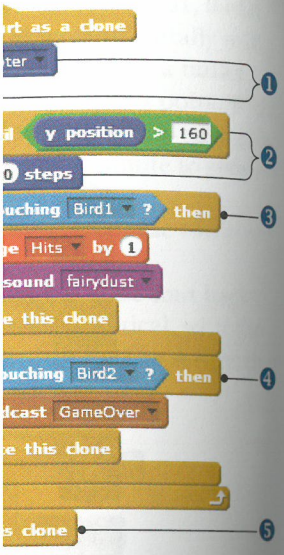


8: The startup handler of a
llet

of the Stage without touch-
exits ❺, and the clone is
time the bullet moves. If
ot increases the Hits variable
ing. On the other hand,
sts GameOver to signal the
ted since it has finished

ould add many features to it.

and keep score based on the

erent speeds. Reward the
rds.

dify the game with some of
th a few of your own and

Free-Fall Simulation

FreeFall.sb2

In this section, I'll present an application that simulates the motion of a falling object. Ignoring the effects of buoyancy and air resistance, when an object at rest is dropped from some height, the distance d (in meters) fallen by the object during time t (in seconds) is given by $d = \frac{1}{2} gt^2$, where $g = 9.8$ m/s^2 is the gravitational acceleration. The goal of this simulation is to show the position of the falling object at times 0.5 s, 1.0 s, 1.5 s, 2.0 s, and so on, until the object reaches the ground. The interface for this simulation is shown in Figure 7-29.

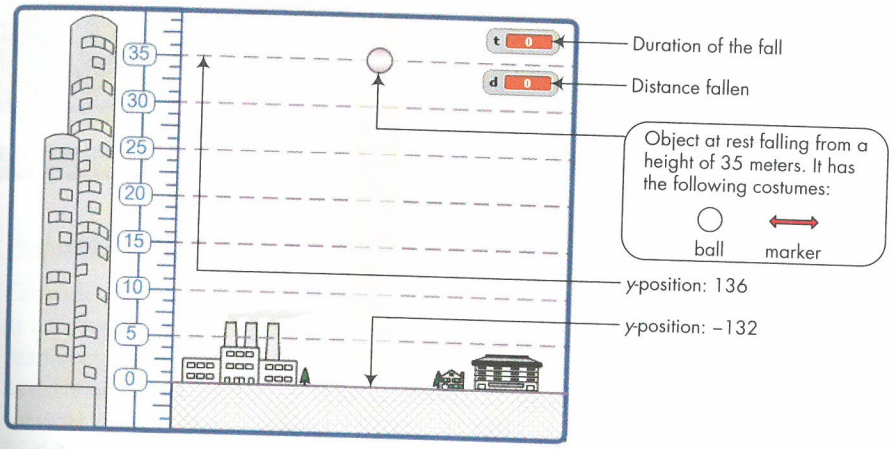


Figure 7-29: User interface for the free-fall simulation

An object at rest (the ball in the figure) will be allowed to fall from a height of 35 m. A simple substitution in the above formula shows that the object will reach the ground after $t = \sqrt{(2 \times 35) / 9.8} = 2.67$ s. The application has one sprite (called Ball) that has the two costumes shown in the figure. When it is time to show the position of the falling ball, the sprite changes momentarily to the marker costume, makes a stamp, and switches back to the ball costume.

The simulation starts when the green flag is clicked. In response, the Ball sprite runs the script shown in Figure 7-30.

During initialization ❶, the sprite moves to its starting position, switches to the ball costume, clears its voice bubble from the previous run, and clears the Stage from any previous stamps. It then initializes t and counter to 0. The variable t represents the duration of the fall, and counter keeps track of the number of loop repetitions.

The script then enters an infinite loop ❷ to calculate the simulation parameters at different time intervals. It performs those calculations and updates the ball's position every 0.05 s ❸ to ensure the ball's smooth movement. Every 0.05 s, the value of the time variable t is updated, and the distance the ball has fallen (d) is calculated. The value of the counter variable is also incremented by 1.

