3

FRUIT LOOT: CREATING A SIMPLE ANIMATED GAME

In this chapter, you'll create a simple game called "Fruit Loot" that uses components from the Drawing and Animation, Sensors, and Media drawers to let players catch falling fruit.

You'll program these components with App Inventor's built-in Math and Variables blocks and component-specific blocks so that the game will use *animation*, or movement, with corresponding sound effects; unpredictability to make the game challenging; and the ability to keep score so players can see how well they're doing.

First, let's explore the key components and underlying programming concepts that you'll use to create the game.

ANIMATING AND MOVING RANDOMLY

To play the game, a player moves a fruit picker character back and forth across the screen trying to catch pieces of falling fruit. The pieces of fruit fall continuously at random speeds from random points at the top of the screen. Because of this random animation, players won't know exactly where to move the picker to catch the fruit. This unpredictability should challenge players and keep them engaged.

PROGRAMMING MOVING IMAGES

We'll use the `Canvas` and `ImageSprite` components from the Drawing and Animation drawer to create the moveable game character and constantly dropping fruit. The `Canvas` component is a layer or sheet that we place on the app's screen so users can draw. The `Canvas` is also where *sprites*, which are flat images, can move around. The game character and different pieces of fruit are all `ImageSprites`, which we'll place on a `Canvas` to make them move, collide with other sprites, and bounce off the edge of the screen.

The height and width of the `Canvas` are measured in *pixels*, a unit of measurement used in computer graphics, and App Inventor uses a common computer screen coordinate system to determine the exact location of an `ImageSprite` on the `Canvas`. In that coordinate system, the top-left point of the `ImageSprite` is located at the point represented by its x- and y-coordinates or properties (`X`,`Y`). The `X` property is the image's distance in pixels from the `Canvas`'s left edge, and the `Y` property is the picture's distance from the `Canvas`'s top edge.

NOTE *In math class, you may have plotted points on a coordinate plane that contains four quadrants. App Inventor's coordinate system is like the lower-right quadrant in that coordinate plane, where the point of origin (0, 0) is at the top left, and the size of the x-coordinate increases from left to right, while the size of the y-coordinate increases from top to bottom. The difference is that, in the math plane, the increasing y-coordinate numbers are negative, while they're positive in App Inventor.*

As shown in Figure 3-1, in App Inventor's coordinate system, an `ImageSprite`'s `X` property value increases as the graphic moves to the right, and its `Y` property value increases as it moves down.

When adding an `ImageSprite` to the `Canvas`, we set its initial `X` and `Y` property values to the point where we place it or to other values we enter into the Designer window's Property pane. To move the `ImageSprite`, we use program blocks to change either property value.

For this game, you'll program button click event handlers to let players move the fruit picker. Also, to constantly animate the fruit, you'll program the `Clock` so the fruit moves automatically at a time interval you'll set. JavaScript and other programming languages handle animation the same way, by having images change location in response to user or automated actions.

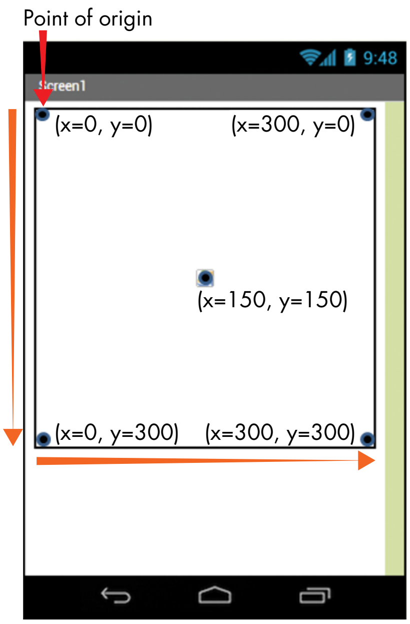


Figure 3-1: The Viewer screen with a 300×300 pixel Canvas *showing the point of origin and* X *and* Y *property values of different points*

SETTING UP RANDOM APPEARANCE, SPEED, AND LOCATION

Like other games that deal with chance, this game relies on randomness to keep players from plotting exactly how, when, or where to make their next moves. Because of the need for randomness in games and other applications, such as simulations, most traditional programming languages include *pseudorandom number generators*, which are functions based on mathematical algorithms.

In its built-in Math blocks, App Inventor provides two pseudorandom number generators. The blocks also include *arithmetic operators* that perform multiplication, division, addition, and subtraction functions on *operands* or values, just like similar operators in other programming languages. In your game, you'll combine one of App Inventor's pseudorandom number generators, called the *random integer block*, with arithmetic operator blocks to develop simple formulas to ensure that the appearance, speed, and location of each piece of falling fruit will be determined randomly.

Thanks to these formulas, although our players will quickly realize that fruit drops constantly, they won't know where and at what speed it will fall, keeping them from scoring points too easily.

DECLARING AND INITIALIZING VARIABLES

As players earn points, we'll need a way to let them know their score. We'll do this by declaring and initializing a couple of *variables*—uniquely named containers of memory that programmers create to hold values that can change, or vary. Variables allow us to store necessary information that we can update from time to time as conditions in the app change. We can use the unique variable name to refer to that changing information throughout our code and perform operations on the information as the app runs, no matter what value the variable holds at any given time.

For instance, in your game, you'll store the score in a variable and compute and display the changing score during the game. To compute the score, you'll use a Math addition operator block to *increment* the score, or increase it by a fixed number—in this case, 1—whenever a player earns a point.

In traditional programming languages, you must follow specific syntactical rules to *declare*, or create, a variable and to *initialize* it, or assign its first value. In some languages, you also have to identify the type of data the variable will hold when you create it. In App Inventor, you must declare and initialize variables using the built-in Variables blocks, and you can store *strings* (sets of characters that can include letters, numbers, and other characters), individual numbers, Boolean values, and lists by snapping in blocks from the Text, Math, Logic, and Lists drawers.

As you work with variables, you'll notice that they're a lot like component properties in that both variables and properties hold data that can be set, reset, and accessed by the blocks used in an app. In fact, as soon as you create a variable, App Inventor creates getter and setter blocks for it, similar to those available for properties, and adds them to the Variables blocks drawer.

In your game, you'll create variables that have a *global scope*, which means you can use them in all of your event handlers. In later chapters, you'll experiment with *local* variables, which you'll create within an event handler or procedure for use only within that handler or procedure. All programming languages use global and local variables.

BUILDING THE "FRUIT LOOT" APP

Now that you understand how to create variables and program animation and randomness in App Inventor, you're ready to create "Fruit Loot."

To get started, log into App Inventor following the instructions outlined in "Getting Started with App Inventor" on page xviii. In the dialog for the project name, enter `FruitLoot` without any spaces, and then click **OK**.

DECOMPOSING "FRUIT LOOT"

In "Fruit Loot," the player moves a fruit picker across the screen to catch rapidly and randomly dropping fruit. The player earns a point for each fruit caught and sees the score on the screen. We can decompose the game activity into five steps:

1. When a player presses the start button, start the game.
2. When the `Clock` timer fires, drop fruit from the top of the Canvas at different speeds.
3. When a fruit hits the bottom of the Canvas, return it to a random point at the top of the Canvas and display another fruit at random. Increase the total fruits dropped by one.
4. When a player clicks the left and right buttons, move the picker left and right to catch the falling fruit.
5. When the picker catches a fruit, play a sound, increase the player's score by one, display the score, and hide the fruit.

