3
Detecting Collisions

In this chapter, we will continue to develop the game that we started to create in Chapter 2, Let's make a game. Although we have many of the basic game mechanics in place, we do not yet have a very interesting game that will engage our players, so we're going to introduce an additional element of interactivity to make the game more challenging.

In its current form, the game allows our monkey to navigate the jungle scene without any difficulty, and that isn't going to be much fun for players of our game — we need a more interesting environment!

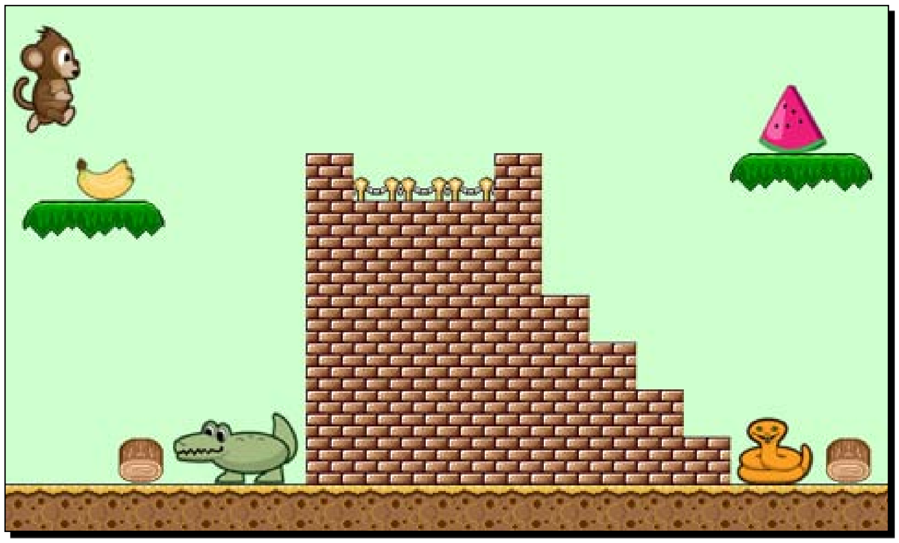
To make the game more challenging, we're going to add some dangerous enemies for the monkey to avoid, and we're also going to create a challenge for the player to complete — the monkey will have to collect some items as it makes its journey through the jungle.

In order to be able to collect these items, we need our monkey to detect collisions with other objects, so our game can respond accordingly. In this chapter, we're going to learn the important aspects of implementing collision detection within Stencyl.

By the end of this chapter, we will have a good understanding of the following topics:

- Working with collision detection in Stencyl
- Modifying an actor's collision shapes
- Configuring collision shapes for tiles
- Adding enemies and collectibles
- Working with collision groups
- Using collision sensors
- Implementing terrain collision shapes

The target for this chapter is to progress with our development of the game so that it looks something like the following screenshot:



As we can see, our monkey now has fruits to collect and dangerous animals to avoid!

Working with collision detection in Stencyl

If our monkey has to collect items and avoid enemies, we will need to detect collisions between the monkey and the various actors that will represent the enemies and collectible items. In other words, our game needs to know when these objects have bumped into each other!

Stencyl, as we may now expect, makes the job of working with collisions much easier than it would be if we had to hand code our game using a traditional programming language. We can easily specify which parts of our tiles and actors will cause a collision event, and we can also easily configure behaviors to carry out certain actions for us when collision events occur.

In fact Stencyl is already automatically managing collisions within our game! Let's have a look behind the scenes, using a very useful feature that has been built into Stencyl to help us manage the collision detection:

Time for action – enabling the Debug Drawing feature

The game file to import and load for this session is `5961_03_01.stencyl`.

Stencyl makes it incredibly easy to see how collision detection is configured within our game; all we have to do is use the `Enable Debug Drawing` feature.

1. On the main menu in Stencyl, select **Run** | **Enable Debug Drawing**.
2. Test the game.
3. Navigate the monkey around the scene and take note of where the colored lines appear around the monkey and the tiles.
4. Close the **Adobe Flash Player** window.

What just happened?