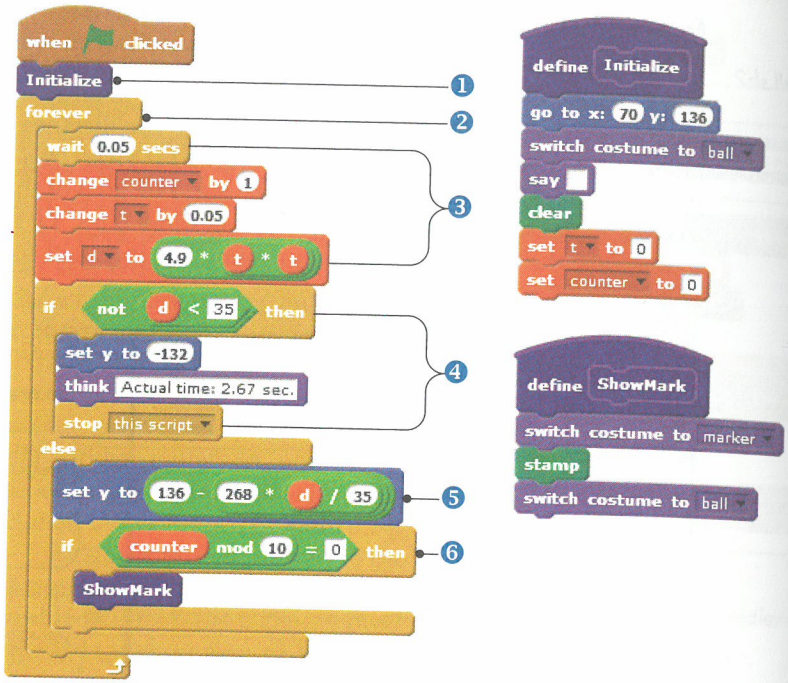


Figure 7-30: Script for the Ball *sprite in the free-fall simulation*

If the ball reaches the ground (which happens at d ≥ 35), the script sets the ball's *y*-position to that of the ground, displays the actual duration of the journey, and stops the script to end the simulation ❹.

Otherwise, the script sets the vertical position of the ball in accordance with the fallen distance ❺. Since a height of 35 m corresponds to 268 pixels on the Stage (see Figure 7-29), a distance of *d* meters corresponds to 268 * (d / 35). The final *y*-position is established by subtracting this number from the initial *y*-position, which is 136.

Since the iteration duration is 0.05 s, it takes 10 iterations to get 0.5 s. Thus, when the counter becomes 10, 20, 30, and so on, the Ball sprite switches to (and stamps) the marker costume to show the position of the falling ball at those instants ❻.

Figure 7-31 illustrates the result of running this simulation. Note how the distance fallen in each time interval increases as the object falls. Because of gravity, the ball accelerates—its velocity increases—at a rate of 9.8 m/s^2.

TRY IT OUT 7-9

Open the application and run it to understand how it works. Try converting the simulation into a game in which players drop the ball to hit a moving object on the ground. You can add a score, change the speed of the target, or even set the action on another planet (change the gravitational acceleration).

```
define Initialize
go to x: 70 y: 136
switch costume to ball
say
clear
set t to 0
set counter to 0
```

```
define ShowMark
switch costume to marker
stamp
switch costume to ball
```

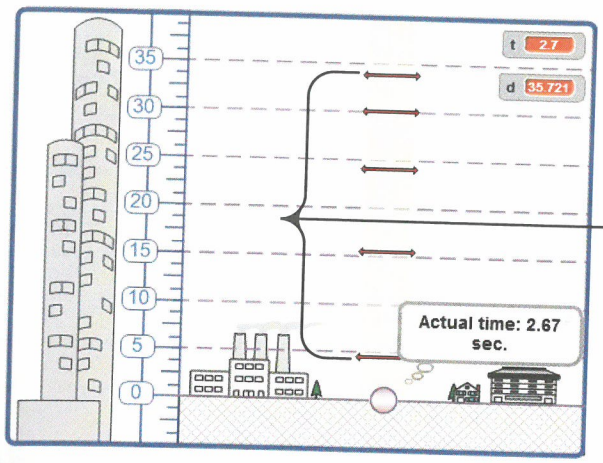
simulation

appens at d ≥ 35), the script
ıd, displays the actual duration
he simulation ❹.
osition of the ball in accor-
eight of 35 m corresponds to
distance of *d* meters corre-
ı is established by subtracting
ch is 136.
takes 10 iterations to get 0.5 s.
), and so on, the Ball sprite
e to show the position of the

ıing this simulation. Note
l increases as the object falls.
velocity increases—at a rate

-9

ow it works. Try converting the
ball to hit a moving object on
eed of the target, or even set the
al acceleration).



These markers indicate the ball's position at times 0.5, 1.0, 1.5, 2.0, and 2.5 seconds.

