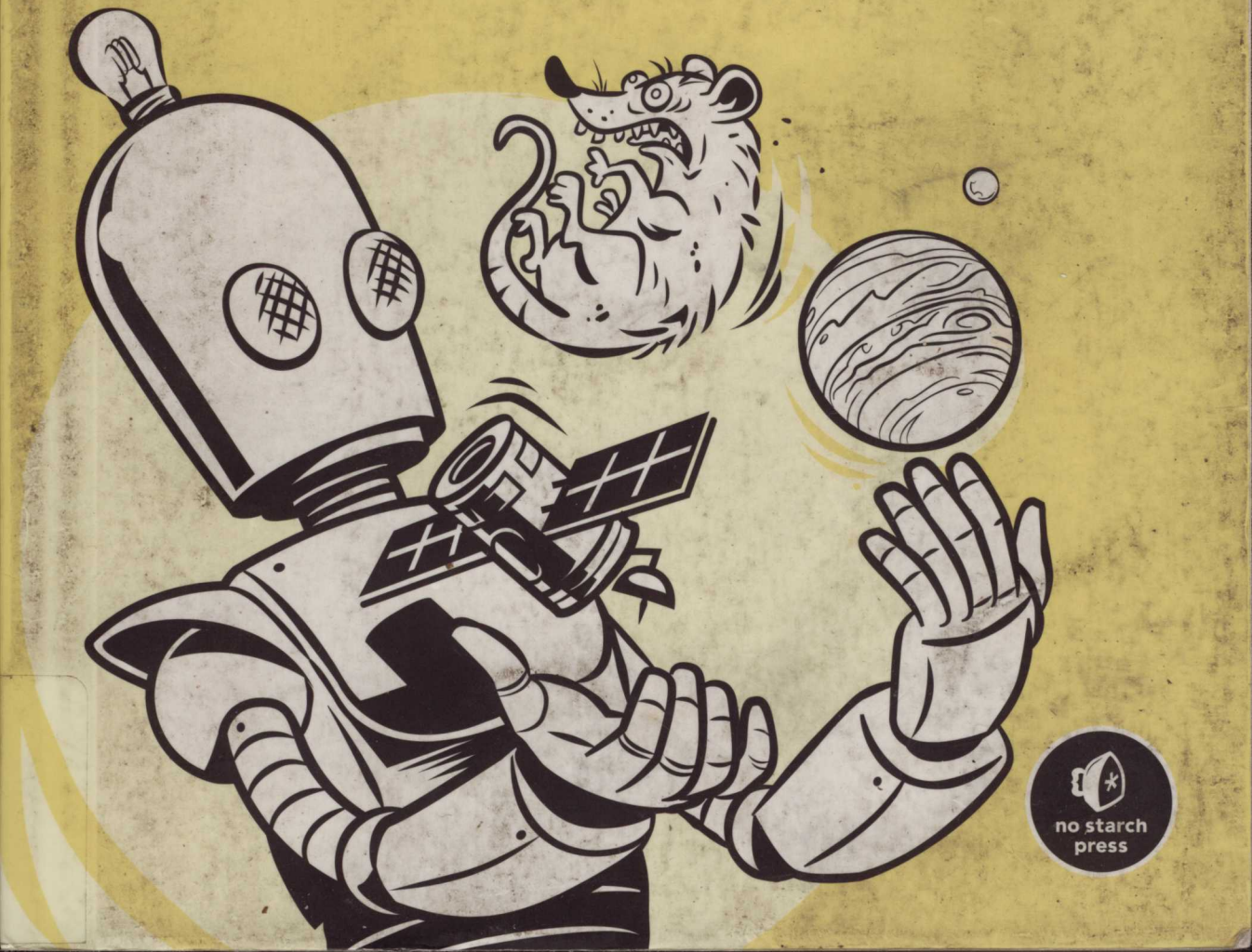
IMPRACTICAL PYTHON PROJECTS

PLAYFUL PROGRAMMING ACTIVITIES TO *MAKE YOU SMARTER*

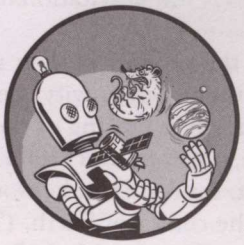
LEE VAUGHAN



no starch press

14

MAPPING MARS WITH THE MARS ORBITER



The *Mars Orbiter* space probe has been successfully injected into Martian orbit, but all is not well. The orbit is highly elliptical, and the project's mapping objectives require a low-altitude circular orbit. Fortunately, there's just enough propellant on board to correct things, assuming the eggheads at Mission Control have the patience and skill to pull it off!

In this chapter, you'll design and build a game based on this scenario. You'll use `pygame` again (for an overview of `pygame`, see "A Slice of `pygame`" on page 267), and you'll do your part to advance STEM (science, technology, engineering, and mathematics) education by making the game real enough to teach players the fundamentals of orbital mechanics.

NOTE *Although they share the same name, the* Mars Orbiter *space probe in the game ha... no direct relationship to the* Mars Orbiter Mission *launched by the* Indian Spac... Research Organization (ISRO) *in 2014. The game probe is patterned after the* Mars Global Surveyor, *launched by NASA in 1996.*

Astrodynamics for Gamers

Because you'll want your game to be as realistic as possible, a quick review of some of the basic science underlying spaceflight is in order. This will be short, sweet, and tailored to game development and play.

The Law of Universal Gravity

The theory of gravity states that massive objects—like stars and planets—warp both space and time around them, similar to how a heavy bowling ball placed on a mattress causes a depression that is sudden and sharp near the ball but quickly levels off. This behavior is captured mathematically by Isaac Newton's law of universal gravitation:

$$F = \frac{m_1 * m_2}{d^2} G$$

where F is the force of gravity, m_1 is the mass of object 1, m_2 is the mass of object 2, d is the distance between objects, and G is the gravitational constant ($6.674 \times 10^{-11}\ N \cdot m^2 \cdot kg^{-2}$).

Two objects pull on each other according to the product of their masses divided by the square of the distance between them. So, gravity is much stronger when objects are close together, like the deep bowing of the mattress just beneath the bowling ball. To illustrate, a 220-pound (100 kg) man would weigh over half a pound less on top of Mt. Everest than he would at sea level, where he would be 8,848 m closer to the center of Earth. (This assumes the mass of the planet is 5.98×10^{24} kg and sea level is 6.37×10^6 m from the center.)

Today, we generally think of gravity as a *field*—like the mattress in the bowling ball analogy—rather than as Newton's point of attraction. This field is still defined with Newton's law and results in *acceleration*, usually expressed in m/sec^2.

According to Newton's second law of motion, force is equal to mass × acceleration. You can calculate the force exerted by object 1 (m_1) on object 2 (m_2) by rewriting the gravitational equation as:

$$a = \frac{-G * m_1}{d^2}$$

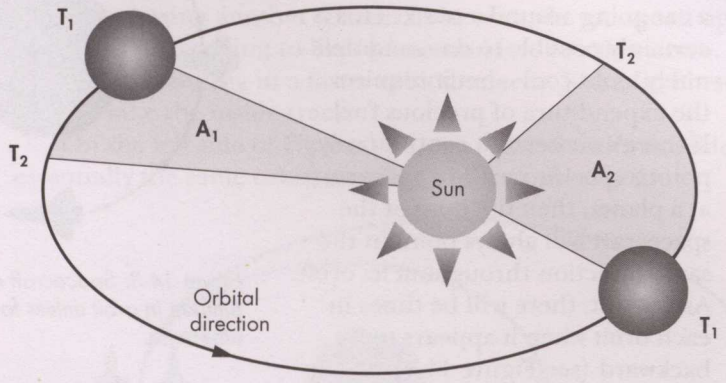
where a = acceleration, G is the gravitational constant, m_1 is the mass of one of the objects, and d is the distance between objects. The direction of force is from object 2 toward the center of mass of object 1 (m_1).