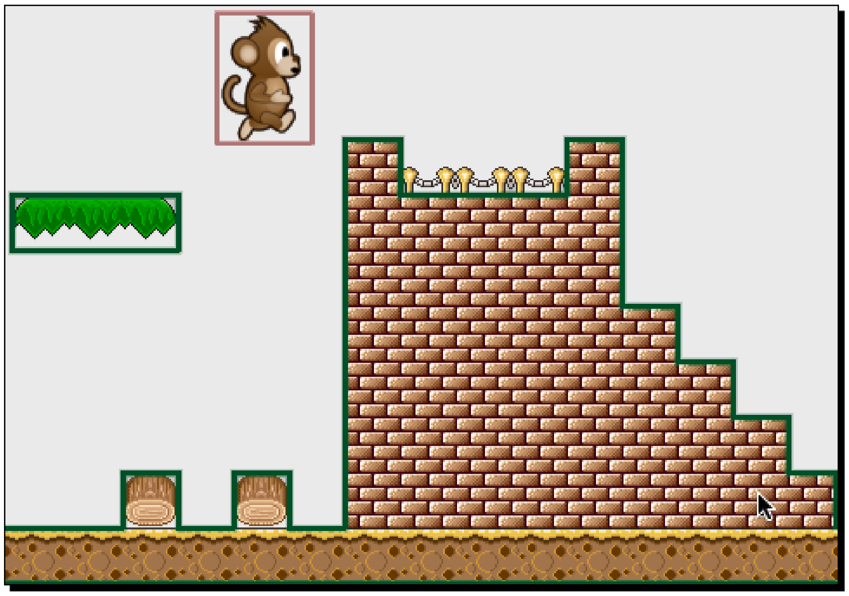
We've enabled the option that displays the debug drawing feature in Stencyl. When we tested the game with debug drawing enabled, we could see that colored lines are drawn around the monkey actor and the tiles.

These lines, which are normally invisible to players of our game, are used by Stencyl to determine when collisions have occurred, that is, when the lines of any two objects intersect (or touch) each other. If we examine some of the colored lines more closely, we can see that they completely enclose each object; for example, the monkey and the floating leafy platforms are each enclosed by their own rectangles.

When adjoining tiles form a larger, more complex shape, Stencyl will automatically create a single, large collision object. As we can see in the following screenshot, the ground, the walls of the temple ruins, and the logs are all considered by Stencyl to be a single solid object, even though we know that we created the jungle environment using small rectangular tiles.

The combining of tiles in this manner is an optimization process automatically carried out by Stencyl; we don't need to do any extra work in order for this to happen!

Optimization is the process of making computer code do its job more efficiently, which, in practical terms, means that our games will run more smoothly, even when there is a lot of onscreen action and background calculation.



Debug drawing is extremely useful when designing a game, because it enables us to visualize the way in which Stencyl processes collisions, and this can help us see problems that we may otherwise have overlooked; we can debug collision detection within our game!

The colored lines around the objects represent the collision shapes for those objects, but we didn't need to define the collision shapes, because they had already been configured in the actor and the tiles that we downloaded from StencylForge.

We can also see, in the previous screenshot, that most of the tiles are enclosed within collision shapes, but some are not. For example, the rope barrier on top of the temple ruins does not have a collision shape defined, which is why the monkey doesn't bump into the rope during gameplay. This feature allows us to use tiles that are purely for decorative purposes. There is no technical reason for them to be in our game; they just look nice.

Modifying an actor's collision shapes

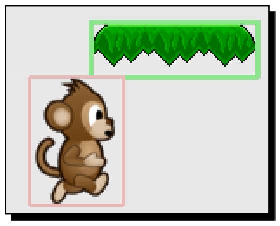
Now that we know what collision shapes are, we can learn about modifying these shapes to meet our exact requirements.

Although the initial testing of our game demonstrated that the monkey can successfully traverse the jungle environment by running on the ground and jumping on the various obstacles that we created with tiles, there are some improvements to be made!

Note how the collision box around the leafy platform is larger than it needs to be; it projects far lower from the bottom of the platform than is necessary. This is likely to be annoying for players of our game, if the monkey jumps up when it is beneath the platform, it will bump the top of its head before it actually hits the platform. Even worse, the collision box around the monkey is also too large, which may lead to even greater frustration for our players.

To examine this problem more closely, test the game again, and make the monkey jump up when it is standing immediately beneath a platform. Although it's not a major issue while we're designing the game, players of our game will quickly give up if the same problem occurs when the monkey is standing close to an enemy actor – a collision will occur before a player can see the actors touching, and they will consider it to be unfair. This is a guaranteed way to annoy players, and they'll quickly start looking for a more fun game elsewhere!

We can see how this problem occurs in the following screenshot; a collision is detected whenever the collision shapes of the monkey and the platform intersect, but there is still quite a big gap between the image of the monkey and the platform.



