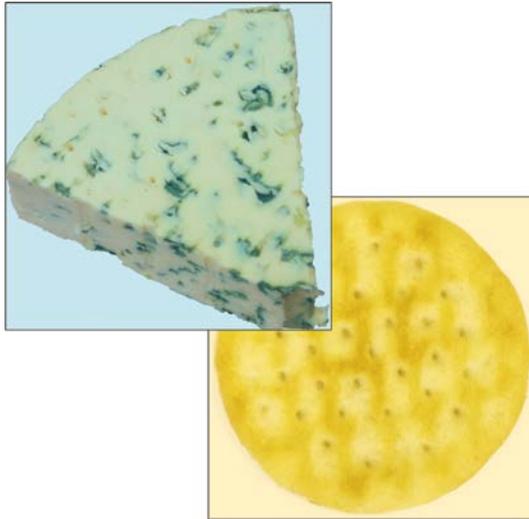


Figure 16-6 Intersecting sprites

Figure 16-7 shows how the test will work. In this figure, the two sprites are not intersecting because the right edge of the cheese is to the left of the left edge of the cracker. In other words, the cheese is too far to the left to intersect with the cracker. This would also be true if the cheese were above, below, or to the right of the cracker. We can create a method that tests for these four situations. If any of them are true, the rectangles do not intersect.

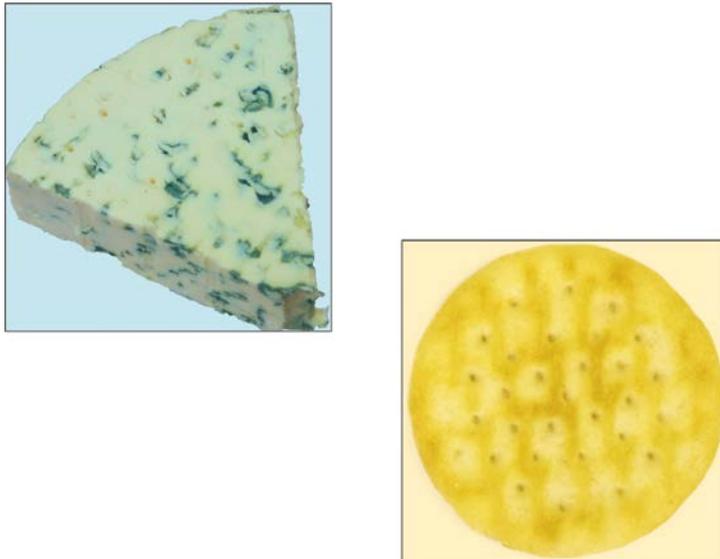


Figure 16-7 Non-intersecting sprites

```

def intersects_with(self, target):
    """
    Returns True if this sprite intersects with
    the target supplied as a parameter
    """
    max_x = self.position[0]+self.image.get_width()
    max_y = self.position[1]+self.image.get_height()

    target_max_x = target.position[0]+target.image.get_width()
    target_max_y = target.position[1]+target.image.get_height()

    if max_x < target.position[0]:
        return False

    if max_y < target.position[1]:
        return False

    if self.position[0] > target_max_x:
        return False

    if self.position[1] > target_max_y:
        return False

    # if we get here, the sprites intersect
    return True

```

Get the right edge of this sprite
 Get the bottom edge of this sprite
 Get the right edge of the target
 Get the bottom edge of the target
 Is this sprite to the left?
 Is this sprite underneath?
 Is this sprite to the right?
 Is this sprite above?
 Return True because the sprites intersect

The method is an attribute of a `Sprite` object, which returns `True` if the sprite intersects with a particular target. We add this method to the `Sprite` class so that all sprites can use it. Now we can add an `update` method to the `Cracker` class that checks to see whether the cracker intersects with the cheese:

```

def update(self):
    if self.intersects_with(game.cheese_sprite):
        self.captured_sound.play()
        self.reset()

```

Have we been captured?
 Play our capture sound effect
 Reset the position of the cracker

Add sound

The preceding `update` method plays a sound effect when a cracker is “captured” by the cheese. The pygame framework provides a `Sound` class to manage sound playback. When an instance of `Sound` is created, it is given the name of the file that contains the sound data.

```
cracker_eat_sound = pygame.mixer.Sound('burp.wav')
```