The pull of very small objects on large ones is generally ignored. For example, the force exerted by a 1,000 kg satellite on Mars is about 1.6×10^{-21} times smaller than the force exerted by Mars on the satellite! Thus, you can safely ignore the satellite's mass in your simulation.

NOTE *As a simplification in this project, distance is calculated from the center points of objects. In real life, an orbiting satellite would experience subtle changes in gravitational acceleration due to changes in a planet's shape, topography, crustal density, and so on. According to the* Encyclopedia Britannica, *these changes cause gravitational acceleration at Earth's surface to vary by about 0.5 percent.*

Kepler's Laws of Planetary Motion

In 1609, astronomer Johann Kepler discovered that planetary orbits are ellipses, allowing him to explain and predict the motion of the planets. He also found that a line segment drawn between the sun and an orbiting planet sweeps out equal areas in equal time intervals. This idea, known as Kepler's second law of planetary motion, is demonstrated in Figure 14-1, where a planet is shown at different points in its orbit.



Area A_1 = Area A_2

$(T_2 - T_1$ in $A_1) = (T_2 - T_1$ in $A_2)$

Figure 14-1: Kepler's second law of planetary motion: orbital speed increases as planets near the sun.

This law applies to all celestial bodies, and it means that an orbiting object speeds up as it gets close to the body it is orbiting and slows down as it travels farther away.

Orbital Mechanics

Orbiting is basically free-falling forever. You're falling into the core of a planet's gravity well—located at its literal core—but your tangential velocity is fast enough that you keep missing the planet (see Figure 14-2). As long as you balance your momentum with the force of gravity, the orbit will never end.

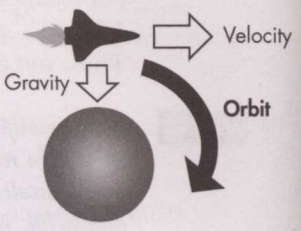


Figure 14-2: Orbit is achieved when a spacecraft's velocity keeps it "free-falling" around a celestial body.

Some counterintuitive things can happen when you orbit a planet in the vacuum of space. With no friction or wind resistance, spacecraft can behave in unexpected ways.

Flying Backward

If you've ever watched an episode of *Star Trek*, you've probably noticed how the orbiting *Enterprise* seems to steer its way around planets, like a car going around a track. This is certainly possible to do—and definitely looks cool—but it requires the expenditure of precious fuel. If there's no need to continuously point a specific part of a spacecraft at a planet, then the nose of the spacecraft will always point in the same direction throughout its orbit. As a result, there will be times in each orbit when it appears to fly backward (see Figure 14-3).

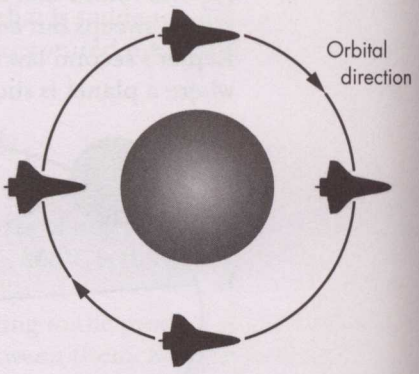


Figure 14-3: Spacecraft retain the same attitude in orbit unless forced to do otherwise.

You can blame this on Newton and his law of inertia, which states that an object at rest stays at rest and an object in motion stays in motion with the same speed and in the same direction unless acted upon by an unbalanced force.

Raising and Lowering Orbits

Brakes don't work in space, there's no friction, and inertia takes itself very seriously. To lower a spacecraft's orbit, you have to fire thrusters to reduce its velocity so that it falls farther into a planet's gravity well. To accomplish this, you have to *retrograde* your spacecraft so that its nose faces away from the present velocity vector—a fancy way of saying you have to fly tail-first. This assumes, of course, that the main thrusters are at the back of the spacecraft. Conversely, if you want to raise the orbit, you have to *prograde* the spacecraft, so that its nose will be pointed in the direction you are traveling. These two concepts are shown in Figure 14-4.

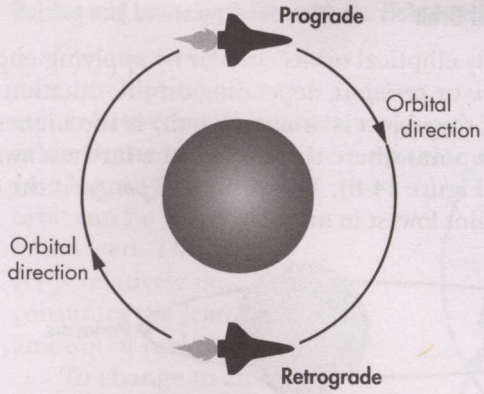


Figure 14-4: Prograde and retrograde are defined by the orientation of a spacecraft's nose with respect to the direction it is traveling around the body it is orbiting.

Taking the Inside Track

If you're chasing another spacecraft in orbit, do you speed up or slow down to catch it? According to Kepler's second law, you slow down. This will lower your orbit, resulting in a faster orbital velocity. Just as in horse racing, you want to take the inside track.

On the left side of Figure 14-5, two space shuttles are side by side in essentially the same orbit, traveling at the same velocity.

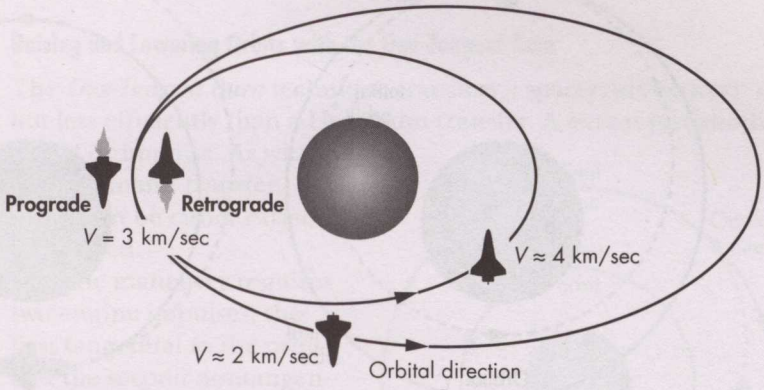


Figure 14-5: The orbital paradox: slow down to speed up!

The shuttle closest to the planet rotates 180 degrees and does a retrograde thrust to slow its immediate velocity. The outer shuttle performs a prograde thrust that increases its immediate velocity. They simultaneously stop thrusting, and the inner shuttle drops to a lower orbit while the outer shuttle transfers to a higher orbit. After an hour or so, the inner shuttle is traveling much faster, due to its closer proximity to the planet, and is well on its way to catch and lap the outer shuttle.

Circularizing an Elliptical Orbit

You can make highly elliptical orbits circular by applying engine impulses at either the *apoapsis* or *periapsis*, depending on the situation. The apoapsis (called the *apogee* if the object is orbiting Earth) is the highest point in an elliptical orbit—the point where the object is the farthest away from the body it is orbiting (Figure 14-6). The periapsis (*perigee* if the object's orbiting Earth) is the point lowest in an orbit.

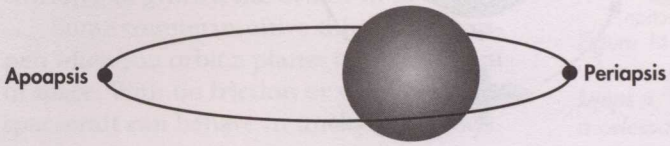


Figure 14-6: Location of the apoapsis and periapsis in an elliptical orbit

To raise the periapsis, the spacecraft performs a prograde thrust at the apoapsis (see the left-hand side of Figure 14-7). To lower the orbit while circularizing, the spacecraft must perform a retrograde thrust at the periapsis (see the right-hand side of Figure 14-7).