The good news is that Stencyl equips us with the tools to make the required adjustments. We can use the **Animation Editor** to alter an actor's existing collision shapes and to add new ones. We can even add multiple collision shapes to an actor, so that we can form more complex shapes that closely represent the shape of our actor.

[When discussing collision detection, we can also refer to an actor's **collision bounds**; this simply denotes a single collision shape or multiple collision shapes that are used to specify the collision-detection areas for an actor.]

Let's have a look at how we can improve the collision detection for the monkey.

Time for Action – modifying the monkey's collision shapes

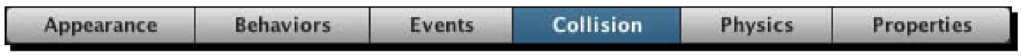
The game file to import and load for this session is `5961_03_01.stencyl`.

We're going to fine-tune the collision shapes for the monkey, so ensure that the monkey actor is visible in the Animation Editor:

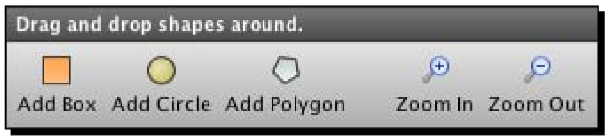
1. Go to the **Dashboard** tab, click on **Actor Types**, then double-click on the monkey actor's thumbnail.

   Now that we can see the monkey animations, we can go ahead and modify the collision bounds.

2. Click on the **Collision** button in the row of buttons at the upper-center of the screen.

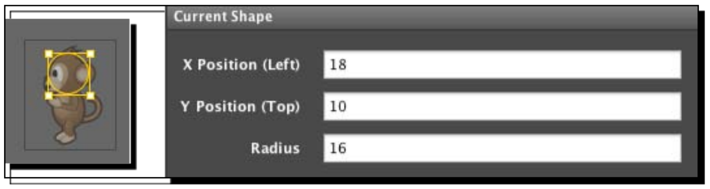


3. Look at the center of the screen, where the collision shape for one of the monkey animations is displayed.
4. Click on the **Idle Left** thumbnail in the **Animations** (left) panel to display the collision bounds for the **Idle Left** animation.
5. Click somewhere just inside the border of the orange rectangular collision shape, so that the box is selected and press the *Delete* key on the keyboard to remove the collision shape.
6. Locate the collision shapes tools at the top of the screen.



7. Click on **Add Circle** to display the **Add Circle** dialog box.
8. Click on **OK**.
9. Click inside the orange border of the new collision circle to ensure that it is selected.
10. Use the arrow keys on the keyboard to position the circle over the monkey's head. It should have an **X position (Left)** of `18` units and a **Y position (Top)** of `10` units as displayed in the **Current Shape** panel at the upper-right of the screen.

The position of the collision circle and the configuration of the **Current Shape** panel are shown in the following screenshot.



X Position (Left) 18
Y Position (Top) 10
Radius 16

What just happened?

Firstly, we displayed the collision configuration screen for the monkey's **Idle Left** animation, and then we deleted the existing rectangular collision shape.

The next step was to add a new collision shape. In this case, we chose a circle because it more accurately represents the shape of the collision that we want to detect. We then moved the collision circle so that it was positioned over the monkey's head.

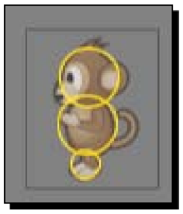
It's not desperately important for the collision circle to exactly cover the monkey's head. In fact, it's often better if the collision shapes for actors are very slightly smaller than the image of the actor, because it will appear to be fairer to the player, than if the collision shape is too large.

It's the game designer's decision to make sure that the collision bounds for each actor are appropriate for the game, and often an appropriate configuration can only be determined by playtesting the game. This is why it's important to find impartial testers to play our games and provide feedback, as discussed in the *Chapter 2* section entitled, *Finding game testers*.

At this point, we can clearly see that we have a problem; the collision circle that we have added will not detect when the monkey's body and feet collide with other objects! We can resolve this problem by adding more collision shapes; we aren't restricted to only one shape for each animation.