Here are the components you'll need:

- **`Button`** (3) for the player to click to manually start the action and play the game
- **`Canvas`** to enable use of `ImageSprites` and game animation
- **`Clock`** to fire after the player clicks the start button and move `ImageSprites` at a set interval
- **`HorizontalArrangement`** (2) to hold start button, score label, and play buttons
- **`ImageSprite`** (4) to display moving images
- **`Label`** to display `Variable` values
- **`Sound`** to play the game sound effect
- Variable (2) to store game data

LAYING OUT "FRUIT LOOT" IN THE DESIGNER

Now let's lay out the app in the Designer. First, change the Screen's horizontal alignment so that everything we place on it will be centered. Click **`Screen1`** in the Components pane, click the drop-down arrow under **`AlignHorizontal`** in the Properties pane, and select **Center: 3**.

Next, let's add a background image to the Screen by clicking the text box under **`BackgroundImage`** in the Properties pane. Follow the image upload instructions outlined in "Uploading a Picture" on page 27 to upload *fence-tree.png*, which comes with the resources for this book. You can download the resources from *https://nostarch.com/programwithappinventor/*.

Now let's change the screen *orientation*, which generally means whether the screen displays vertically (in *portrait* mode) or horizontally (in *landscape* mode). By default, `ScreenOrientation` is set to `Unspecified`, which means that the orientation changes depending on how a user rotates the device.

To give our picker `ImageSprite` a wider screen area to move across to catch fruit, let's change the orientation to landscape mode to make sure the screen displays horizontally regardless of how the device is held. Click the drop-down arrow under **`ScreenOrientation`** and select **Landscape**. Also, unclick the checkbox under both **`ShowStatusBar`** and **`TitleVisible`** to keep the device status bar and Screen title from showing and taking up space when the game displays on a device.

ADDING AND ARRANGING USER INTERFACE COMPONENTS

Since we have limited vertical screen space in landscape orientation mode, we need to take up as little of that space as possible with our user interface components. But we still need to make sure those components are easy for players to see and use. To accomplish this, let's place our `Buttons` and `Label` in two `HorizontalArrangements`, one across the top of the screen and one across the bottom.

Drag two **`HorizontalArrangement`**s from the Layout drawer onto the Viewer. Then, click each in the Components pane, and rename the first one **`TopArrangement`** and the second **`BottomArrangement`**. Then, in the Properties pane, center both of their horizontal alignments by clicking the drop-down arrow under **`AlignHorizontal`** and selecting **Center: 3**, which should center all the components we place inside. Next, make **`BottomArrangement`**'s width **Fill parent**, the same way you did with components in Chapters 1 and 2, so that it stretches all the way across the screen.

Now drag a **`Button`** and a **`Label`** from the User Interface drawer into **`TopArrangement`**. Then, in the Components pane, click **`Button1`** and rename it **`StartBtn`**, and in the Properties pane, change its text size to 18 point by clicking the text box under **`FontSize`**, deleting the current number, and entering **`18`**. Also change the default text showing on `StartBtn` by clicking the text box under **Text**, deleting the current text, and entering **`Start the Fruit Loot Game`**. Then, in the Components pane, click **`Label1`**, and in the Properties pane, remove the `Label`'s default text by clicking the text box under **Text** and deleting the current text so no text will show until the game starts. Then, center the text by clicking the drop-down arrow under **`TextAlignment`** and selecting **Center: 1**.

Next, drag the remaining two **`Button`**s from the User Interface drawer into **`BottomArrangement`**, click each in the Components pane, and rename the one on the left **`LeftBtn`** and the other **`RightBtn`**. Also make the width for each `Button` **Fill parent**, which makes each take up half the width of `BottomArrangement`. Now change the text showing on `LeftBtn` to **`<<<< Left`** and `RightBtn` to **`Right >>>>`**.

Finally, make the background color orange for all three `Buttons` and the `Label` by clicking the box under **`BackgroundColor`** and selecting **`Orange`** from the color list dialog. Also make the text displaying on the Buttons and Label bold by clicking the checkbox under **FontBold**. Next, update the font size on all but StartBtn by clicking the text box under **FontSize** and entering **10** to replace the existing number. Finally, for **BottomArrangement**, unclick the checkbox under **Visible** so LeftBtn and RightBtn won't show when the app opens.

SETTING UP THE CANVAS AND IMAGESPRITES

Now, click the **Drawing and Animation** drawer and drag a **Canvas** onto the Viewer between TopArrangement and BottomArrangement. Remember that you must place a Canvas on the Screen before you can add any other Drawing and Animation component. In the Properties pane, make the Canvas transparent so it doesn't hide the background image by clicking the box under **BackgroundColor** and then clicking **None** when the color list dialog opens. Then make its height and width **Fill parent**.

Now drag four **ImageSprite**s from the Drawing and Animation drawer onto the Canvas, click the **ImageSprite**s in the Components pane, and rename the first three **FruitSprite1**, **FruitSprite2**, and **FruitSprite3** and the last **PickerSprite**. Next, under **Picture** in the Properties pane, for the fruit ImageSprites, upload *1.png*, *2.png*, and *3.png*, and for the picker ImageSprite, upload *picker.png*. (All of these images come with the resources for this book.)

Finally, you can either drag the ImageSprites around the Canvas or enter numbers in the text boxes under their X and Y properties to position them on the Canvas the way they should appear when the game starts. We want the fruit ImageSprites spread out evenly across the top of the Canvas and PickerSprite in the center at the bottom. To place the components this way on a screen that's approximately 450 pixels wide, enter the numbers in Table 3-1 into the Property pane text boxes under **X** and **Y** for each ImageSprite.

Table 3-1: Initial X and Y Property Values for "Fruit Loot" ImageSprites on a 450-pixel-wide screen

ImageSprite	X property	Y property
FruitSprite1	10	0
FruitSprite2	230	0
FruitSprite3	440	0
PickerSprite	180	150

Now you're ready to add and adjust the non-visible components.

ADDING AND PREPARING NON-VISIBLE COMPONENTS

From the Media drawer, drag in a **Sound** component, and from the Sensors drawer, drag in a **Clock** component. In the Components pane, click the **Sound** component, and in the Properties pane, set the media clip that it will play by clicking the text box under **Source** and uploading the *clunk.mp3* file that comes with the book resources.

Then click the **Clock**, replace its default TimerInterval property by entering **150**, and unclick the checkbox under **TimerEnabled** so the timer won't start when the app opens. Shortly, we'll program the blocks to enable the timer once the player clicks StartBtn.

At this point, your screen should look like Figure 3-2.