A somewhat counterintuitive part of this maneuver is that the initial orbit—that's the orbit that would have been—and the final, or actual, orbit will coincide at the point the engine impulse was applied.

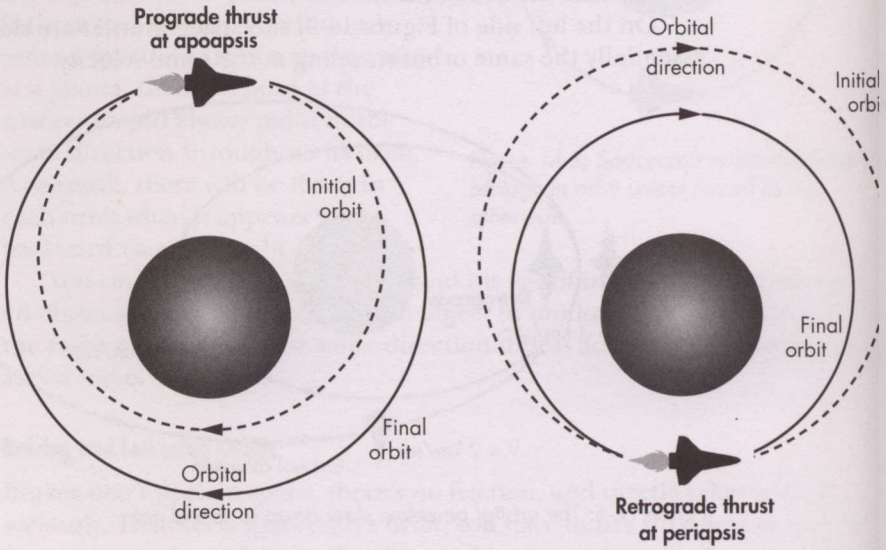


Figure 14-7: Circularizing and raising an orbit at apoapsis (left) and circularizing and lowering an orbit at periapsis (right)

Raising and Lowering Orbits with the Hohmann Transfer

A *Hohmann transfer orbit* uses an elliptical orbit to switch between two circular orbits in the same plane (see Figure 14-8). The orbit can be either raised or lowered. The maneuver is relatively slow, but it consumes the least possible amount of fuel.

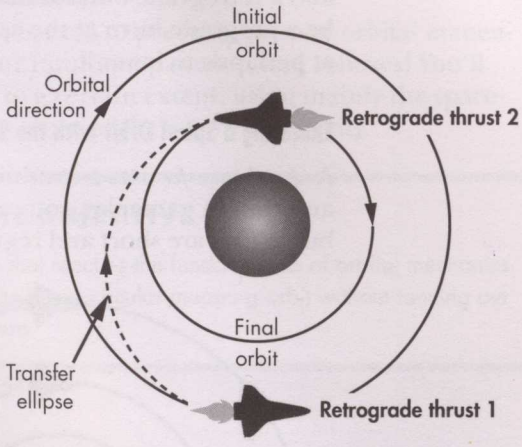


Figure 14-8: Transferring to a lower circular orbit with the Hohmann transfer technique

To change to an orbit with both a different periapsis *and* apoapsis, a spacecraft requires two engine impulses. One impulse moves the spacecraft onto the transfer orbit, and another moves it onto the final, destination orbit. When raising an orbit, the spacecraft applies the change in velocity in the direction of motion, and when lowering an orbit, it applies the change of velocity opposite to the direction of motion. The velocity changes have to occur at opposite sides of the orbit, as shown in Figure 14-8. Without the second thrust, the orbits will still intersect at the point of the first thrust, as shown on the right side of Figure 14-7.

Raising and Lowering Orbits with the One-Tangent Burn

The *One-Tangent Burn* technique transfers a spacecraft between orbits faster but less efficiently than a Hohmann transfer. A *burn* is just another term for thrust or impulse. As with the Hohmann transfer, orbits can be either raised or lowered.

The maneuver requires two engine impulses, the first tangential to the orbit and the second nontangential (see Figure 14-9). If the initial orbit is circular, as in the figure, then all points along it represent both the apoapsis and the periapsis, and the spacecraft can apply its first burn at any time.

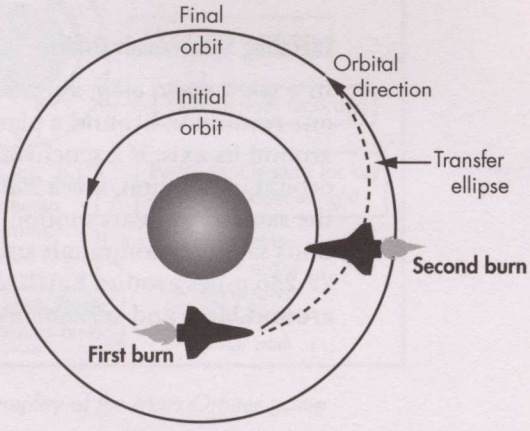


Figure 14-9: Transferring to a higher circular orbit with the One-Tangent Burn

Just as with the Hohmann transfer, a prograde burn raises the orbit, and a retrograde burn lowers it. If the orbit is elliptical, the first burn would be a prograde burn at the apoapsis to raise the orbit, or a retrograde burn at periapsis to lower it.

Executing a Spiral Orbit with the Spiral Transfer

A *spiral transfer* uses a continuous, low-thrust burn to change the size of an orbit. In gameplay, you can simulate this using retrograde or prograde burns that are short and regularly spaced, like those shown in Figure 14-10.

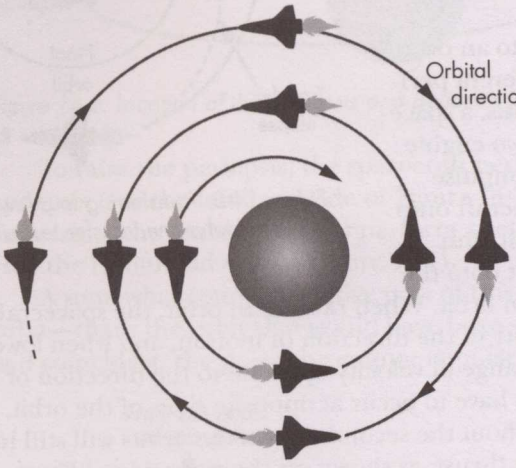


Figure 14-10: Executing a spiral orbit using short retrograde burns at regular intervals

To lower an orbit, all the burns must be retrograde; to raise an orbit, the spacecraft uses prograde burns.

Executing Synchronous Orbits

In a *synchronous orbit*, a spacecraft takes the same amount of time to make one revolution around a planet as it takes the planet to make one rotation around its axis. If a synchronous orbit is parallel to the equator, with no orbital inclination, it is a *stationary* orbit; to an observer on the orbited bod the satellite appears motionless in a fixed position in the sky. Communications satellites commonly use *geostationary* orbits, which have an altitude of 22,236 miles around Earth. A similar orbit would be called *aerostationary* around Mars and *selenostationary* around the moon.

Project #22: The Mars Orbiter Game

In real life, a series of equations is used to precisely execute orbital maneuvers. In gameplay, you'll use your intuition, patience, and reflexes! You'll also need to fly by instruments to a certain extent, using mainly the spacecraft's altitude readout and a measure of the orbit's circularity.

THE OBJECTIVE

Use pygame to build an arcade game that teaches the fundamentals of orbital mechanics. The game's goal is to nudge a satellite into a circular mapping orbit without running out of fuel or burning up in the atmosphere.

The Strategy

Start the design phase with a game sketch, as you did in Chapter 13. This sketch should capture all of the salient points of the game, like how it will look, how it will sound, how things will move, and how the game will communicate with the player (Figure 14-11).