Adding multiple collision shapes

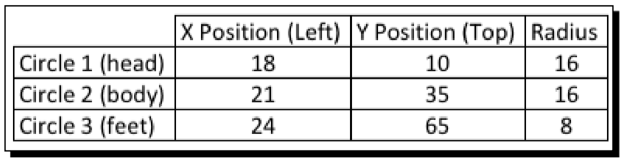
We're going to add two more collision shapes to ensure that the collision detection for the **Idle Left** animation is accurate, and our goal is for the collision shapes, for the **Idle Left** animation, to look like the following screenshot:



Time for action – adding more collision shapes to the monkey

The game file to import and load for this session is `5961_03_02.stencyl`.

We will use the following information to position the additional collision shapes correctly in step 3:



	X Position (Left)	Y Position (Top)	Radius
Circle 1 (head)	18	10	16
Circle 2 (body)	21	35	16
Circle 3 (feet)	24	65	8

1. Ensure that the monkey's **Idle Left** animation is visible in the center panel of Collision Editor.
2. Click on the **Add Circle** tool at the top of the center panel.
3. Referring to the previous diagram, enter the **X Position (Left)**, **Y Position (Top),** and **Radius** information for the body's collision circle (Circle 2) into the **Add Circle** dialog box, and click on **OK** to confirm.
4. Repeat steps 2 and 3 to create a collision circle for the monkey's feet.
5. Save the game.

What just happened?

We've added two more collision shapes to the monkey actor to ensure that it can more appropriately interact with tiles and other actors in our game.

We can see that the collision bounds for the **Idle Left** animation now consists of three circles. Circular collision shapes were chosen for the monkey actor because they enable it to more easily slide off the edge of platforms and other obstacles. However, it's a good idea, when developing a game, to experiment with different collision shapes in order to achieve the best results for collision detection.

We can test the game now, if desired, but be aware that we have only made changes to the **Idle Left** animation. We still have to modify the collisions for the remaining animations for the monkey actor.

Planning the collision shapes

Before spending too much time refining the collision shapes for each of an actor's animations, it's a good idea to plan in advance and carefully consider the requirements for each animation. Also, consider the following tips, which could save some time!

- Don't spend too long agonizing over each collision shape; to start with, only reasonable accuracy is required. We can always revisit the collision bounds at a later stage in development if the existing configuration doesn't work as required.
- Consider the requirements for each animation. For example, in our game, the monkey's *dead* animation does not require any collision detection. So, no collision shapes are required!
- Once a collision shape has been created, the arrow keys on the keyboard can be used to adjust the position of the shape; this can be much faster than retyping the coordinates in the Animation Editor's **Current Shape** panel.
- When several animations are similar, configure one of them, and then jot down the collision shape settings, so they can be copied for the other similar animations. It will then just take a few moments to enter these settings and adjust them slightly, if required.
- We don't have to use circles for the collision shapes; rectangular collision shapes, and even polygonal shapes should be used if appropriate for the shape of the actor!
- Take into account what a player might consider to be fair or unfair when playing the game. Don't make collision shapes too large or too small.
- Remember to save the game regularly!

Have a go hero

The game file to import and load for this session is `5961_03_03.stencyl`.

We've configured the collision bounds for the monkey's **Idle Left** animation, but all the other monkey animations (**Idle Right**, **Jump Left**, and so on) need their collision bounds updated.

Update the collision bounds for each of the monkey's animations, taking note of the planning tips provided previously.

Testing the updated collision bounds

When all the monkey animations' collision bounds have been created and adjusted, we can save and test the game again. Be sure to thoroughly test to ensure that the collision shapes for the monkey are suitable for the environment; for example, if the monkey jumps into a small gap, can it jump out again, or will it become trapped?

We may need to move some objects, such as the log tiles, within the scene to ensure that the monkey can move around the scene freely. These are the types of problems that need to be resolved before a game can be released to the public, so don't be afraid to make changes to the collision bounds or to adjust the layout of the scene to account for any gameplay difficulties that are discovered during the testing process.

There should now be a great improvement in the collision-detection accuracy when the monkey collides with other objects and with most of the tiles, but we still need to do some work on the collision shapes of the leafy platforms.

Configuring collision shapes for tiles

When we need to fine-tune the collision bounds for tiles; the process is slightly different than it is for actors, so let's learn how we can improve the collision detection for the leafy platform.