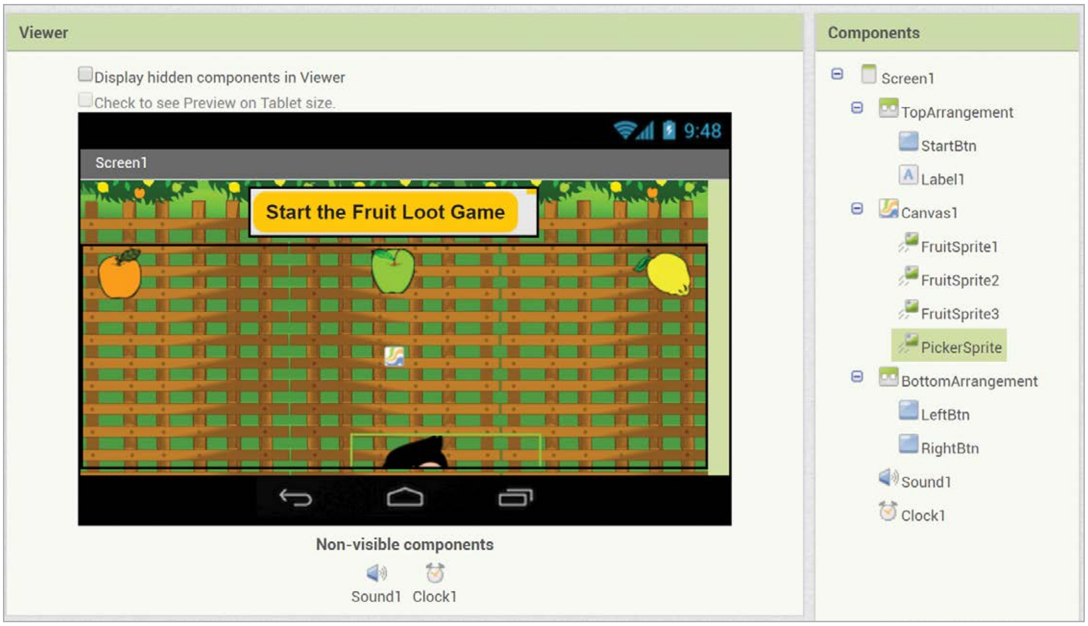


Figure 3-2: The Viewer and Component panes after we lay out "Fruit Loot"

Once your screen looks like it should, click the **Canvas** in the Components pane, and unclick the checkbox under **Visible**. This way, none of the ImageSprites should show when the app opens. Next, we'll program the blocks to make these components, LeftBtn and RightBtn, visible once the player clicks StartBtn.

PROGRAMMING "FRUIT LOOT" IN THE BLOCKS EDITOR

Now that you've laid out all the components, you can move to the Blocks Editor to program the app. For "Fruit Loot" we'll program 10 event handlers. Three respond to events generated by the user's button clicks. One directs the app's action after a timer goes off at the time interval we've set. The rest respond to ImageSprites reaching the edge of the Canvas or colliding with one another.

You'll notice that most of the event handlers contain duplicate code. We're programming them this way because you haven't yet learned the advanced programming structures that would eliminate the repetition.

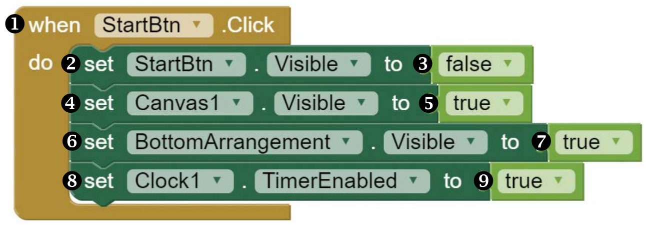
As you learn about those structures in later chapters, we'll be able to revisit the "Fruit Loot" code and *refactor* it, which means to restructure and improve it.

Click the **Blocks** button to switch to the Blocks Editor, and let's begin programming the five steps of "Fruit Loot" in order.

STEP 1: STARTING THE GAME

We start by telling the app what to do when the player clicks `StartBtn`. That's when we want the `StartBtn` to disappear, the `Canvas` with its `ImageSprites` and `BottomArrangement` with its `Buttons` to appear, and the `Clock`'s timer to begin to fire.

Here is the button click event handler with its four setter blocks that do what we want in step 1.



```
❶ when StartBtn .Click
do ❷ set StartBtn . Visible to ❸ false
   ❹ set Canvas1 . Visible to ❺ true
   ❻ set BottomArrangement . Visible to ❼ true
   ❽ set Clock1 . TimerEnabled to ❾ true
```

In the Blocks pane, click **`StartBtn`** and, when the blocks for the component appear, drag the **`whenStartBtn.Click`** event handler block ❶ to the Viewer. Then, in the Blocks pane, click **`StartBtn`** again, and drag its **`set StartBtn.Visibleto`** block ❷ into the **`whenStartBtn.Click`** block next to the word `do`. Next, in the Blocks pane, click the **Logic** blocks drawer, drag the **`false`** block ❸ to the Viewer, and snap it to the right side of the **`setStartBtn .Visibleto`** block. These blocks set `StartBtn`'s `Visible` property to `false` so that it disappears after the player clicks the start button.

Next, click **`Canvas1`**, drag the **`setCanvas1.Visibleto`** block ❹ to the Viewer, and snap it inside the **`whenStartBtn.Click`** block under the `setStartBtn.Visibleto` block. Then, in the Blocks pane, click the **Logic** blocks drawer again, drag the **`true`** block ❺ to the Viewer, and snap it to the right side of the **`setCanvas1 .Visibleto`** block. These blocks set the `Visible` property for `Canvas1` and its contents to `true` so the `ImageSprites` will appear after the player clicks the start button.

Then, click **`BottomArrangement`** in the Blocks pane, drag the **`setBottom Arrangement.Visibleto`** block ❻ to the Viewer, and snap it inside the **`whenStart Btn.Click`** block under the `setCanvas1.Visibleto` block. Then click the **Logic** blocks drawer again, drag another **`true`** block ❼ to the Viewer, and snap it to the right side of the **`setBottomArrangement.Visibleto`** block. These blocks set the `Visible` property for `BottomArrangement` to `true`, making the buttons inside of it appear after the player clicks the start button.

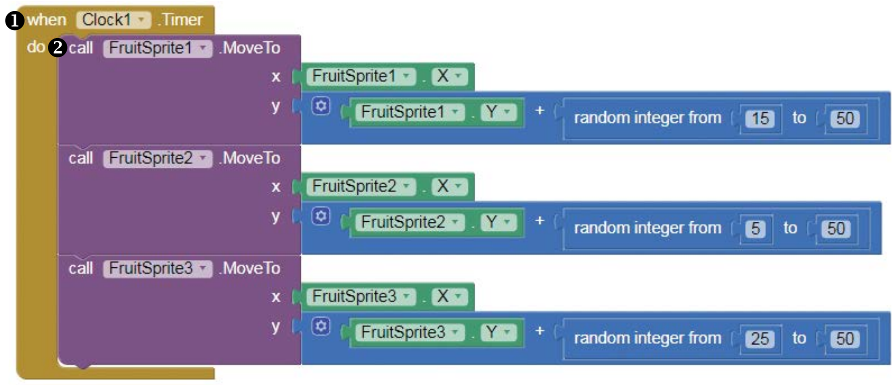
Finally, click **`Clock1`**, drag the **`setClock1.TimerEnabledto`** block ❽ to the Viewer, and snap it inside the **`whenStartBtn.Click`** block under the `setBottom Arrangement.Visibleto` block. Then, in the Blocks pane, drag another **`true`** block ❾ from the **Logic** blocks drawer, and snap it to the right side of the `setClock1.TimerEnabledto` block. These blocks set the value of `Clock1`'s `Timer Enabled` property to `true`. This starts `Clock1`'s timer, which will move the fruit `ImageSprites` down the `Canvas` the entire time the game is in play.

Together, the blocks for step 1 start the game. In sum, when the player clicks the start button, the blocks set `StartBtn`'s `Visible` property to `false`, set the `Visible` properties of the `Canvas` with `ImageSprites` and the `Horizontal Arrangement` with play buttons to `true`, and set the `Clock`'s `TimerEnabled` property to `true`.