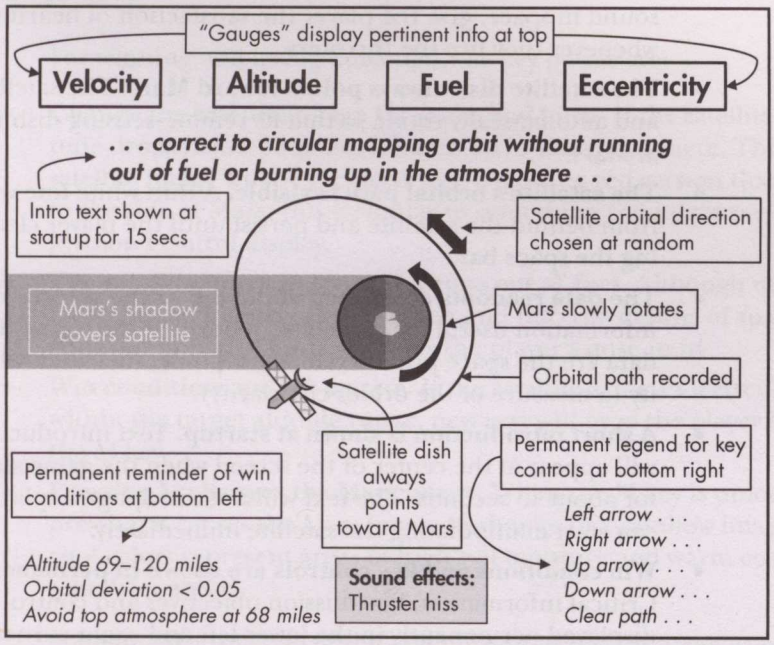


Figure 14-11: Sketch of the main gameplay of the Mars Orbiter game

The sketch in Figure 14-11 describes the main gameplay. You'll need a separate sketch to describe the win-lose conditions. For the main gameplay, the key points are:

- **The viewpoint is Mission Control.** The game screen should resemble a monitor at Mission Control from which the player can operate the errant space probe.
- **Mars is at front and center.** Everybody loves the Red Planet, so it will occupy the center of the jet-black screen.
- **Mars is animated.** The Martian globe will slowly rotate around its axis and cast a shadow. The satellite will dim appreciably when it passes through this shadow.
- **The satellite's initial orbit is chosen at random.** The satellite will appear at startup with a randomized—but constrained—orientation and velocity. On rare occasions, this may result in an instant game loss. That's still better than real missions, which fail 47 percent of the time!
- **There's no need to prograde or retrograde the satellite.** Constantly rotating the space probe before firing its thrusters greatly diminishes gameplay. Assume that attitudinal thrusters are arrayed around the fuselage and use the arrow keys to choose which thrusters to fire.
- **Firing thrusters causes an audible hiss.** Despite the fact that there's no sound in space, give the player the satisfaction of hearing a nice hiss whenever they fire the thrusters.
- **The satellite dish always points toward Mars.** The satellite will slowly and automatically rotate so that its remote-sensing dish is always aimed at Mars.
- **The satellite's orbital path is visible.** A thin white line will trail out from behind the satellite and persist until the player clears it by pressing the space bar.
- **The data readouts are placed at the top of the screen.** You will display information useful for gameplay in boxes at the top of the window. Key data are the space probe's velocity, altitude, fuel, and orbital eccentricity (a measure of the orbit's circularity).
- **A short introduction is shown at startup.** Text introducing the game will appear at the center of the screen when the game starts and stay up for about 15 seconds. The text will not disrupt gameplay, so the player can start manipulating the satellite immediately.
- **Win conditions and key controls are shown in permanent legends.** Critical information, like mission objectives and control keys, will be displayed permanently in the lower-left and -right corners of the screen.

The game sketch in Figure 14-12 describes what happens in success and failure cases. The player needs a reward when they win and an interesting outcome when they lose.

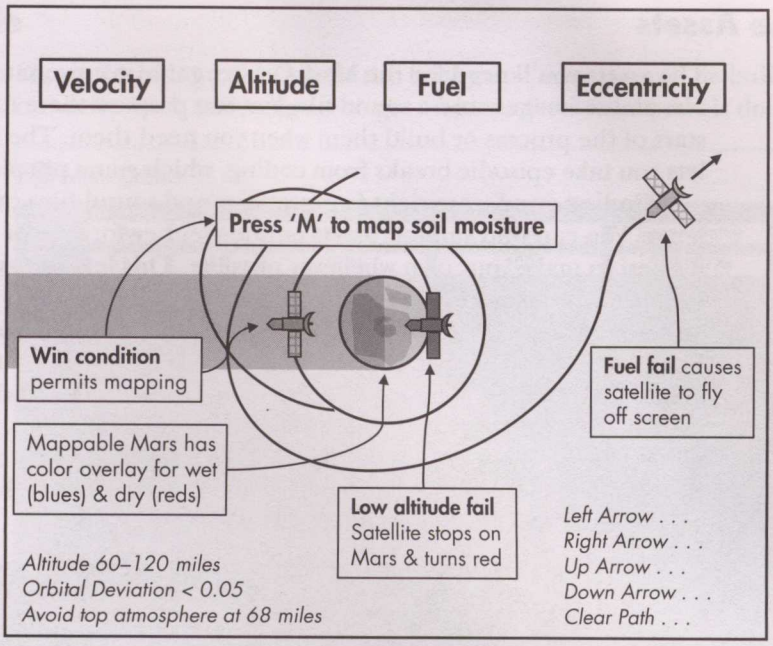


Figure 14-12: Game sketch of winning versus losing outcomes in the Mars Orbiter game

For winning and losing outcomes, the key points are:

- **Change the satellite image for crash and burn.** If the satellite's altitude drops below 68 miles, it burns up in the atmosphere. The moving satellite image will be replaced with a glowing red version that sticks to the side of Mars; this is similar to something you might see on a real Mission Control display.
- **The satellite is lost in space if it runs out of fuel.** Although unrealistic, have the satellite fly off the screen and into the depths of space if it runs out of fuel. This really rubs the player's nose in it!
- **Win conditions unlock a prize.** If the satellite achieves a circular orbit within the target altitude range, new text will urge the player to press the M key.
- **Pressing M changes the Mars image.** When the M key is unlocked, pressing it causes the Mars image to change to a rainbow image where cool colors represent areas of high soil moisture and warm colors represent drier areas.

For gameplay, the size of the satellite and its orbital speed won't be realistic, but the overall behavior will be correct. You should be able to correctly execute all of the orbital maneuvers described in "Astrodynamics for Gamers" on page 286.

Game Assets

The assets you'll need for the Mars Orbiter game are two satellite images, two planet images, and a sound file. You can prepare these together at the start of the process or build them when you need them. The latter approach lets you take episodic breaks from coding, which some people prefer.

Finding good, copyright-free graphics and sound files can be a challenge. You can find suitable assets online—either for free or for a fee—but it's best to make your own whenever possible. This lets you avoid any legal issues down the road.

The sprites (2D icons or images) I used for this project are shown in Figure 14-13. You need a satellite, a red "burned" version of the satellite, a view of Mars with a polar cap centered, and the same view with a colorful overlay that will represent mapped soil-moisture gradations. I found the satellite sprite at the free icon site AHA-SOFT (*http://www.aha-soft.com/*) and then copied and recolored it to make the crashed version. Both of the Mars sprites are NASA images modified for the game.