The statement above creates a `Sound` instance called `cracker_eat_sound` from the sound file `burp.wav`. We pass this sound into a `Cracker` when we create a new instance:

```
cracker_sprite = Cracker(image=cracker_image, game=self,
                        captured_sound=cracker_eat_sound) — Store the capture sound in the cracker
```

For this to work, we must modify the `__init__` method in the `Cracker` to store the sound in the cracker:

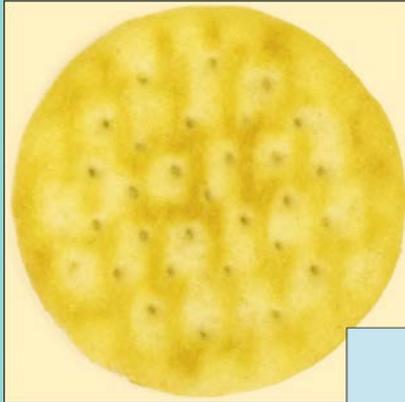
```
def __init__(self, image, game, captured_sound):
    super().__init__(image, game) — Call the constructor in the superclass
    self.captured_sound = captured_sound — Set the sound attribute of the cracker
```

The attribute `captured_sound` in the `Cracker` object can be used to play the sound effect when that cracker is eaten. In the present version of the game, all the crackers make the same sound when they are eaten, but we could use different sound effects for each cracker if we wished. The example program **EG16-09 Capturing crackers** lets the player capture crackers. When a cracker is captured, the game plays a sound effect and the cracker moves to a different location.

If you want to create your own sound effects, you can use the program Audacity to capture and edit sounds. It is a free download from www.audacityteam.org and is available for most operating systems.



Bad collision detection



There are some problems with using bounding boxes to detect collisions. The image above shows that the cheese and the cracker are not colliding, but the game will think that they are. This should not be too much of a problem for our game. It makes it easier for the player, as they don't always have to move the cheese right over the cracker to score a point. However, the player might have grounds for complaint if the game decides they have been caught by a killer tomato because of this issue. There are three ways to solve this problem:

- When the bounding boxes intersect (as they do above), we could check the intersecting rectangle (the part where the two bounding boxes overlap) to see if they have any pixels in common. Doing so provides very precise collision detection, but it will slow down the game.
- Alternatively, we could detect collisions using distance rather than intersection, which works well if the sprites are mostly round.
- The final solution is the one I like best. I could make all the game images rectangular, so the sprites fill their bounding boxes and the player always sees when they have collided with something.

PROGRAMMER'S POINT

When you write a game, you control the universe

One of the reasons I like writing games so much is that I have complete control of what I'm making. If I'm solving a problem for a customer, I must deliver certain outcomes. But in a game, I can change what it does if I find a problem. I can also redefine the gameplay if I make a mistake in the program. Sometimes, this produces a more interesting behavior than the one I was trying to create. This has happened on a number of occasions.

Add a killer tomato

Currently, the game is not much of a game. There is no jeopardy for the player. When you make a game, you set up something that the player is trying to achieve. Then you add some elements that will make this difficult for them. In the case of the game "Cracker Chase," I want to add "killer tomatoes" that will relentlessly hunt down the player. As the game progresses, I want the player to be chased by increasingly more tomatoes until the game becomes all about survival. The tomatoes will be interesting because I'll give them *artificial intelligence* and *physics*.

Add "artificial intelligence" to a sprite

Artificial intelligence sounds very difficult to achieve, but in the case of this game, it is actually very simple. At its heart, artificial intelligence in a game simply means making a program that would behave like a person in that situation. If you were chasing me, you'd do this by moving toward me. The direction you would move would depend on my position relative to you. If I were to your left, you'd move left, and so on. We can put the same behavior into our killer tomato sprite:

```
if game.cheese_sprite.position[0] > self.position[0]:
    self.x_speed = self.x_speed + self.x_accel
else:
    self.x_speed = self.x_speed - self.x_accel

if game.cheese_sprite.position[1] > self.position[1]:
    self.y_speed = self.y_speed + self.y_accel
else:
    self.y_speed = self.y_speed - self.y_accel
```

Is the player to the right of the tomato?

Accelerate to the right

Accelerate to the left

Is the player below the tomato?