To see how these blocks work, live-test with a device, as outlined in "Live-Testing Your Apps" on page xxii. Once you click **Connect ▸ AI Companion** in the top menu bar and scan the QR code with your device's Companion app, your "Fruit Loot" game should open on your device. As long as your blocks are placed as shown in the code examples, you should see the start button until you click it, when it disappears as the other game components appear. For now, nothing else should happen. Leave the game open on your device to keep live-testing.

STEP 2: MAKING FRUIT DROP AT RANDOM

Now let's program step 2 of the app and tell it what to do each time the `Clock`'s timer fires. This is when we want the fruit to drop at varying speeds every 150 milliseconds—the `TimerInterval` we set in the Property pane in the Designer.



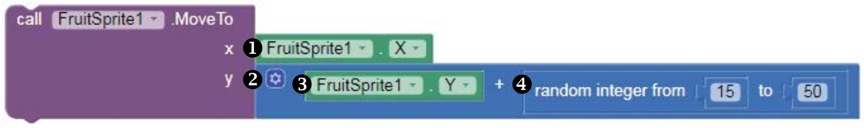
```
❶ when Clock1 .Timer
do ❷ call FruitSprite1 .MoveTo
        x  FruitSprite1 . X
        y  FruitSprite1 . Y  +  random integer from 15 to 50
     call FruitSprite2 .MoveTo
        x  FruitSprite2 . X
        y  FruitSprite2 . Y  +  random integer from 5 to 50
     call FruitSprite3 .MoveTo
        x  FruitSprite3 . X
        y  FruitSprite3 . Y  +  random integer from 25 to 50
```

In the Blocks pane, click **`Clock1`** and, when the blocks for the component appear, drag the **`whenClock1.Timer`** block ❶ to the Viewer. Then, in the Blocks pane, click **`FruitSprite1`**, drag the **`callFruitSprite1.MoveTo`** method block ❷ to the Viewer, and snap it inside the **`whenClock1.Timer`** block next to the word `do`.

NOTE *In both the Components pane in the Designer and the Blocks pane in the Blocks Editor, if you don't see a component that you've nested within a parent component—for example, an `ImageSprite` placed on a `Canvas` or a `Button` dragged within a `HorizontalArrangement`—you'll find it by clicking the plus sign to the left of the parent.*

Setting X and Y Values for FruitSprite1

Let's look closer at the `callFruitSprite1.MoveTo` method block we've placed within the `Clock1 Timer` event handler.


call FruitSprite1 .MoveTo
x ❶ FruitSprite1 . X
y ❷ ❸ FruitSprite1 . Y + ❹ random integer from 15 to 50

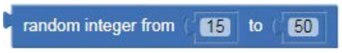
You'll notice that the block requires us to insert values for its `x` and `y` *method parameters*, which are pieces of information the method must have to operate. This means the `ImageSprite`'s `MoveTo` method cannot move `Fruit Sprite1` until we supply *arguments*, or values, for the x- and y-coordinates of the point where we want the `ImageSprite` to move.

For our "Fruit Loot" game, we want the fruit `ImageSprites` to move down only, meaning we'll change their y-coordinates but not their x-coordinates. To keep the same `X` value, click **`FruitSprite1`**, drag its **`FruitSprite1.X`** getter block ❶ to the Viewer, and snap it into the method block's **`x`** socket. This tells the app that, when it moves `FruitSprite1`, it should get the current `X` value for `FruitSprite1` and keep that `X` value the same.

To provide the argument for the y-coordinate of the point where we want `FruitSprite1` to move, click the **Math** blocks drawer, drag out an addition operator block ❷, and snap it into the method block's **`y`** socket. Then, click **`Fruit Sprite1`** and drag its **`FruitSprite1.Y`** getter block ❸ into the addition block's left socket, and click the **Math** blocks drawer and drag a `random integer` block ❹ into the addition block's right socket. This tells the app that, when it moves `FruitSprite1`, it should increase the current value of its y-coordinate by a random number of pixels to move the `ImageSprite` down the `Canvas`.

Dropping FruitSprite1 at Random Speeds

The `random integer` block generates the random number of pixels—from between the specified range of 15 to 50—that we want `FruitSprite1` to fall.

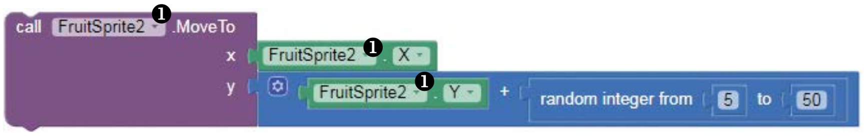


To set that range of numbers in the `random integer` block, click the default **`1`** in its left socket and replace it by entering **`15`**, and click the default **`100`** in its right socket and replace it by entering **`50`**.

Now, altogether, our `callFruitSprite1.MoveTo` method block with the `x` and `y` parameters we've set tells the app that, when it moves `FruitSprite1`, we want the `ImageSprite`'s `X` value to stay the same and its `Y` value to move from its current y-coordinate down a random number of pixels between 15 and 50. This randomness ensures that the `ImageSprite`'s speed will be unpredictable, because, when the `Clock`'s timer fires every 150 milliseconds, `FruitSprite1` will travel at a speed anywhere from a slower 15 pixels per 150 milliseconds (100 pixels per second) to a faster 50 pixels per 150 milliseconds (333 pixels per second).

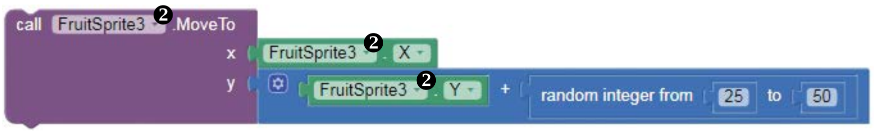
Copying Blocks for FruitSprite2 and FruitSprite3

To complete the blocks for step 2, we now can copy the callFruitSprite1 .MoveTo block and adjust it for FruitSprite2 and FruitSprite3. Right-click the **callFruitSprite1.MoveTo** method block to duplicate it for **FruitSprite2**, and snap the duplicate in under the original.


call FruitSprite2 ❶ .MoveTo
x FruitSprite2 ❶ . X
y FruitSprite2 ❶ . Y + random integer from 5 to 50

In the duplicate blocks, use the drop-down arrows ❶ in the **callFruitSprite1.MoveTo**, **FruitSprite1.X**, and **FruitSprite1.Y** blocks to change to **FruitSprite2**. Also change the number in the left random integer block socket to **5**. These blocks now program FruitSprite2 to move down some unknown number of pixels between 5 and 50 when the Clock's timer fires every 150 milliseconds.

Next, right-click the **callFruitSprite1.MoveTo** block, make another copy to use for FruitSprite3, and snap the duplicate in under the **callFruitSprite2 .MoveTo** block.


call FruitSprite3 ❷ .MoveTo
x FruitSprite3 ❷ . X
y FruitSprite3 ❷ . Y + random integer from 25 to 50

In the duplicate, use the drop-down arrows ❷ in the **callFruitSprite1 .MoveTo**, **FruitSprite1.X**, and **FruitSprite1.Y** blocks to change to **FruitSprite3**, and change the number in the random integer block's left socket to **25**. These blocks program FruitSprite3 to move down a random number of pixels between 25 and 50 every 150 milliseconds.

Now the blocks for step 2 should move the three fruit ImageSprites down the Canvas every 150 milliseconds at random speeds.