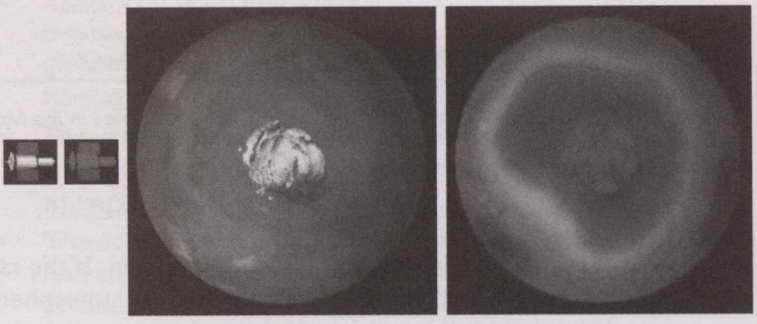


Figure 14-13: The satellite, crashed satellite, Mars, and Mars overlay images used as game sprites

I made a sound file for when the satellite is firing its thrusters using the white noise generator in the open source program Audacity. You can download a free copy of Audacity at *https://www.audacityteam.org/*. I saved the file in *Ogg Vorbis* format, an open source standard audio compression format that is free and works well with Python and `pygame`. You can use other formats, like MP3 and WAV, with `pygame`, but some have documented problems or have proprietary components that can raise legal issues if you try to commercialize your game.

You can download these files from this book's website at *https://www.nostarch.com/impracticalpython/* as *satellite.png*, *satellite_crash_40x33.png*, *mars.png*, *mars_water.png*, and *thrust_audio.ogg*. Download them, preserving the filenames, into the same folder as the code.

The Code

Figure 14-14 is an example of the final game screen you'll be building. You can refer back to this figure to get an idea of what the code is doing.

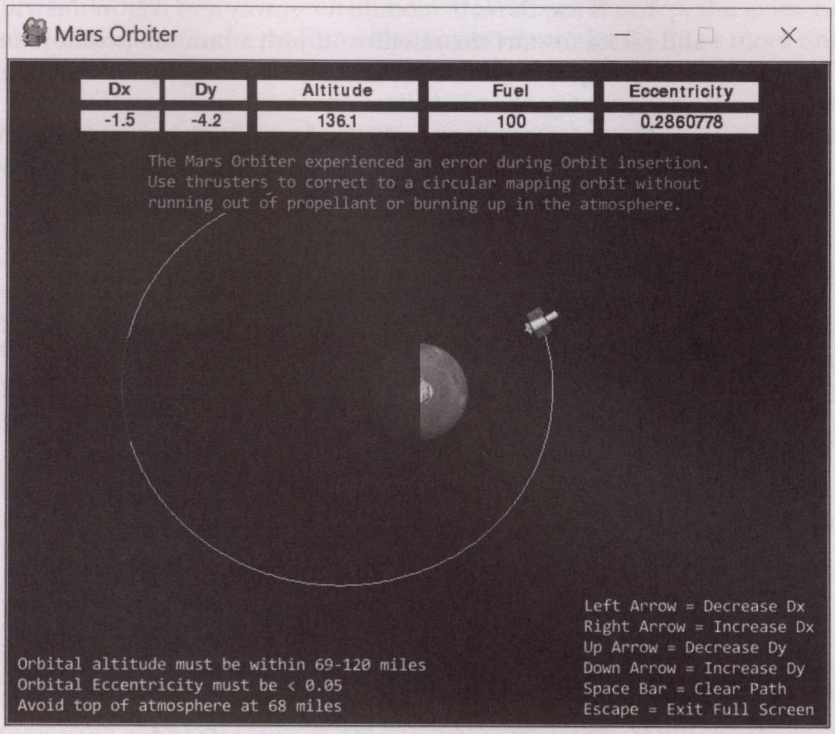


Figure 14-14: Example startup game screen for the final version of mars_orbiter.py

You can download the complete program (*mars_orbiter.py*) at *https://www.nostarch.com/impracticalpython/*.

Importing and Building a Color Table

Listing 14-1 imports the required modules and builds a color table.

mars_orbiter.py, part 1

```
❶ import os
  import math
  import random
  import pygame as pg

❷ WHITE = (255, 255, 255)
  BLACK = (0, 0, 0)
  RED = (255, 0, 0)
  GREEN = (0, 255, 0)
  LT_BLUE = (173, 216, 230)
```

Listing 14-1: Imports modules and builds a color table

First, import the operating system, designated by `os` ❶. The game will launch in full-screen mode, but the player will have the option of escaping out of full screen. This module will let you control the location of the game window after the player presses ESC.

You'll use the `math` module for gravity and trigonometric calculations and `random` to start the satellite off with a random position and velocity. Import `pygame` as you did in Chapter 13, using `pg`, rather than `pygame`, to reduce typing.

Finish by building an RGB color table ❷ as you did in Chapter 13. This lets you type in color names, rather than RGB-value tuples, when you need to assign one of these colors.

Defining the Satellite Class Initialization Method

Listing 14-2 defines the `Satellite` class and its initialization method, which you'll use to instantiate a satellite object in the game. Since this method definition is long, it's split over two listings.

mars_orbiter.py, part 2

```
❶ class Satellite(pg.sprite.Sprite):
      """Satellite object that rotates to face planet & crashes & burns."""

  ❷ def __init__(self, background):
      ❸ super().__init__()
      ❹ self.background = background
      ❺ self.image_sat = pg.image.load("satellite.png").convert()
        self.image_crash = pg.image.load("satellite_crash_40x33.png").convert
      ❻ self.image = self.image_sat
      ❼ self.rect = self.image.get_rect()
      ❽ self.image.set_colorkey(BLACK)  # sets transparent color
```

Listing 14-2: Defines the first part of the `Satellite` class initialization method

Define a class for a `Satellite` object ❶; if you need a refresher on object-oriented programming, read Chapter 11. Pass it the `pygame` `Sprite` class, as objects instantiated from the `Satellite` class will be sprites. As described in Chapter 13, `Sprite` is a built-in class that serves as a template for making sprites. Your new class will inherit features that your sprites will need from this base class. These include important attributes like `rect` and `image`, which you'll deal with shortly.

Next, define the `__init__()` method for the `Satellite` object ❷ and pass it `self`, which—by convention—is a special name within a class definition that refers to the current object. You also need to pass the method a `background` object. The satellite's path will be drawn on this object.

Inside the `__init__()` method, immediately invoke the initialization method for the built-in `Sprite` class using `super` ❸. This will initialize the sprite and establish the `rect` and `image` attributes it needs. With `super`, you don't need to refer to the base class (`Sprite`) explicitly. For more on `super`, see Listing 11-5 on page 229 or visit the docs at *https://docs.python.org/3/library/functions.html?highlight=super#super*.

Next, assign the background to self as an object attribute ❹. Then use pygame's image.load() method to load your two satellite images—one operational and one crashed—and in the same step, run the convert() method on them ❺. This converts the object into a graphic format that pygame can use efficiently once the game loop starts. Without this step, the game may slow noticeably as the png format is converted, on the fly, 30 or more times per second.

You'll use only one of the satellite images at a time, depending on whether or not the player burned up in the atmosphere, so use a generic self.image attribute to hold the loaded and converted image ❻. The unburned satellite image will be the default image; it will be replaced with the red crashed image if the satellite object gets too close to Mars.

Now, get the rectangle information for the image ❼. Remember that pygame places the sprites on rectangular surface objects, and it needs to know the dimensions and location of these rectangles as the game runs.

Finally, make the black parts of the satellite image invisible ❽. The satellite icon is on a field of black (see Figure 14-13), and you want the crashed-and-burned image to plot partially over Mars, so use the BLACK constant with the image object's colorkey() method in order to make the icon's background transparent. Otherwise, you'll see a black box with a red satellite overlapping the Red Planet. Note that if you want to type in the RGB equivalent for black, you need to enter it as a tuple: (0, 0, 0).