Accelerate down

Accelerate up

This condition shows how we can make an intelligent killer tomato. It compares the x positions of the `cheese_sprite` and the tomato. If the cheese is to the right of the

tomato, the `x` speed of the tomato is increased to make it move to the right. If the cheese is to the left of the tomato, it will accelerate in the other direction. The code above then repeats the process for the vertical positions of the two sprites. The result is a tomato that will move intelligently toward the cheese. Note that this means we could make a “cowardly” tomato that runs away from the player by making the acceleration negative so that the tomato accelerates in the opposite direction of the cheese.

PROGRAMMER’S POINT

Using “artificial intelligence” makes games much more interesting

There is a lot of debate as to whether “game artificial intelligence” is actually “proper” artificial intelligence. You can find a very good discussion of the issue here: <https://software.intel.com/en-us/articles/designing-artificial-intelligence-for-games-part-1>. I personally think that you can call this kind of programming “artificial intelligence” because players of a game really do react as if they are interacting with something intelligent when faced with something like our killer tomato. You can make a game much more compelling by giving game objects the kind of intelligence described above.

Add physics to a sprite

Each time the game updates, it can update the position of the objects on the screen. The amount that each object moves each time the game updates is the *speed* of the object. When the player is moving, the cheese’s position is updated by the value 5. In other words, when the player is holding down a movement key, the position of the cheese in that direction is being changed by 5. The updates occur 60 times per second because this is the rate at which the game loop runs. In other words, the cheese would move 300 pixels (60×5) in a single second. We can increase the speed of the cheese by adding a larger value to the position each time it is updated. If we used a speed value of 10, we’d find that the cheese would move twice as fast.

Acceleration is the amount that the speed value is changing. The statements below update the `x_speed` of the tomato by the acceleration and then apply this speed to the position of the tomato.

```
self.x_speed = self.x_speed + self.x_accel
self.position[0] = self.position[0] + self.x_speed
```

Add the acceleration to the speed

Update the position of the sprite

The initial speed of the tomato is set to zero, so each time the tomato is updated, the speed (and hence the distance it moves) will increase. If we do this in conjunction with “artificial intelligence,” we get a tomato that will move rapidly toward the player.

If we just allowed the tomato to accelerate continuously, we'd find that the tomato would just get faster and faster, and the game would become unplayable.

The statement below adds some "friction" to slow down the tomato. The friction value is less than 1, so each time we multiply the speed by the friction, it will be reduced, which will cause the tomato to slow down over time.

```
self.x_speed = self.x_speed * self.friction_value
```

Multiply the speed by the friction

The friction and acceleration values are set in the `reset` method for the `Tomato` sprite:

```
def reset(self):
    self.entry_count = 0
    self.friction_value = 0.99
    self.x_accel = 0.2
    self.y_accel = 0.2
    self.x_speed = 0
    self.y_speed = 0
    self.position = [-100, -100]
```

After some experimentation, I came up with the acceleration value of 0.2 and a friction value of 0.99. If I want a sprite that chases me more quickly, I can increase the acceleration. If I want the sprite to slow down more quickly, I can increase the friction. You can have a lot of fun playing with these values. You can create sprites that drift slowly toward the player and, by making the acceleration negative, you can make them run away from the player.

PROGRAMMER'S POINT

When you write a game, you can always cheat

When you're writing a game, you should always start with the simplest, fastest way of getting an effect to work, and then improve it if necessary.

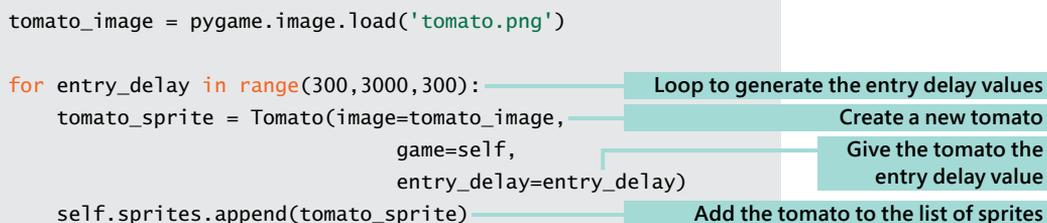
The "physics" that I'm using are not really an accurate simulation of physical objects. The way that I've implemented friction is not very realistic, but it works and gives the player a good experience. I find it interesting that six or seven lines of Python can make something that behaves in such a believable way. The Cracker Chase game uses very simple collision detection, artificial intelligence, and physics, but it is still fun to play. It really feels as if the tomatoes are chasing you. Making the physics model completely accurate would take a lot of extra work and would add very little to the gameplay.