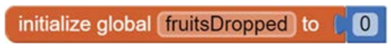
Live-test to see how these blocks work. When you click StartBtn, you should see the three fruit ImageSprites drop to the bottom of the screen, where they stay and all movement stops. If any ImageSprite fails to move, you need to debug. In this instance, you may not have changed your references to the correct ImageSprite when you duplicated the MoveTo blocks. Make any necessary corrections, and test again. Once step 2 is working, move to the next step, where we'll tell the game what to do when the fruit ImageSprites reach the bottom of the Canvas.

STEP 3: CREATING MORE FALLING FRUIT AND COUNTING DROPPED FRUIT

Let's now program step 3 of the app. In this part, when a fruit ImageSprite reaches the bottom edge of the Canvas, we want the app to move the

ImageSprite back up to a random point along the very top of the Canvas, have the ImageSprite display a random picture of fruit, and add 1 to the total number of times an ImageSprite hits the edge.

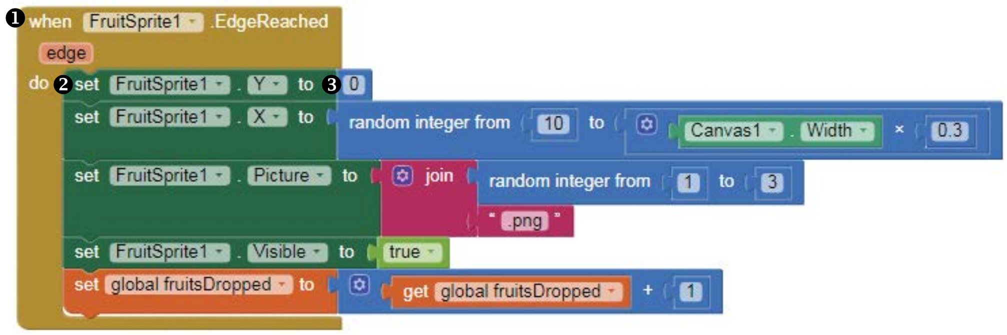
We'll use a global variable to store and update that total number, and we'll start our code for this step by creating and initializing that variable.



```
initialize global fruitsDropped to 0
```

Click the **Variables** blocks drawer and drag an **`initialize global name`** block to the Viewer. Click **`name`** and replace it with the name of our variable, **`fruitsDropped`**. Then drag a **`0`** number block from the **Math** drawer and snap it to the right side of the **`initialize global fruitsDropped`** block. This declares and initializes the variable you'll use to store and update the total number of pieces of fruit dropped in your game. Because the variable is global and can be used by all your event handlers, it stands alone in the code, outside of all your event handler blocks.

Now let's program the event handler for this step. Here are the blocks that handle this `EdgeReached` event for `FruitSprite1`.



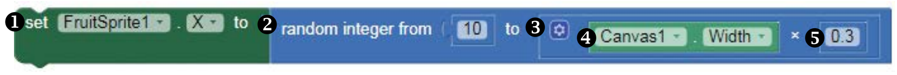
```
❶ when FruitSprite1 .EdgeReached
   edge
do ❷ set FruitSprite1 . Y to ❸ 0
   set FruitSprite1 . X to random integer from 10 to Canvas1 . Width × 0.3
   set FruitSprite1 . Picture to join random integer from 1 to 3
                                      " .png "
   set FruitSprite1 . Visible to true
   set global fruitsDropped to get global fruitsDropped + 1
```

In the Blocks pane, click **`FruitSprite1`** and, when the blocks for the component appear, drag the **`whenFruitSprite1.EdgeReached`** block ❶ to the Viewer. Then, in the Blocks pane, click **`FruitSprite1`** again, drag the **`setFruitSprite1.Y`** block ❷ to the Viewer, and snap it inside the **`whenFruitSprite1.EdgeReached`** block next to the word `do`.

Then, click the **Math** blocks drawer, drag a **`0`** number block ❸ to the Viewer, and snap it to the right of the **`setFruitSprite1.Y`** block. So far, once `FruitSprite1` reaches the edge of the `Canvas`, these blocks move `FruitSprite1` right back up to y-coordinate 0, which is the very top of the `Canvas`.

Moving Fallen Fruit Back Up to a Random Place

We now need to make sure the code also moves `FruitSprite1` to an unpredictable x-coordinate using setter blocks, which keeps your game interesting.



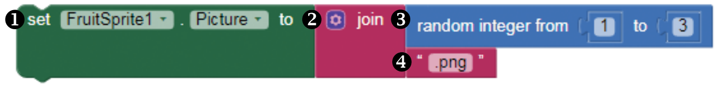
```
❶ set FruitSprite1 . X to ❷ random integer from 10 to ❸ ❹ Canvas1 . Width × ❺ 0.3
```

In the Blocks pane, click the **FruitSprite1** component, drag the **setFruitSprite1.X** block ❶ to the Viewer, and snap it inside the **whenFruitSprite1.EdgeReached** block under the setFruitSprite1.Y block. Then, click the **Math** blocks drawer, snap a random integer block ❷ to the right of the **setFruitSprite1.X** block, and click the default **1** in the random integer block's left socket and replace it by entering **10**. Then, delete the default **100** in the random integer block's right socket, and replace it with a multiplication operator block ❸, also from the Math drawer. Next, in the Blocks pane, click the **Canvas1** component, drag the **Canvas1.Width** getter block ❹ to the Viewer, and snap it into the multiplication block's left socket; then, drag a **0** number block ❺ from the **Math** drawer to the Viewer, click its default **0**, replace it by entering **0.3**, and then snap the **0.3** number block into the multiplication block's right socket.

These blocks set the X value for FruitSprite1 once it reaches the edge of the Canvas. To avoid collisions with other fruit ImageSprites, we want this first fruit ImageSprite to drop somewhere in the left third of the Canvas only. These blocks ensure that by setting the new X position to a random number of pixels between 10 and the width of the Canvas multiplied by 0.3, which is a little less than one-third of the Canvas width. For instance, if the width of the Canvas is 450 pixels, the new X position will be anywhere between 10 and (450 × 0.3) pixels, which equals 135 pixels.

Dropping Random Fruit Images

Next, to keep your game unpredictable, you need to make sure the code randomly changes the type of fruit dropped after FruitSprite1 moves back up to the top of the Canvas. To do this, you'll use setter blocks that set the Picture property for FruitSprite1 to a random image.



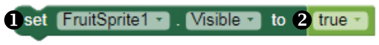
Click the **FruitSprite1** component in the Blocks pane, drag the **setFruitSprite1.Picture** block ❶ to the Viewer, and snap it inside the **whenFruitSprite1.EdgeReached** block under the setFruitSprite1.X block. Then, click the **Text** blocks drawer, drag a join block ❷ to the Viewer, and snap it to the right of the **setFruitSprite1.Picture** block, which will allow you to set the name for the picture by joining two strings.

For the join block's top input, drag in another random integer block ❸ from the Math blocks drawer, click the default **100** in its right socket, and replace it by entering **3**. For the join block's second input, drag in an empty string block ❹, the first block in the Text blocks drawer. Then click the string block's text area and enter **.png**.

These blocks set the name of the image to use as the picture source for FruitSprite1 after it reaches the Canvas edge. Since we've named the three uploaded fruit images *1.png*, *2.png*, and *3.png*, we can use the random integer block to generate the number 1, 2, or 3 that is part of the image name. This should make the app continually display a randomly selected image on FruitSprite1 each time it drops from the top of the Canvas.