Setting the Satellite's Initial Position, Speed, Fuel, and Sound

Listing 14-3 completes the definition of the Satellite class initialization method. The satellite object's initial position and velocity are chosen at random from a limited range of choices; the orientation of the remote-sensing dish is initialized, the fuel tank topped off, and sound effects added.

mars_orbiter.py, part 3

```
❶ self.x = random.randrange(315, 425)
  self.y = random.randrange(70, 180)
❷ self.dx = random.choice([-3, 3])
❸ self.dy = 0
❹ self.heading = 0  # initializes dish orientation
❺ self.fuel = 100
  self.mass = 1
  self.distance = 0  # initializes distance between satellite & planet
❻ self.thrust = pg.mixer.Sound('thrust_audio.ogg')
❼ self.thrust.set_volume(0.07)  # valid values are 0-1
```

Listing 14-3: Completes the Satellite class initialization method by initializing parameters

When the game starts, the satellite will appear at a random point near the top of the screen. You'll choose the exact location from a range of x- and y-values ❶.

You'll also choose the satellite's velocity at random, but it will be slow enough that the satellite can't escape from orbit. Randomly set the velocity to either –3 or 3. Negative values result in a counterclockwise orbit, and vice versa. Use the delta-x (dx) attribute only ❷ and let gravity take care of dy. As

discussed in Chapter 13, pygame moves sprites around the screen using incremental changes in the x-location (called delta-x or *dx*) and incremental changes in the y-location (called delta-y or *dy*). These vector components are calculated and added to the sprite's current position (`self.x`, `self.y`) with each game loop.

Next, set the `dy` attribute to `0` ❸. Later, the `gravity()` method will establish an initial `dy` value when it accelerates the newly instantiated satellite downscreen toward the planet.

Assign an attribute for the satellite's heading ❹. The remote-sensing dish, which will read soil moisture on the planet's surface, should always point toward Mars, and if you remember from Figure 14-3, this won't occur unless you overcome inertia. You'll use a method to actually rotate the satellite, so for now, just initialize the `heading` attribute with `0`.

Now, top off the fuel tank with 100 units of fuel ❺. If you want to relate this to real life, it would probably represent 100 kilograms of hydrazine, similar to what was used in the *Magellan* probe that mapped Venus.

Next, set the object's mass to `1`. This basically means you'll just use the mass of Mars in the gravity equation, because you multiply the masses of two objects together. As stated earlier, the pull of the satellite on Mars is inconsequential, so you don't need to calculate it. The satellite's `mass` attribute is included for completeness and as a placeholder in case you want to experiment with different values later.

The following `distance` attribute stores the distance between the satellite and the body it is orbiting. The actual value will be calculated by a method you'll define later.

It's time to add sound effects. You'll initialize pygame's sound mixer in the `main()` function, but for now, name a `thrust` attribute for the thrusting sound effect ❻. Pass the mixer's `Sound` class the short clip of white noise in Ogg Vorbis format (*.ogg*). Finally, set the playback volume, using values between 0 and 1 ❼. You may need to calibrate this to your PC. Ideally, you want a value that every player will be able to at least *hear* and then fine-tune with their own computer's volume control.

Firing Thrusters and Checking for Player Input

Listing 14-4 defines the `thruster()` and `check_keys()` methods of the `Satellite` class. The first determines the actions taken if one of the satellite's thrusters is fired. The second checks whether a player has interacted with the thrusters by pressing an arrow key.

mars_orbiter.py, part 4

```
❶ def thruster(self, dx, dy):
      """Execute actions associated with firing thrusters."""
   ❷ self.dx += dx
     self.dy += dy
   ❸ self.fuel -= 2
   ❹ self.thrust.play()

❺ def check_keys(self):
      """Check if user presses arrow keys & call thruster() method."""
   ❻ keys = pg.key.get_pressed()
```

```
        # fire thrusters
❼       if keys[pg.K_RIGHT]:
❽           self.thruster(dx=0.05, dy=0)
        elif keys[pg.K_LEFT]:
            self.thruster(dx=-0.05, dy=0)
        elif keys[pg.K_UP]:
            self.thruster(dx=0, dy=-0.05)
        elif keys[pg.K_DOWN]:
            self.thruster(dx=0, dy=0.05)
```

Listing 14-4: Defines the `thruster()` and `check_keys()` methods for the `Satellite` class

The `thruster()` method takes `self`, `dx`, and `dy` as arguments ❶. The last two arguments, which can be positive or negative, are immediately added to the satellite's `self.dx` and `self.dy` velocity components ❷. Next, the fuel level is decreased by two units ❸. Altering this value is one way to make the game either harder or easier. Finish by calling the `play()` method on the `thrust` audio attribute to make the hissing sound ❹. Note that, instead of *returning* values, OOP methods *update* existing object attributes.

The `check_keys()` method takes `self` as an argument ❺. First you use the `pygame key` module to determine whether the player has pressed a key ❻. The `get_pressed()` method returns a tuple of Boolean values—`1` for `True` and `0` for `False`—that represent the current state of each key on the keyboard. `True` means a key has been pressed. You can index this tuple by using the key constants. You can find a list of all the keyboard constants at *https://www.pygame.org/docs/ref/key.html*.

For example, the right arrow key is `K_RIGHT`. If this key has been pressed ❼, call the `thruster()` method and pass it `dx` and `dy` values ❽. In `pygame`, x-values increase toward the right of the screen, and y-values increase toward the bottom of the screen. So, if the user presses the left arrow key, subtract from `dx`; likewise, if the up arrow is pressed, decrement the `dy` value. The right arrow will increase `dx`, and the down arrow will increase `dy`. Readouts at the top of the screen will help the player relate the satellite's movements to the underlying `dx` and `dy` values (see Figure 14-14).

Locating the Satellite

Still in the `Satellite` class, Listing 14-5 defines the `locate()` method. This method calculates the distance of the satellite from the planet and determines the heading for pointing the dish at the planet. You'll use the distance attribute later when calculating the force of gravity and the *eccentricity* of the orbit. Eccentricity is a measurement of the deviation of an orbit from a perfect circle.

ars_orbiter.py,
rt 5

```
❶   def locate(self, planet):
        """Calculate distance & heading to planet."""
❷       px, py = planet.x, planet.y
❸       dist_x = self.x - px
        dist_y = self.y - py
        # get direction to planet to point dish
❹       planet_dir_radians = math.atan2(dist_x, dist_y)
```

```
❺ self.heading = planet_dir_radians * 180 / math.pi
❻ self.heading -= 90  # sprite is traveling tail-first
❼ self.distance = math.hypot(dist_x, dist_y)
```

Listing 14-5: Defines the `locate()` method for the `Satellite` class

To locate the satellite, you need to pass the `locate()` method the `satellite` (`self`) and `planet` objects ❶. First, determine the distance between the objects in x-y space. Get the planet's x- and y-attributes ❷; then subtract them from the satellite's x- and y-attributes ❸.

Now, use these new distance variables to calculate the angle between the satellite's heading and the planet so you can rotate the satellite dish toward the planet. The `math` module uses radians, so assign a local variable called `planet_dir_radians` to hold the direction in radians and pass `dist_x` and `dist_y` to the `math.atan2()` function to calculate the arc tangent ❹. Since `pygame` uses degrees (sigh), convert the angle from radians to degrees using the standard formula; alternatively, you could use `math` to do this, but sometimes it's good to see the man behind the curtain ❺. This should be a sharable attribute of the satellite object, so name it `self.heading`.

In `pygame`, the front of a sprite is to the east by default, which means the satellite sprite is orbiting tail-first (see the satellite icon in Figure 14-13). To get the dish to point toward Mars, you need to subtract 90 degrees from the heading, because negative angles result in *clockwise* rotation in `pygame` ❻. This maneuver will use none of the player's fuel allotment.

Finally, get the Euclidian distance between the satellite and Mars by using the `math` module to calculate the hypotenuse from the x- and y-components ❼. You should make this an attribute of the satellite object since you will use it later in other functions.