Create timed sprites

It's important that a game be progressive. If the game started with lots of killer tomatoes, the player would not last very long and would not enjoy the experience. I'd like each tomato to appear every 5 seconds. We can do this by giving each tomato an "entry delay" value when we construct it:

```
tomato_image = pygame.image.load('tomato.png')

for entry_delay in range(300, 3000, 300):
    tomato_sprite = Tomato(image=tomato_image,
                           game=self,
                           entry_delay=entry_delay)
    self.sprites.append(tomato_sprite)
```



The code is annotated with callouts:

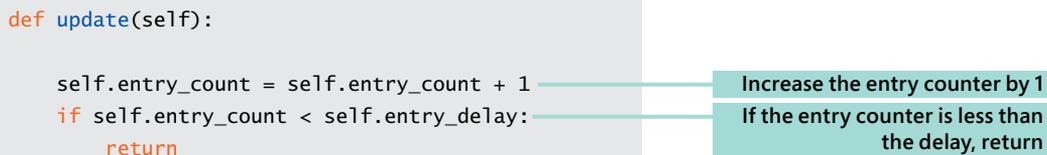
- `for entry_delay in range(300, 3000, 300):` — Loop to generate the entry delay values
- `tomato_sprite = Tomato(image=tomato_image,` — Create a new tomato
- `game=self,` — Give the tomato the
- `entry_delay=entry_delay)` — entry delay value
- `self.sprites.append(tomato_sprite)` — Add the tomato to the list of sprites

This code uses a version of the `range` function that we haven't seen before. The first argument to the `range` is the start value, which in this case is `300`. The second argument is the upper limit, and the third argument is the "step" between values. This will give us values of `entry_delay` that start at `300` and then go up in steps to `2700` (note that the value `3000` is the limit).

The `__init__` method in the `Tomato` class stores the value of `entry_delay` and is used to delay the entry of the sprite:

```
def update(self):

    self.entry_count = self.entry_count + 1
    if self.entry_count < self.entry_delay:
        return
```



The code is annotated with callouts:

- `self.entry_count = self.entry_count + 1` — Increase the entry counter by 1
- `if self.entry_count < self.entry_delay:` — If the entry counter is less than the delay, return

The `update` method is called 60 times per second. The first tomato has an entry delay of `300`, which means that it will arrive at `300/60` seconds, which is 5 seconds after the game starts. The next tomato will appear 5 seconds after that, and so on, up until the last one. The example program **EG16-10 Killer tomato** shows how this works. It can get rather frantic after a few tomatoes have turned up and are chasing you.

Complete the game

We now have a program that provides some gameplay. Now we need to turn this into a proper game. To do so, we need to add a start screen, provide a way that the player can start the game, detect and manage the end of the game, and then, because it adds a lot to the gameplay, add a high score.

Add a start screen

A start screen is where the player will—you guessed it—start the game. Then, when the game is complete, the game returns to the start screen. We can add a start screen to the Cracker Chase game by using a flag value to indicate the mode of the game:

```
def start_game(self):  
    for sprite in self.sprites:  
        sprite.reset()  
    self.score=0  
    self.game_running = True
```

- Reset all the sprites
- Clear the game score
- Set the flag to indicate that the game is running

Above is the method that starts a game playing. It resets all the sprites, sets the score to zero, and sets the `game_running` flag to `True`. The `game_running` flag controls the behavior of the game loop:

```
while True:  
    clock.tick(60)  
    if self.game_running:  
        self.update_game()  
        self.draw_game()  
    else:  
        self.update_start()  
        self.draw_start()  
    pygame.display.flip()
```

- Repeat game forever
- Keep the frame rate to 60 frames per second
- Is the game active?
- Update the game
- Draw the game
- Update the start screen
- Draw the start screen
- Display the back buffer

This is the game loop for the game. The code that updates the game and draws it is now in methods that are called if the game is running. If the game is not running, methods are called to update and draw the start screen.

```

def update_start(self):
    for e in pygame.event.get():
        if e.type == pygame.KEYDOWN:
            if e.key == pygame.K_ESCAPE:
                pygame.quit()
                sys.exit()
            elif e.key == pygame.K_g:
                self.start_game()

```

The start screen `update` behavior checks for two keys:

- If the G key is pressed, the `start_game` method is called to start the game.