Making Sure Fruit is Visible

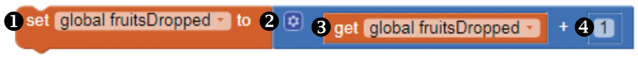
We also need to make sure `FruitSprite1` and the other fruit `ImageSprites` are visible once they move back up to the top of the `Canvas`, because later we'll make them invisible if they hit the picker `ImageSprite`. Here is the setter block that turns the `Visible` property on.



Click **`FruitSprite1`**, drag the **`setFruitSprite1.Visibleto`** block ❶ to the Viewer, and snap it inside the **`whenFruitSprite1.EdgeReached`** block under the `setFruitSprite1.Pictureto` block. Then, in the Blocks pane, click the **Logic** blocks drawer, drag the **`true`** block ❷ to the Viewer, and snap it to the right side of the **`setFruitSprite1.Visibleto`** block. These blocks reset the `ImageSprite`'s `Visible` property to `true` in case it collides with the picker `ImageSprite`, after which our blocks in step 5 will set it to `false`.

Counting the Number of Fruits Dropped

Finally, we need to program the app to keep track of how many fruits are dropped. Each time a fruit `ImageSprite` hits the bottom of the `Canvas`, the game should add 1 to the value of `fruitsDropped`, which is the variable that keeps track of the number of times an `ImageSprite` hits the edge. The following blocks increment the value of `fruitsDropped`.


❶ set global fruitsDropped to ❷ ❸ get global fruitsDropped + ❹ 1

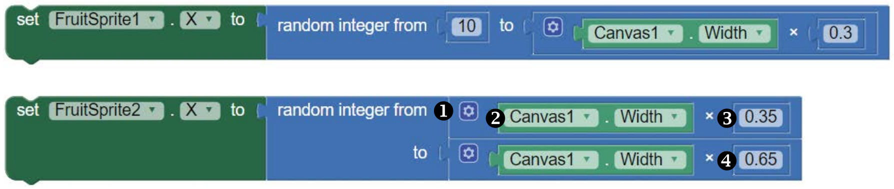
Mouse over the **`initialize global fruitsDropped`** block that we placed at the beginning of this step, drag the **`set global fruitsDropped`** block ❶ to the Viewer, and snap it inside the **`whenFruitSprite1.EdgeReached`** block under the `setFruitSprite1.Visibleto` block. Then drag an addition operator block ❷ from the Math drawer and snap it to the right of the **`set global fruitsDropped`** block. Fill that addition block's sockets by mousing over the **`initialize global fruitsDropped`** block, dragging the **`get global fruitsDropped`** block ❸ into the addition block's left socket, and dragging a **`1`** number block ❹ from the Math drawer into the addition block's right socket. These blocks keep track of the game's total number of fruits dropped by adding 1 to the current value of the `fruitsDropped` variable each time `FruitSprite1` reaches the edge of the `Canvas`.

Copying Blocks for FruitSprite2 and FruitSprite3

Our final task for step 3 is to duplicate our code to program similar `EdgeReached` event handlers for `FruitSprite2` and `FruitSprite3`.

For `FruitSprite2`, right-click the **`whenFruitSprite1.EdgeReached`** block and select **duplicate**. When you make this duplicate, you'll see a red X appear to the left of the word `when` in both the original and duplicate event handlers. This red X warns you that you have two event handlers for the same event, which is not allowed. Once you change the duplicate handler's event, the red X should disappear. To change the event, use the drop-down arrow in every block where you see `FruitSprite1` and change to **`FruitSprite2`**.

The only other adjustment we need to make is to set an `X` property for `FruitSprite2` that avoids collisions with the other fruit `ImageSprites` when it moves down the `Canvas`. To accomplish this, make sure this second fruit `ImageSprite` consistently drops somewhere in the middle third of the `Canvas` by changing the `FruitSprite2 X` value.



```
set FruitSprite1 . X to   random integer from 10 to  Canvas1 . Width × 0.3

set FruitSprite2 . X to   random integer from ❶  ❷ Canvas1 . Width × ❸ 0.35
                                          to      Canvas1 . Width × ❹ 0.65
```

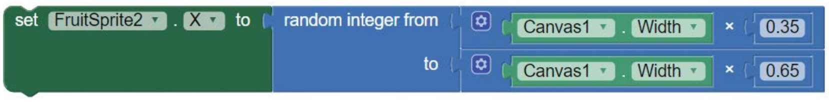
NOTE *App Inventor provides a way for you to display the* `random integer`, `join`, *and other blocks that require multiple inputs with either* inline inputs, *as shown in the* `random integer` *block inside the* `FruitSprite1` *setter, or* external inputs, *as shown in the* `random integer` *block inside the figure's* `FruitSprite2` *setter, which takes up less horizontal space. You can right-click a block to switch between inline and external inputs.*

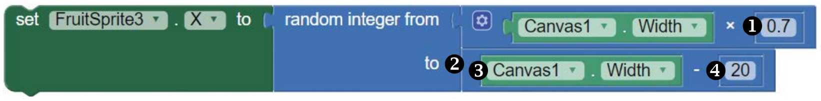
Replace the **`10`** number block in the **`random integer`** block's top socket with a multiplication operator block ❶ from the Math drawer. Then fill the multiplication block's sockets by clicking the **`Canvas1`** component in the Blocks pane, dragging the **`Canvas1.Width`** block ❷ into its left socket, and dragging a **`0.35`** number block ❸ from the Math drawer into its right socket. Then, in the `random integer` block's bottom socket, change the **`0.3`** number block to a **`0.65`** number block ❹.

These blocks set the new `X` position for `FruitSprite2` to a random number of pixels between the width of the `Canvas` multiplied by 0.35 and the width of the `Canvas` multiplied by 0.65, which is some random point in the middle third of the `Canvas`. For instance, if the width of the `Canvas` is 450 pixels, the new `X` position will be anywhere between 450 × 0.35 pixels, which equals 158 pixels, and 450 × 0.65 pixels, which equals 293 pixels.

Now, to create the `EdgeReached` event handler for `FruitSprite3`, right-click the **`whenFruitSprite2.EdgeReached`** block and select **duplicate**. In the duplicate blocks, be sure to use the drop-down arrow in every block where you see `FruitSprite2` and change to **`FruitSprite3`**.

To avoid collisions with the other fruit `ImageSprites`, we'll also need to change the `FruitSprite3 X` value so this third fruit `ImageSprite` consistently drops somewhere in the right third of the `Canvas`.





To do this, replace the **`0.35`** number block in the right socket of the first multiplication block with a **`0.7`** number block ❶. Also, replace the second multiplication block with a subtraction operator block ❷ from the Math blocks drawer, and fill the subtraction block by dragging the **`Canvas1.Width`** block ❸ into its left socket and a **`20`** number block ❹ into its right socket.

These blocks set the new `X` position of `FruitSprite3` to a random number of pixels between the width of the `Canvas` multiplied by 0.7 and the width of the `Canvas` minus 20 pixels, which is some random point in the right third of the `Canvas`. For instance, if the width of the `Canvas` is 450 pixels, the new `X` position will be anywhere between 450 × 0.7 pixels, which equals 315 pixels, and 450 – 20 pixels, which equals 430 pixels.