NOTE *In real life, there are multiple ways to keep the dish of a satellite pointed toward a planet without expending large amounts of fuel. Techniques include slowly tumbling or spinning the satellite, making the dish end heavier than the opposite end, using magnetic torque, or using internal flywheels—also known as reaction wheels or momentum wheels. Flywheels use electric motors that can be powered by solar panels, eliminating the need for heavy and toxic liquid propellant.*

Rotating the Satellite and Drawing Its Orbit

Listing 14-6 continues the `Satellite` class by defining methods for rotating the satellite dish toward the planet and drawing a path behind it. Later, in the `main()` function, you'll add code that lets the player erase and restart the path by pressing the space bar.

mars_orbiter.py, part 6

```
❶ def rotate(self):
      """Rotate satellite using degrees so dish faces planet."""
   ❷ self.image = pg.transform.rotate(self.image_sat, self.heading)
   ❸ self.rect = self.image.get_rect()

❹ def path(self):
```

```
        """Update satellite's position & draw line to trace orbital path."""
    ❺ last_center = (self.x, self.y)
    ❻ self.x += self.dx
        self.y += self.dy
    ❼ pg.draw.line(self.background, WHITE, last_center, (self.x, self.y))
```

Listing 14-6: Defines the rotate() and path() methods of the Satellite class

The `rotate()` method will use the `heading` attribute, which you calculate in the `locate()` method, to turn the satellite dish toward Mars. Pass `self` to `rotate()` ❶, which means `rotate()` will automatically take the name of the satellite object as an argument when it is called later.

Now, rotate the satellite image using `pygame`'s `transform.rotate()` method ❷. Pass it the original image followed by the `heading` attribute; assign these to the `self.image` attribute so you don't degrade the original master image. You'll need to transform the image with each game loop, and transforming an image rapidly degrades it. So always keep a master image and work off a new copy every time you do a transformation.

End the function by getting the transformed image's `rect` object ❸.

Next, define a method called `path()` and pass it `self` ❹. This will draw a line marking the satellite's path, and since you need two points to draw a line, assign a variable to record the satellite's center location as a tuple prior to moving it ❺. Then increment the x- and y-locations with the `dx` and `dy` attributes ❻. Finish by using `pygame`'s `draw.line()` method to define the line ❼. This method needs a drawing object, so pass it the `background` attribute, followed by the line color and the previous and current x-y location tuples.

Updating the Satellite Object

Listing 14-7 updates the satellite object and completes the class definition. Sprite objects almost always have an `update()` method that is called once per frame as the game runs. Anything that happens to the sprite, such as movement, color changes, user interactions, and so on, is included in this method. To keep them from becoming too cluttered, `update()` methods mostly call other methods.

's_orbiter.py,
17

```
❶ def update(self):
        """Update satellite object during game."""
    ❷ self.check_keys()
    ❸ self.rotate()
    ❹ self.path()
    ❺ self.rect.center = (self.x, self.y)
        # change image to fiery red if in atmosphere
    ❻ if self.dx == 0 and self.dy == 0:
            self.image = self.image_crash
            self.image.set_colorkey(BLACK)
```

Listing 14-7: Defines the update() method for the Satellite class

Start by defining the `update()` method and passing it the object, or `self` ❶. Next, call the methods that you defined earlier. The first of these checks

for player interactions made through the keyboard ❷. The second rotates the satellite object so that the dish keeps pointing toward the planet ❸. The final method updates the satellite's x-y location and draws a path behind it so you can visualize the orbit ❹.

The program needs to keep track of the satellite sprite's location as it orbits Mars, so assign a `rect.center` attribute and set it to the satellite's current x-y location ❺.

The final bit of code changes the satellite image in the event the player crashes and burns in the atmosphere ❻. The top of the Martian atmosphere is about 68 miles above its *surface*. For reasons I'll explain later, assume that an altitude value of 68—which is measured in pixels from the *center* of the planet—equates to the top of the atmosphere. If the satellite dips below this altitude during gameplay, the `main()` function will set its velocity—represented by `dx` and `dy`—to `0`. Check that these values are both `0`, and if so, change the image to `image_crash` and set its background to transparent (as you did previously for the main satellite image).

Defining the Planet Class Initialization Method

Listing 14-8 defines the `Planet` class, which you'll use to instantiate a `planet` object.

mars_orbiter.py, part 8

```
❶ class Planet(pg.sprite.Sprite):
      """Planet object that rotates & projects gravity field."""

❷   def __init__(self):
        super().__init__()
❸       self.image_mars = pg.image.load("mars.png").convert()
        self.image_water = pg.image.load("mars_water.png").convert()
❹       self.image_copy = pg.transform.scale(self.image_mars, (100, 100))
❺       self.image_copy.set_colorkey(BLACK)
❻       self.rect = self.image_copy.get_rect()
        self.image = self.image_copy
❼       self.mass = 2000
❽       self.x = 400
        self.y = 320
        self.rect.center = (self.x, self.y)
❾       self.angle = math.degrees(0)
        self.rotate_by = math.degrees(0.01)
```

Listing 14-8: Begins definition of the `Planet` class

You are probably very familiar with the initial steps to creating the `Planet` class by now. First, you name the class with a capital letter, then pass it the `Sprite` class so it will conveniently inherit features from this built-in `pygame` class ❶. Next, you define an `__init__()`, or initialization, method for your `planet` object ❷. Then you call the `super()` initialization method, as you d[o] for the `Satellite` class.

Load the images as attributes and convert them to `pygame`'s graphic format at the same time ❸. You need both the normal Mars image and t[he]

one for mapped soil moisture. You were able to use the satellite sprite at its native size, but the Mars image is too large. Scale the image to 100 pixels × 100 pixels ❹ and assign the scaled image to a new attribute so repeated transformations won't degrade the master image.

Now, set the transformed image's transparent color to black, as you did earlier with the satellite image ❺. Sprites in `pygame` are all "mounted" on rectangular surfaces, and if you don't make black invisible, the corners of the planet surface may overlap and cover the white-colored orbital path drawn by the satellite (see Figure 14-15).

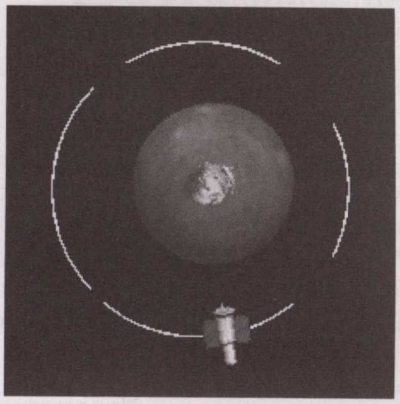


Figure 14-15: Corners of Mars `rect` covering orbital path

As always, get the sprite's `rect` object ❻. There's another transformation to come, so copy the image attribute again and assign it the logical name of `self.image`.

To apply the force of gravity, the planet needs mass, so name a `mass` attribute and assign it a value of `2000` ❼. Earlier, you assigned the satellite a mass of `1`; this means that Mars is only 2,000 times as massive as a satellite! That's okay, because you aren't working in real-world units, and the time and distance scales differ from reality. If you scale distances so that the satellite is only a few hundred pixels from Mars, you have to scale gravity as well. Despite this, the satellite will still behave realistically with respect to gravity.

The planet's mass value was determined through experimentation. To scale the force of gravity, you can either change this mass value or use the gravitational constant (`G`) variable later.

Set the `planet` object's `x` and `y` attributes to the center point of the screen—you'll use a screen size of 800 × 645 in the `main()` function—and assign these values to the `rect` object's center ❽.