If the Escape key is pressed, the method shuts down pygame by calling `quit` and then using the `exit` method from the `sys` module to end the program.

Use `exit` to shut down Python

The `exit` method is in the `sys` module, which means that the game must import the module:

```
import sys
```

Once we have imported `sys`, we can call the `exit` function from the module to exit a Python program instantly.

```
sys.exit()
```

Draw text in pygame

The start screen will display information for the player, as shown in **Figure 16-8**. The pygame framework can draw text on the screen. It uses a `Font` object that is created when the game starts.

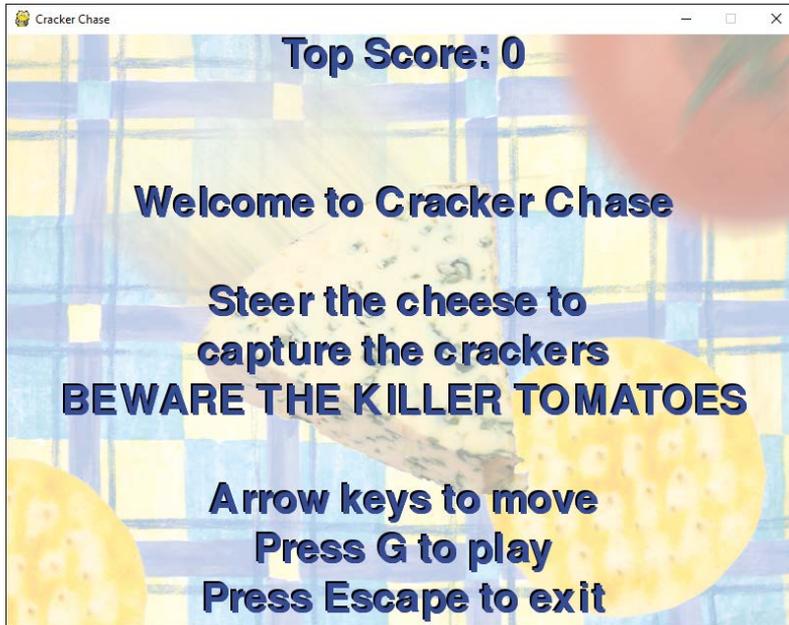


Figure 16-8 Start screen

```
self.font = pygame.font.Font(None, 60)
```

The initializer for the font accepts two parameters—the font design to use and the size of the font. The statement above specifies `None` for the font design, which will select the default pygame font. The size of 60 gives a text size that works well for the game. To place a message on the screen, the game first renders the text using the font.

```
text = self.font.render('hello world', True, (255,0,0))
```

The `render` method accepts three arguments:

- The first is a string that contains the text to be rendered.
- The second argument selects *aliasing*. This technique smooths the edges of the characters, and you should use it to make your text look nice.
- The third argument specifies the color of the text. It contains the amount of red, blue, and green that the text color should contain. The maximum color intensity is 255.

The code above will render “hello world” in bright red.

Once the text has been rendered, the next step is to blit it onto the display. We do this the same way we blit images.

```
self.surface.blit(text, (0,0))
```

The first argument to the `blit` method is for the text to be drawn; the second argument is the location on the screen. The statement above would render “hello world” in the top left corner of the screen. A program can get the width and the height of rendered text, which can be used to center text on the screen. The `CrackerChase` class contains a little method that draws text on the screen:

```
def display_message(self, message, y_pos):
    """
    Displays a message on the screen
    The first argument is the message text
    The second argument is the vertical position
    of the text
    The text is drawn centered on the screen
    It is drawn with a black shadow
    """
    shadow = self.font.render(message, True, (0,0,0))
    text = self.font.render(message, True, (0,0,255))
    text_position = [self.width/2 - text.get_width()/2, y_pos]
    self.surface.blit(shadow, text_position)
    text_position[0] += 2
    text_position[1] += 2
    self.surface.blit(text, text_position)
```

The diagram shows a code block with several lines of code. Lines of code are connected to callout boxes on the right by thin lines. The callouts are: 'Render the text in black' pointing to the `shadow` render line; 'Render the text in blue' pointing to the `text` render line; 'Calculate the position of the text' pointing to the `text_position` assignment line; 'Draw the shadow' pointing to the first `blit` call; 'Move the draw position across' pointing to the `text_position[0] += 2` line; 'Move the draw position down' pointing to the `text_position[1] += 2` line; and 'Draw the text' pointing to the second `blit` call.