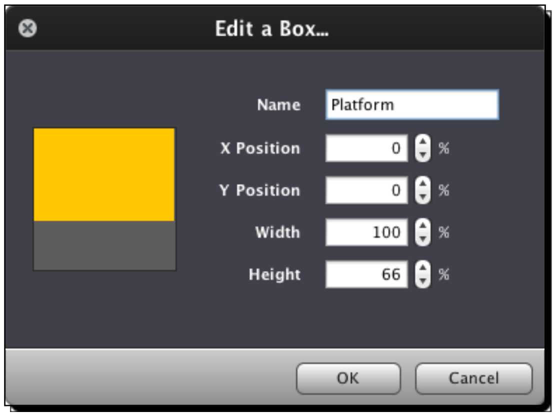
Time for action – modifying the collision bounds of a tile

The game file to import and load for this session is `5961_03_04.stencyl`.

As we've seen, the collision box is too large for the leafy platforms; it only needs to surround the top two-thirds of the tile, so let's create a new tile collision shape that meets that requirement.

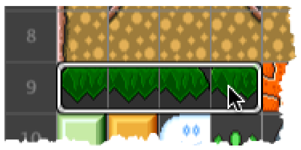
We're going to be modifying tile collisions, so we need to display the Tileset Editor:

1. Go to the **Dashboard** tab, click on **Tilesets**, then double-click on the thumbnail for our tileset in the main panel.
2. Click on any tile in the tileset in the left-hand panel; the collision-shapes panel will be displayed at the top of the right-hand panel.
3. In the upper-right panel, click on the gray **+** button and then select **Create Box…** in the pop-up menu.
4. In the **Create a Box…** dialog box, enter the **Name**, **X Position**, **Y Position**, and the **Width** and **Height** information as shown in the following screenshot:

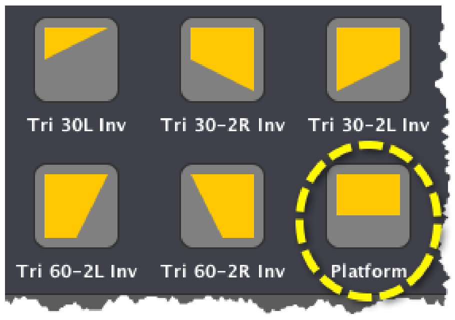


Name: Platform
X Position: 0 %
Y Position: 0 %
Width: 100 %
Height: 66 %

5. Click on **OK**.
6. In the left panel, click-and-drag the mouse on the first leafy platform tile (position A9 in the grid) across to the fourth leafy platform tile (position D9), and then release the mouse. The four tiles are selected as shown in the following screenshot:



7. In the right-hand panel, click on the newly created **Platform** collision shape. It may be necessary to scroll down to the bottom of the right-hand panel in order to see the new collision shape.



8. Select the **Jungle scene** tab (or open it, if it is not already open) and save the game.

What just happened?

Using the Tileset Editor, we created a new tile collision shape called `Platform`. We then selected the four tiles that we wanted to modify, and clicked on the icon for the new collision shape, so that it was applied to all the platform tiles that we selected.

Because we changed the collision shape for existing tiles in a scene, we had to open the scene and save it to ensure that the changes to the collision shapes were applied to the relevant tiles in the scene.

Note that the new collision shape we created in the Tileset Editor is also available to the *whole* tileset, so any other tiles can use it without us having to define it again.

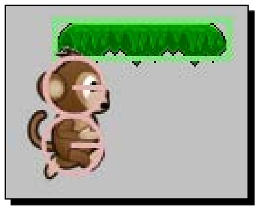
We could have specified the collision shape for each tile one at a time, but selecting the four tiles was much quicker, and if an appropriate collision shape had already existed in the upper-right panel, we could have just clicked on the existing collision shape to apply it to the selected tiles.

Have a go hero

The game file to import and load for this session is `5961_03_05.stencyl`.

Now that we've updated the collision shapes for both the monkey actor and the platform tiles, the collision detection in our game should be much improved.

Test the game, ensuring that the **Enable Debug Drawing** option is selected in the **Run** menu, and have a look at the accuracy of the collision between the monkey and the platforms that we have just modified.



We can see that the collision is now very accurate; there isn't a large gap between the monkey's head and the platform. In fact, there is no gap at all. The monkey's hair slightly overlaps the overhanging leaves on the platform, and this interaction between the actor and the platform will be much more acceptable to players of our game.

Perhaps a little more fine-tuning of the collision bounds might be required before the game is completed, but that is the time to ensure that some serious playtesting is carried out, and feedback from our testers should be reviewed, with adjustments to the game being made as required.

Adding enemies and collectibles

Now that we have refined the collision detection for the monkey and the tiles, it's time to add some enemies for our monkey to avoid, and we also need to find some interesting items for the monkey to collect. Adding these objects into our game will introduce the first level of challenge for our players.