Figure 7-31: Output of the free-fall simulation

Projectile Motion Simulator

Projectile.sb2

Consider a ball fired at some initial velocity (v_0) from a cannon that points at an angle θ from the horizontal. You can analyze the ball's trajectory by resolving the velocity vector (v_0) into its horizontal and vertical components at different times. The horizontal component remains constant, but the vertical component is affected by gravity. When the motions corresponding to these two components are combined, the resulting path is a parabola. Let's examine the equations that govern projectile motion (neglecting air resistance).

The origin of our coordinate system is the point at which the ball begins its flight, so the ball's *x*-coordinate at any time, *t*, is given by $x(t) = v_{0x}t$, and the *y*-coordinate is $y(t) = v_{0y}t - (0.5)gt^2$, where $v_{0x} = v_0 \cos\theta$ is the *x*-component of v_0; $v_{0y} = v_0 \sin\theta$ is the *y*-component of v_0; and $g = 9.8$ m/s^2 is the gravitational acceleration. Using these equations, we can calculate the total flight time, the maximum height, and the horizontal range of the ball. The equations for these quantities are shown in Figure 7-32.

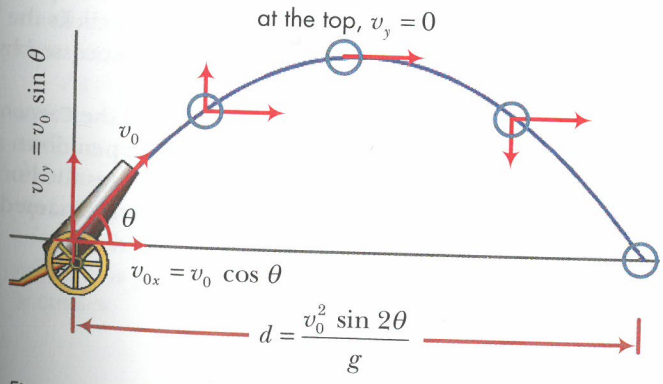


Figure 7-32: Parabolic trajectory of a ball

from 0 to 23. We need the Hour hand to point toward 0° (that is, up) for hour 0, 30° for hour one, 60° for hour two, and so on, as illustrated in the figure. Of course, if the current time is, let's say, 11:50, we don't want the Hour hand to point exactly at 11 but rather more toward 12. We can make this adjustment by taking the current minutes into account.

Since every hour (or 60 minutes) corresponds to 30° on the face of the clock, every minute is worth 2°. Therefore, every minute, we need to adjust the angle of the Hour hand by the current number of minutes divided by 2, as shown in the script.

The script for the Time sprite is trivial and isn't shown here. It uses nested **join** blocks to construct a display string of the form *hour*:*min*:*sec* and shows this string in a think bubble, as shown in Figure 7-20.

TRY IT OUT 7-7

Open the application and run it. Change the script for the Min sprite to make it move smoothly, instead of jumping every minute. (Hint: Use the same idea we applied to smooth the movement of the hour hand.) Also, change the script of the Time sprite to display a string of the form "3:25:00 PM" (12-hour format) instead of "15:25:00" (24-hour format). Think of other ways to enhance the application and try to implement them as well.

Bird Shooter Game

BirdShooter.sb2

Now, let's make a simple game that uses most of the blocks we introduced in this chapter. The player's goal will be to knock two birds out of the sky, and you can see the user interface in Figure 7-23.