This method actually draws the text twice. The first time, the text is drawn in black, and then the text is drawn again in blue. The second time the text is drawn, it is moved slightly to make it appear that the black text is a shadow.

This method uses the `+=` operator, which can be used to increase the value of a variable. Rather than writing:

```
text_position[0] = text_position[0]+2
```

You can write:

```
text_position[0] += 2
```

There are similar operators for subtract (`-=`), multiply (`*=`) and divide (`/=`).

If you look closely at Figure 16-8, you can see that the result of this extra drawing is that text looks three-dimensional, which makes text stand out on the screen.

PROGRAMMER'S POINT

Don't worry about making the graphics hardware work for you

You might think it's rather extravagant to draw all the text on the screen twice just to get a shadow effect. However, modern graphics hardware is perfectly capable of many thousands of drawing operations per second. I've been known to draw text twenty times just to get a nice blurred shadow effect behind it. If you think something might look good, my advice is to try it and only worry about performance if the game seems to run very slowly after you've done it.

```
def draw_start(self):
    self.start_background_sprite.draw()
    self.display_message(message='Top Score: ' + str(self.top_score), y_pos=0)
    self.display_message(message='Welcome to Cracker Chase', y_pos=150)
    self.display_message(message='Steer the cheese to', y_pos=250)
    self.display_message(message='capture the crackers', y_pos=300)
    self.display_message(message='BEWARE THE KILLER TOMATOES', y_pos=350)
    self.display_message(message='Arrow keys to move', y_pos=450)
    self.display_message(message='Press G to play', y_pos=500)
    self.display_message(message='Press Escape to exit', y_pos=550)
```

Above is the `draw_start` method for the game, which draws the sprite that contains the background image and then displays the help messages on the display.

PROGRAMMER'S POINT

Make sure you tell people how to play your game

In my long and distinguished career in computing, I've judged quite a few game development competitions. I've lost count of the number of games that I've tried to play and failed because the game doesn't tell me what to do. The problem is usually that everyone focuses on making the game, and not on telling people how to play it. Failing at a game while you work out which keys you are supposed to press doesn't make for a very good introduction to it, so make sure that you make the instructions clear and present them right at the start.

End the game

The start screen allows the player to play the game. We've seen that the game has two states, which are managed by the `game_running` attribute. This attribute is set to `True` when the game is running and `False` when the start screen is displayed. Now we need to create the code that manages the `game_running` value. At the start of this section, we saw that the game contained a method that started the game. The game also contains a method to end it.

```
def end_game(self):
    self.game_running = False
    if self.score > self.top_score:
        self.top_score = self.score
```

The `end_game` method sets `game_running` to `False`. It also updates the `top_score` value. If the current score is greater than the highest score so far, it is updated to the new top score.

PROGRAMMER'S POINT

Adding a high score makes a game much more interesting

Adding a high score to a game makes the game much more compelling. Players will spend a lot of time trying to beat their previous scores. A good improvement to this game would be to make it save the high score in a file and load the high score when the game starts.

Detect the game end

The game ends when the player collides with a killer tomato, which is detected in the `update` method for the tomato sprite:

```
def update(self):
    ' position update code for the tomato here'
    if self.intersects_with(game.cheese_sprite):
        self.game.end_game()
```

We can add more logic to make the game more interesting. We could give the player a health value that reduces each time he or she collides with a tomato. We could make the health slowly recover over time. We could even add the traditional "three lives" that are standard for games like this.