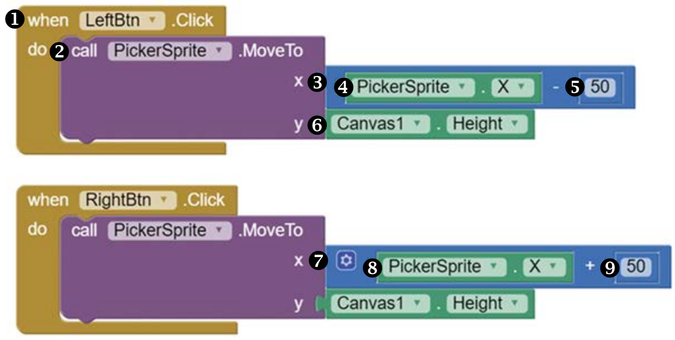
Altogether, the blocks for step 3 move each fruit `ImageSprite` to a random point at the very top of the `Canvas`, have that `ImageSprite` display a random picture of fruit, and increase the count of total fruits dropped by 1 each time an `ImageSprite` hits the bottom of the `Canvas`, just as we planned.

Now live-test the game again. This time, when you click `StartBtn`, you should see the three fruit `ImageSprites` drop to the bottom of the screen continuously, randomly changing the image displayed. You'll also notice that the speed at which each `ImageSprite` drops and its `X` location changes with each drop.

If any of the `ImageSprites` fail to move, or if two or more appear to drop in the same third of the `Canvas`, debug your code. Here again, you may not have made the correct changes to your duplicate blocks. Make any necessary corrections, and test again. Once step 3 is working, let's move to the next part, where we'll program `PickerSprite`'s movement.

STEP 4: LETTING PLAYERS MOVE THE PICKER TO CATCH THE FRUIT

Now let's program step 4 of the app, telling it what to do when the player clicks the left and right play buttons. When the player clicks `LeftBtn`, we want `PickerSprite` to move to the left 50 pixels, and when the player clicks `RightBtn`, we want `PickerSprite` to move 50 pixels to the right.


```
❶when LeftBtn .Click
do ❷call PickerSprite .MoveTo
                       x ❸ ❹PickerSprite . X - ❺50
                       y ❻Canvas1 . Height

when RightBtn .Click
do call PickerSprite .MoveTo
                       x ❼ ❽PickerSprite . X + ❾50
                       y Canvas1 . Height
```

In the Blocks pane, click **`LeftBtn`** and, when the blocks for the component appear, drag the **`whenLeftBtn.Click`** block ❶ to the Viewer. Then, in the Blocks pane, click **`PickerSprite`**, drag the **`callPickerSprite.MoveTo`** block ❷ to the Viewer, and snap it inside the **`whenLeftBtn.Click`** block next to the word `do`.

Now we need to provide the `MoveTo` block's method parameters to tell the app where we want to move `PickerSprite`, keeping in mind that, for this game, we want `PickerSprite` to move from side to side only, along the very bottom of the `Canvas`. That means we want to change its x-coordinate but leave its y-coordinate at the bottom. To do this, click the **Math** blocks drawer, drag a subtraction operator block ❸ to the Viewer, and snap it to the right of **`x`**. Then, click **`PickerSprite`** and drag its **`PickerSprite.X`** block ❹ into the subtraction block's left socket, and drag a **`50`** number block ❺ from the Math drawer into the subtraction block's right socket.

These blocks tell our app to move `PickerSprite`'s x-coordinate left to its current location minus 50 pixels whenever the `Button` is clicked. For instance, if `PickerSprite`'s x-coordinate is at 240 pixels, when the player clicks `LeftBtn`, the x-coordinate should move 50 pixels to the left to 190 pixels, since 240 pixels – 50 pixels = 190 pixels.

Next, click **`Canvas1`**, drag its **`Canvas.Height`** block ❻ to the viewer, and snap it into the **`callPickerSprite.MoveTo`** block's **`y`** socket. This tells the app that, when it moves `PickerSprite`, we want the `ImageSprite`'s `Y` value to stay the value that equals the height of the `Canvas`, positioned at the bottom. For instance, if the `Canvas` is 300 pixels in height, these blocks will keep `PickerSprite`'s y-coordinate at the `Canvas`'s bottommost point, 300 pixels, when `LeftBtn` is clicked.

Now copy the **`LeftBtn`** event handler and modify the duplicate blocks to program **`RightBtn`**. First, in the duplicate event handler, be sure to use the drop-down arrow to change **`LeftBtn`** to **`RightBtn`**. Then replace the subtraction block after the letter `x` with an addition block ❼ from the Math drawer, click **`PickerSprite`** and drag its **`PickerSprite.X`** block ❽ into the addition block's left socket, and drag a **`50`** number block ❾ from the Math drawer into the addition block's right socket.

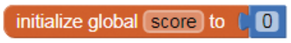
These blocks say move `PickerSprite`'s x-coordinate to its current location plus 50 pixels when the button is clicked. So, if `PickerSprite`'s x-coordinate is at 240 pixels when the player clicks `RightBtn`, the x-coordinate should move 50 pixels to the right to 290 pixels, since 240 pixels + 50 pixels = 290 pixels.

Now live-test the game again, and if LeftBtn and RightBtn don't work correctly after you click StartBtn, try debugging. LeftBtn and RightBtn should move PickerSprite back and forth across the screen, while fruit ImageSprites occasionally collide with PickerSprite. Since the player's goal in the game is to collide with, or "catch," the fruit, we need to program quite a bit of activity to take place when those fruit ImageSprites hit PickerSprite. We'll program that action in the next, and final, step.

STEP 5: HIDING CAUGHT FRUIT AND KEEPING SCORE

Now we'll program the last part of the game so that each time a fruit ImageSprite hits PickerSprite, the player "catches" the piece of fruit, hears a noise that sounds like the fruit hitting the picker's bucket, earns a point, and sees the total score displayed on the screen. We'll also hide the ImageSprite that hit PickerSprite so that, instead of continuing to fall to the bottom of the Canvas, the fruit looks like it landed in the picker's bucket.

To keep the player's score, we'll use a variable to store and update that information. Let's start our code for this step by creating and initializing the score variable.

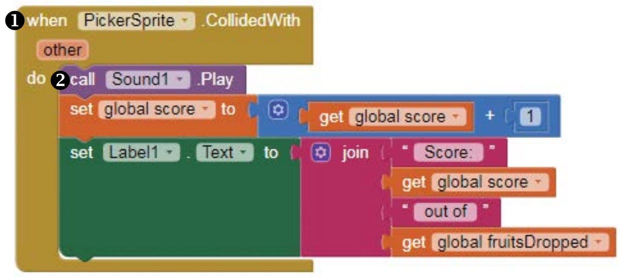


```
initialize global score to 0
```

Click the **Variables** block drawer and drag an **initialize global name** block to the Viewer. Click **name**, and replace it with the name of our variable, **score**. Then drag a **0** number block from the Math drawer and snap it to the right side of the **initialize global score** block. This declares and initializes the global variable we'll use to store and update the player's game score.

Playing a Sound When Fruit Hits the Picker

Let's now program the event handler for when a fruit ImageSprite hits PickerSprite.

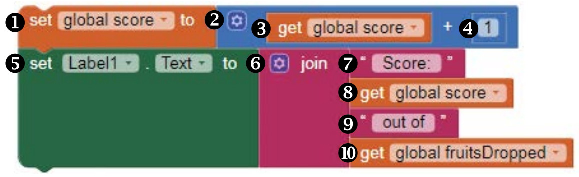


```
❶when PickerSprite .CollidedWith
  other
do ❷call Sound1 .Play
   set global score to  get global score + 1
   set Label1 . Text to  join " Score: "
                              get global score
                              " out of "
                              get global fruitsDropped
```

In the Blocks pane, click **PickerSprite** and, when the blocks for the component appear, drag the **whenPickerSprite.CollidedWith** block ❶ to the Viewer. Then, in the Blocks pane, click **Sound1**, drag the **callSound1.Play** block ❷ to the Viewer, and snap it inside the **whenPickerSprite.CollidedWith** block next to the word do. This should play our clunking sound each time an Image Sprite hits PickerSprite.