Finally, assign the attributes you'll need to slowly rotate Mars about its axis ❾. You'll use the same `transform.rotate()` method you used to turn the satellite, so you need to create an `angle` attribute. Then, use a `rotate_by` attribute to assign the increment—in degrees—by which this rotation angle changes with each game loop.

Rotating the Planet

Listing 14-9 continues the `Planet` class by defining its `rotate()` method. This method rotates the planet around its axis, making small changes with each game loop.

mars_orbiter.py, part 9

```
❶ def rotate(self):
      """Rotate the planet image with each game loop."""
    ❷ last_center = self.rect.center
    ❸ self.image = pg.transform.rotate(self.image_copy, self.angle)
      self.rect = self.image.get_rect()
    ❹ self.rect.center = last_center
    ❺ self.angle += self.rotate_by
```

Listing 14-9: Defines a method to rotate the planet around its axis

The `rotate()` method also takes the object as an argument ❶. As the square Mars image is rotating, the bounding rectangle object (`rect`) remains stationary and must expand to accommodate the new configuration (see Figure 14-16). This change in size can affect the center point of the `rect`, so assign a `last_center` variable and set it to the planet's current center point ❷. If you don't do this, Mars will wobble around its axis as the game runs.

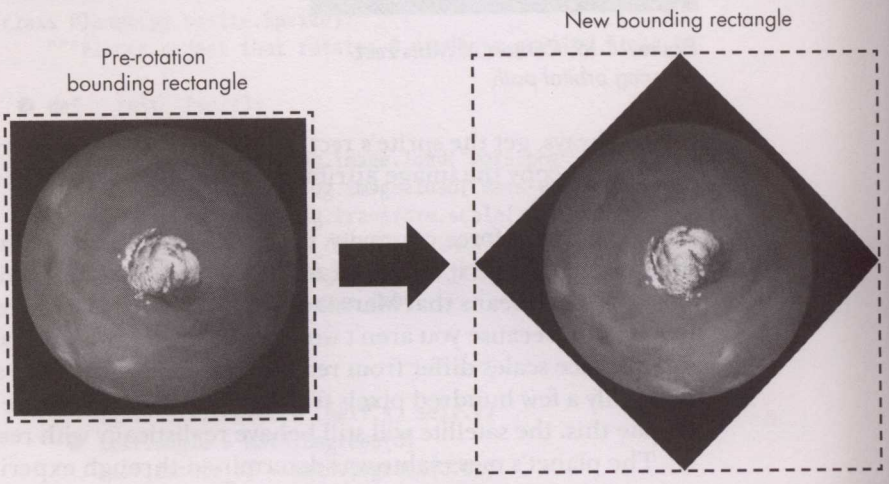


Figure 14-16: The bounding rectangle changes size to accommodate rotating images.

Next, rotate the copied image using `pygame`'s `transform.rotate()` method and assign it to the `self.image` attribute ❸; you need to pass the method the copied image and the `angle` attribute. Immediately after rotating, reset the image's `rect` attribute and move its center location back to `last_center` in order to mitigate any shifting of `rect` that occurred during rotation ❹.

When the `planet` object is instantiated, the angle attribute will start at 0 degrees, then increase by 0.1—assigned in the `rotate_by` attribute—with each frame ❺.

Defining the gravity() and update() Methods

Listing 14-10 completes the `Planet` class by defining the `gravity()` and `update()` methods. In Chapter 13, you treated gravity as a constant applied in the y-direction. The method applied here is slightly more sophisticated, because it takes into account the distance between two objects.

mars_orbiter.py, part 10

```
❶ def gravity(self, satellite):
      """Calculate impact of gravity on satellite."""
   ❷ G = 1.0  # gravitational constant for game
   ❸ dist_x = self.x - satellite.x
      dist_y = self.y - satellite.y
      distance = math.hypot(dist_x, dist_y)
      # normalize to a unit vector
   ❹ dist_x /= distance
      dist_y /= distance
      # apply gravity (dx & dy represent pixels/frame)
   ❺ force = G * (satellite.mass * self.mass) / (math.pow(distance, 2))
   ❻ satellite.dx += (dist_x * force)
      satellite.dy += (dist_y * force)

❼ def update(self):
      """Call the rotate method."""
      self.rotate()
```

Listing 14-10: Defines the gravity() and update() methods of the Planet class

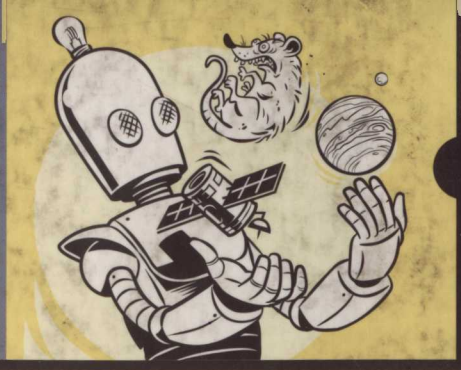
Define the `gravity()` method and pass it `self` and the satellite object ❶. You're still in the `Planet` class, so `self` here represents Mars.

Start by naming a local variable `G`; an uppercase G is the *universal gravitational constant*, also known as the *constant of proportionality* ❷. In real life, this is a very small, empirically derived number, which is basically a conversion number to get all the units to work out correctly. You're not using real-world units in the game, so set this to `1`; this way, it won't have an impact on the gravity equation. During game development, you can tweak this constant up or down to fine-tune the force of gravity and its effect on orbiting objects.

You need to know how far apart the two objects are, so get their distance in the x-direction and the y-direction ❸. Then, use the `math` module's `hypot()` method to get the Euclidian distance. This will represent the r in the gravity equation.

Since you're going to directly address the *magnitude* of the distance between the satellite and Mars in the gravity equation, all you need from the distance vector is *direction*. So, divide `dist_x` and `dist_y` by `distance` to "normalize" the vector to a unit vector with a magnitude of `1` ❹. You are basically dividing the length of each side of a right triangle by its hypotenuse. This preserves the vector's direction, represented by the relative differences in `dist_x` and `dist_y`, but sets its magnitude to `1`. Note that if you don't perform this normalization step, the results will be unrealistic but interesting (see Figure 14-17).

MODEL, EXPERIMENT, SIMULATE, PLAY



COVERS PYTHON 3

Python undeniably makes programming easier than ever to learn. But once you understand the basics, what do you do next? Maybe you just need some inspiration for your next weekend project.

With *Impractical Python Projects*, you'll explore the farthest reaches of the galaxy, the souls of poets, the world of high finance, the trickery of spies, and more—using modules like tkinter, matplotlib, cProfile, Pylint, pygame, pillow, and python-docx.

Follow along and flex your problem-solving skills to:

- Help James Bond crack a high-tech safe with a hill-climbing algorithm
- Write poems using Markov chain analysis
- Breed a race of gigantic rats with genetic algorithms
- Plan a secure retirement with a Monte Carlo simulation
- Model the Milky Way and calculate our odds of detecting alien civilizations
- Map Mars and learn orbital mechanics with your own personal space probe

And so much more! Whether you're looking to pick up some new Python skills or just want some creative programming exercises, you'll find endless educational and geeky fun with *Impractical Python Projects*.

ABOUT THE AUTHOR

Lee Vaughan is a programmer, pop culture enthusiast, and educator. Lee's professional work involves the construction and review of computer models; the development, testing, and commercialization of software; and the training of geoscientists and engineers. He wrote *Impractical Python Projects* to help self-learners hone their Python skills and have fun doing it!

no starch press

THE FINEST IN GEEK ENTERTAINMENT™

www.nostarch.com

MIX
Paper from responsible sources
FSC® C008955
www.fsc.org

"I LIE FLAT."
This book uses a durable binding that won't snap shut.

$29.95 ($39.95 CDN)

ISBN: 978-1-59327-890-8

9 781593 278908 52995

SHELVE IN: PROGRAMMING LANGUAGES/PYTHON