PROGRAMMER'S POINT

Always make a playable game

Something else I noticed while judging game development competitions was that some teams would produce a brilliant piece of gameplay but not attach it to a game. You'd start playing the game and find that it never actually ended. You should make sure that your game is a complete game from the very start. The game should have a beginning, middle, and end. As you have seen in this section, it's easy to do this, but when people start making a game, they seem to leave it to the last minute to create the game start screen and the game ending code, so that what they produce is not a game, but more of a technical demo, which is not quite the same thing. Making your game into a proper game right from the start also makes it much easier for people to try it and then give you feedback.

Score the game

Each time the cheese collides with a cracker, the game score is increased. The score is updated in the `update` method for the cracker sprite:

```
def update(self):
    if self.intersects_with(game.cheese_sprite):
        self.captured_sound.play()
        self.reset()
        self.game.score += 10
```

Update the game score

The score is displayed on the screen each time the game display is drawn by the `draw_game` method.

```
def draw_game(self):
    for sprite in self.sprites:
        sprite.draw()
    status = 'Score: ' + str(game.score)
    self.display_message(status, 0)
```

Draw all the game sprites

Assemble the score message

Display the score at the top of the screen

You can find the completed game in the folder **EG16-11 Complete Game**. It's fun to play for short bursts, particularly if there are a few of you trying to beat the high score. My highest score so far is 380, but I never was any good at playing video games.



Make a game of your own

The Cracker Chase game can be used as the basis of any sprite-based game you might like to create. You can change the artwork, create new types of enemies, make the game two-player, or add extra sound effects. When I said at the start of this book that programming is the most creative thing you can learn to do, this is the kind of thing I was talking about. You can create a game called “Closet frenzy” where you are chased around by coat hangers while you search for a matching sock. You could create “Walrus Space Rescue,” where you must steer an interplanetary walrus through an asteroid minefield. Anything you can think up, you can build. However, one word of caution. Don’t have too many ideas. I’ve seen lots of game development teams get upset because they can’t get all their ideas to work at once. It is much more sensible to get something simple working and then add things to it later.

What you have learned

In this chapter, you created a playable game and discovered how the pygame framework lets you work with graphics and sound. You found that a class hierarchy, with a sprite superclass and different game objects as subclasses of this is a great way to create game objects. You also discovered that games work by having a “game loop” that repeatedly updates and draws items on the screen. You used the event mechanism of pygame to capture keyboard input, and you used events to control an object on the screen. You’ve seen that “artificial intelligence” can be created with a couple of if conditions, and physics can be implemented using a few calculations. You also implemented a start screen and a game screen to make a complete game experience.

Hopefully, you’ve also taken a few ideas of your own and used them to create some more games.

Here are some points to ponder about game development.

Do all games work using a game loop?

Most games use a game loop. A text-based adventure will work by reading in what you type and replying, but most modern games work with a loop.

Why are draw and update separate methods?

You might wonder why I separated the draw and update behaviors in the game. Although they are separate methods, they always seem to be called together. Why not have just one method (perhaps called `do_game`) which does both?

The answer has to do with performance. For simple games like Cracker Chase, it's perfectly fine for the drawing and updating to take place at the same rate. However, if you're running on a low-performance platform, you may want to update the game at a different rate from the rate you draw it. The reason for this is that people are much more tolerant of the game display "flickering" than they are for changes in the speed of a game update. If the game update slows down, it can cause problems with collisions not being detected (for example, bullets might pass right through things without the game noticing they had collided). For this reason, a game should separate drawing and updating so that the two processes can be made to run at different speeds if required.

How would I create an attract mode for my game?

Currently, our game just has two states, the start screen and the game screen. Many games have an "attract mode" screen as well, which displays some gameplay. Creating an attract mode screen is quite easy. We could make an "AI player" who moved the cheese around the screen in a random way, and then just run the game with the random player at the controls. We could add an "attract mode" behavior to the tomatoes so that they were aiming for a point some distance from the player, to make the game last longer in demo mode.

How could I make the gameplay the same each time the game is played?

The game uses the Python random number generator to produce the position of the crackers, which means each time the game runs, the crackers are in a different position. We can use the [seed](#) function from the Random module to give the Python random number generator the same seed before each game. This would mean that the crackers would be drawn and would respawn in the same sequence each time the game was played. A determined player could learn the pattern and use this to get a high score.

Is the author of the game always the best person at playing it?

Most definitely not. I'm often surprised how other people can be much better than me at playing games I've created. Sometimes they even try to help me with hints and tips about things to do in the game to get a higher score.