Increasing and Displaying the Score

Next, let's place the blocks that increment and display the game score each time a fruit ImageSprite hits the picker.



❶ set global score to ❷ ❸ get global score + ❹ 1
❺ set Label1 . Text to ❻ join ❼ " Score: "
❽ get global score
❾ " out of "
❿ get global fruitsDropped

Mouse over the **initialize global score** block, drag the **set global score to** block ❶ to the Viewer, and snap it inside the **whenPickerSprite.Collided With** block under the callSound1.Play block. Then drag an addition operator block ❷ from the Math drawer and snap it to the right of the **set global score to** block. Next, mouse over the **initialize global score** block again, and drag the **get global score** block ❸ into the addition block's left socket and a **1** number block ❹ from the Math drawer into its right socket. These blocks add 1 to the current value of the score variable each time a fruit ImageSprite collides with PickerSprite.

To display the score and also let the player know how many of the total number of dropped fruits PickerSprite has caught, click **Label1**, drag the **set Label1.Textto** block ❺ to the Viewer, and snap it inside the **whenPickerSprite .CollidedWith** block under the set global score to block. Then, click the **Text** blocks drawer, drag a **join** block ❻ to the Viewer, and snap it to the right of the **setLabel1.Textto** block.

Here, we'll join four strings to set the text and numbers we want Label1 to display, although by default the join block allows us to combine only two strings. Figure 3-3 shows how to change the block to create space for the additional inputs we'll need.

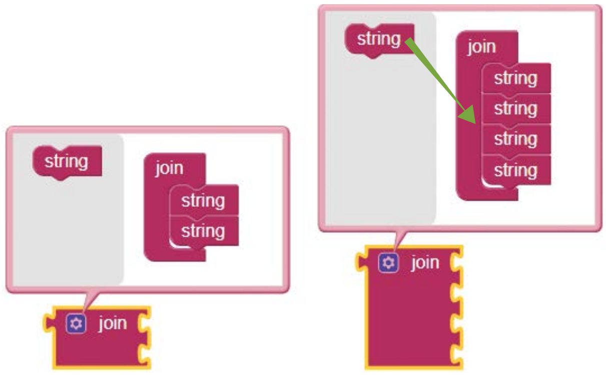


Figure 3-3: Adding inputs to the join block

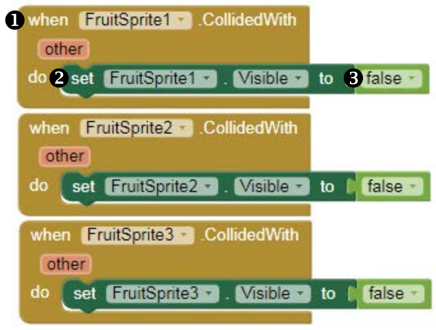
Click the blue *mutator* icon to the left of the word join, and drag additional string blocks to the join block in the dialog that opens.

Now we can snap our four strings into the join block inputs. In the top input, drag in an empty string block ❼ from the Text blocks drawer and enter

Score: , making sure to include the space after the colon so that, when the strings combine, the characters won't run together without proper spacing. Then mouse over the **initialize global score** block and drag the **get global score** block ❽ into the join block's second input. In the join block's third input, drag in another empty string block ❾ and enter **out of** , leaving a space before out and after of. Then mouse over the **initialize global fruitsDropped** block and drag the **get global fruitsDropped** block ❿ into the join block's last input. These blocks display the number of points and total number of fruits dropped on Label1 for the player to see at the top of the screen. For instance, if the player's score is 6 points and a total of 20 fruits have dropped, the label should display "Score: 6 out of 20."

Hiding Caught Fruit

Finally, let's add the blocks that will make the fruit ImageSprites disappear after they collide with PickerSprite.



❶ when FruitSprite1 .CollidedWith
other
do ❷ set FruitSprite1 . Visible to ❸ false
when FruitSprite2 .CollidedWith
other
do set FruitSprite2 . Visible to false
when FruitSprite3 .CollidedWith
other
do set FruitSprite3 . Visible to false

Click **FruitSprite1** and, when the blocks for the component appear, drag the **whenFruitSprite1.CollidedWith** block ❶ to the Viewer. Then, click **Fruit Sprite1**, drag the **setFruitSprite1.Visibleto** block ❷ to the Viewer, and snap it inside the **whenFruitSprite1.CollidedWith** block next to the word do. Next, in the Blocks pane, click the **Logic** blocks drawer, drag the **false** block ❸ to the Viewer, and snap it to the right side of the **setFruitSprite1.Visibleto** block.

Now duplicate these blocks for FruitSprite2, taking care to use the drop-down arrows both places you see FruitSprite1 to change to **FruitSprite2**. Then duplicate the blocks again, and be sure to change to **FruitSprite3**. These three event handlers hide the fruit ImageSprites when they hit PickerSprite so it looks like the picker successfully caught the fruit in the bucket. In the next chapter, you'll learn how to eliminate these duplicate blocks and accomplish the same task using a more sophisticated programming structure.

Now, following the plan for step 5, each time a fruit ImageSprite touches PickerSprite, the app should play a sound, increase the player's score by 1, display the score and the total pieces of fruit, and hide the fruit ImageSprite.

It's time to test the completed game! Open the app on your device, and you should see StartBtn at the top of the screen. Click it, and when it disappears, you should see the other game components appear. Now the fruit starts to randomly drop, and you can click LeftBtn and RightBtn to move PickerSprite back and forth across the screen to try to catch it.

Whenever `PickerSprite` catches a piece of fruit, you should hear a sound and see your score increase. If you placed your blocks correctly, the game should work as described, and you'll have successfully created the "Fruit Loot" game!

SUMMARY

In this chapter, you built the animated "Fruit Loot" app, a game where a player moves a fruit picker back and forth across the screen and earns points when the picker catches rapidly and randomly dropping fruit.

You learned how programmers animate an object by moving its x- and y-coordinates; use pseudorandom number generators to add randomness in games, simulators, and other applications; and work with arithmetic operators to manipulate data. You also practiced declaring and initializing variables to store and change information, and you learned how to provide required arguments for built-in methods with parameters.

In the next chapter, you'll do more with math operators and random number blocks and begin to make selections in your code using Control blocks. You'll use those tools to create part 1 of the "Multiplication Station" quiz app, which generates random, timed multiplication problems, evaluates solutions the user inputs, and then speaks to declare those answers right or wrong.

ON YOUR OWN

Save new versions of "Fruit Loot" as you modify and extend it working on these exercises. You can find solutions online at *https://nostarch.com/programwithappinventor/*.

1. Change the app so that it calculates and keeps track of how many pieces of fruit the picker fails to catch during a game. How can you calculate, store, and display this information using the existing event handlers and adding the smallest number of additional blocks?
2. Extend the game so that the frustrated owner of the fruit trees, who can't keep the fruit from falling over the fence, drops rocks down the fence to try to keep the picker from attempting to catch the falling fruit. Reduce the player's score each time the rock hits another sprite. What components and blocks will you add?
3. Extend the game even further so that the score label displays the number of times the rock hits another sprite.