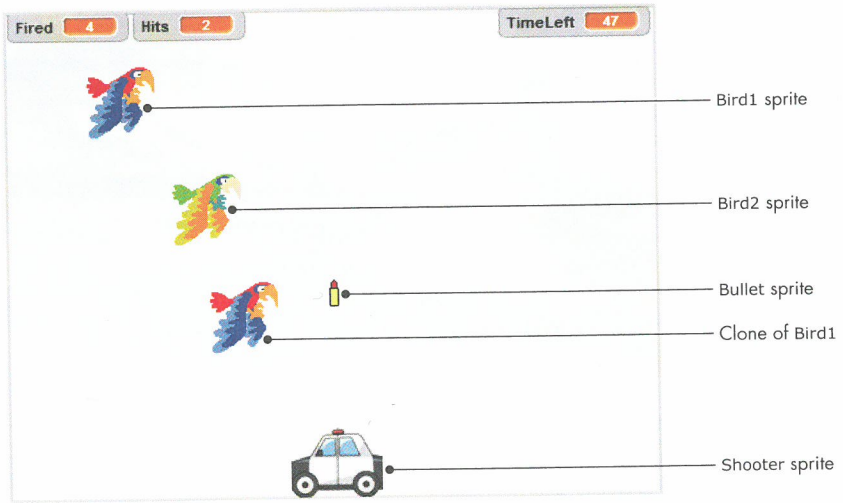


Figure 7-23: User interface of the bird shooter game

As shown, the game contains five sprites: Bird1, a clone of Bird1, Bird2, a shooter, and a bullet. The player can move the shooter horizontally using the left and right keyboard arrows. Pressing the spacebar fires a bullet into the sky. If the bullet hits Bird1 or its clone, the player gets a point. Bird2 is an endangered species, so the player isn't allowed to shoot that one; if the bullet hits that sprite, the game ends. The player has one minute to shoot as many birds as possible.

Each bird uses two costumes. When switching between these two costumes, the birds appear to be flapping their wings.

The Stage has two backgrounds named start and end. The start background is shown in Figure 7-23. The end background is identical, with the addition of the words *Game Over* to the center of the image. The scripts that belong to the Stage are shown in Figure 7-24.

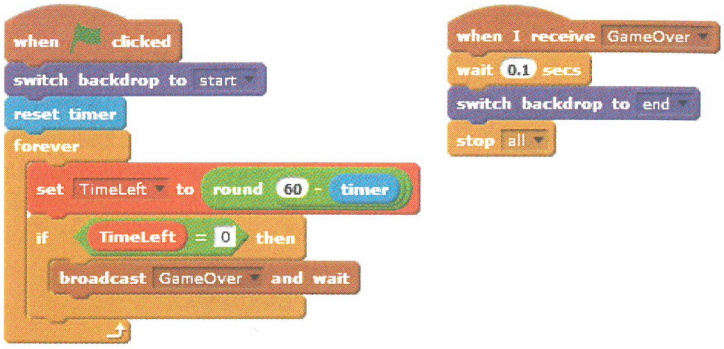


Figure 7-24: The scripts for the Stage in the bird shooter game

When the green flag icon is pressed, the Stage switches to the start background, resets the timer, and starts a loop that updates and checks the remaining game time, which is tracked by the TimeLeft variable. When TimeLeft reaches 0 or when the Stage receives the GameOver broadcast message, it executes the GameOver handler. This script waits for a short time to allow the birds to hide themselves, switches to the end backdrop, and calls **stop all** to end any running scripts. As you'll see soon, the GameOver message will be sent by the Bullet sprite when it hits Bird2. Let's now take a look at the script for the Shooter sprite, shown in Figure 7-25.

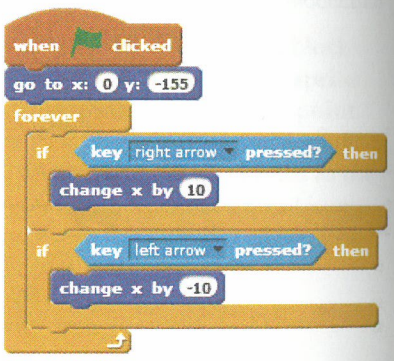


Figure 7-25: The script for the Shooter *sprite*

1, a clone of Bird1, Bird2,
shooter horizontally using
spacebar fires a bullet
e player gets a point. Bird2
ed to shoot that one; if the
has one minute to shoot

g between these two cos-
s.
and end. The start back-
und is identical, with the
the image. The scripts that

ame

25: The script for the Shooter

This script starts by positioning the shooter in the middle of the bottom edge of the Stage. The script then enters an infinite loop that detects whether the left or right arrow keys have been pressed and moves the shooter in the corresponding direction. Now let's move on to the scripts for Bird1, shown in Figure 7-26.

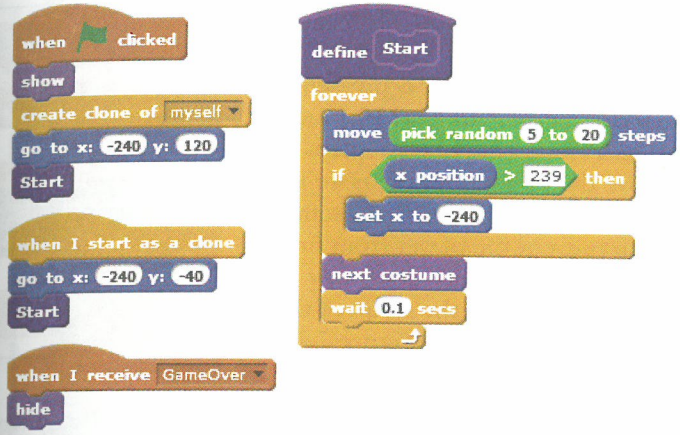


Figure 7-26: The scripts for the Bird1 *sprite*

When the game starts, Bird1 clones itself, moves to left edge of the Stage, and calls the **Start** procedure. The clone also starts at the left edge of the Stage (but at a different height) and calls **Start**. This procedure uses a **forever** loop to move the bird and its clone horizontally across the Stage, from left to right with random steps. When the bird approaches the right edge of the stage, it is moved back to the left edge, as if it wraps around and reappears. The last script hides both birds when the GameOver message is broadcast.

The scripts for Bird2 are very similar to those of Bird1, so we won't show them here. When the green flag is clicked, Bird2 moves to the right edge of the Stage at a height of 40 and then executes a loop similar to that of the **Start** procedure of Figure 7-26. The bird simply moves from left to right, wrapping around when it reaches the right edge of the Stage. Bird2 also responds to the GameOver broadcast by hiding itself.

Of course, the player can't hit any birds just by moving the shooter around, and that's where the Bullet sprite comes in. The main script for this sprite is shown in Figure 7-27.

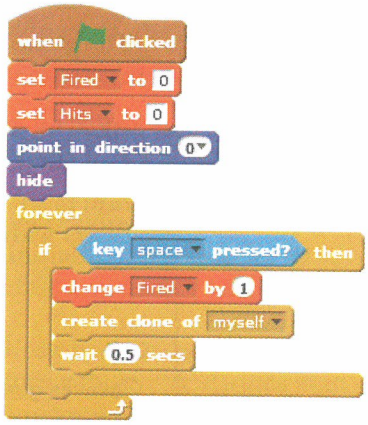


Figure 7-27: The main script of the Bullet *sprite*

When the green flag is clicked, this script initializes the variables Fired (the number of bullets fired) and Hits (how many birds have been hit) to 0. It then points the Bullet sprite up and hides it. After that, it enters an infinite loop to repeatedly check the status of the spacebar key. When spacebar is pressed, the script increments Fired by 1 and creates a clone of the Bullet sprite to move the bullet upward, as we'll see next. The script then waits some time to prevent the player from firing another bullet too soon. Now we're ready to study the script of the cloned bullet, shown in Figure 7-28.

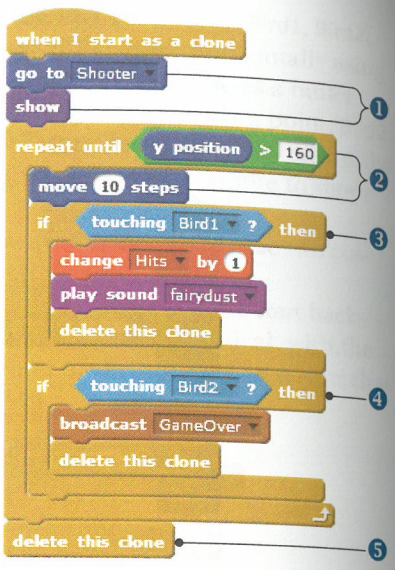


Figure 7-28: The startup handler of a cloned Bullet

First, the Bullet is moved to the center of the Shooter and is made visible ❶. The Bullet is then moved upward in increments of 10 steps using a **repeat until** block ❷. If the bullet's *y*-coordinate exceeds 160, then the Bullet has reached the upper edge of the Stage without touching any birds. In this case, the **repeat until** block exits ❺, and the clone is deleted. A hit check, however, is performed each time the bullet moves. If the bullet touches Bird1 (or its clone) ❸, the script increases the Hits variable and plays a sound to make the game more exciting. On the other hand, if the bullet touches Bird2 ❹, the script broadcasts GameOver to signal the end of the game. In both cases, the clone is deleted since it has finished its job.

The game is now fully functional, but you could add many features to it. Here are two suggestions:

- Give the player a limited number of bullets and keep score based on the number of missed shots.
- Add more birds and have them move at different speeds. Reward the player with more points for hitting faster birds.

TRY IT OUT 7-8

Open the game and play it to see how it works. Modify the game with some of the enhancements suggested above—or come up with a few of your own and implement those!