We've already learned how to import actors from StencylForge, and that's where we'll find the new actors we're going to place into the jungle scene.

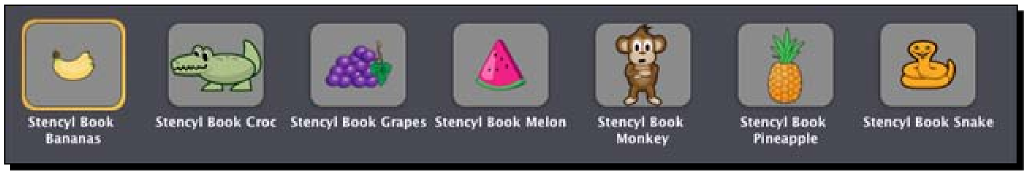
Downloading the enemies and collectibles

The game file to import and load for this session is `5961_03_05.stencyl`.

We already know how to search for and download actors from StencylForge, so let's download the actors with the following names:

- `Stencyl Book Croc`
- `Stencyl Book Snake`
- `Stencyl Book Melon`
- `Stencyl Book Bananas`
- `Stencyl Book Pineapple`
- `Stencyl Book Grapes`

If we now go to the **Dashboard** tab and click on **Actor Types**, we should see the following display in the main panel:



Now that we have added all the actors, we need to save the game!

Placing the new actors into the jungle scene

The game file to import and load for this session is `5961_03_06.stencyl`.

Now that we've downloaded the actors from StencylForge, it's time to place them into the jungle scene. As a designer, it's our job to place the enemies and collectibles into locations that will provide a challenge for players of our game, but as with the design of the collision bounds for the actors, we must not make the game too frustrating nor should we make it too easy.

We have already added actors to the scene by clicking on the **Add to Scene** button when viewing the actor in the Animation Editor. However, there is a more efficient way to add multiple actors into a scene without having to display each actor's animations first.

If we display the scene, we can see two buttons at the top of the right-hand panel: one is for displaying the available tiles and the other is for displaying the available actors. If the available actors are not already on display, click on the **Actors** button. We can now add actors to the scene in exactly the same way that we added tiles: click on an actor, then add it to the scene by clicking on the required location. It's great to know that all the skills we learned when adding tiles work equally as well for actors!

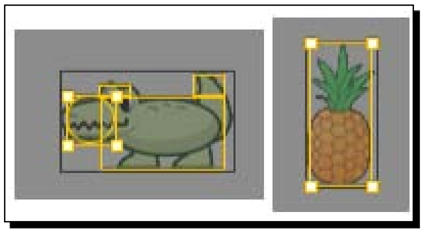
Here's another useful tip before we start adding more actors to the scene; holding the *Shift* key on the keyboard while positioning actors on a scene snaps the actor into place on an invisible grid. This can be very useful for placing actors at regular evenly spaced positions within a scene. Also have a look at the tools available within the Scene Designer, at the upper-right of the main panel. Hovering the mouse pointer over each tool will provide a tip. We're most interested in the three grid tools at this point!

- Add each of the new actors (fruits and enemies) to the jungle scene.
- Save the game.

> If we test the game right now, we'll see that there is a problem with the collision testing, and we may also experience some strange effects, such as some of the fruit or enemies moving unexpectedly.
>
> Don't worry, it's just the Stencyl's physics engine behaving as it should; we're going to fix these problems!

We can now test the game. Although all the enemy actors and the collectible fruits are clearly visible, when the monkey collides with them, nothing happens; he runs straight past them without even the slightest bump!

This might lead us to believe that there is a problem with the collision bounds of the newly added actors. However, if we have a look at the collision bounds for these actors, we can see that they are all in place. The collision shapes of two of the new actors are shown as follows.



Clearly, something is wrong with the collision detection! We have one final task to complete to ensure that Stencyl can detect collisions correctly.

Working with collision groups

Behind the scenes, collision detection is quite a complex process, but Stencyl gives us great control over which collisions will be detected between different actors and tiles. This control is achieved by allowing us to place actors into groups, and then providing us with the ability to specify which groups will collide with each other.

For example, we may want our monkey actor to collide with enemy actors and also with the collectible fruit actors, but we might not want the enemy actors to collide with the fruit actors.

Let's have a look at how the collision groups are currently configured.

Time for action – examining the collision group settings

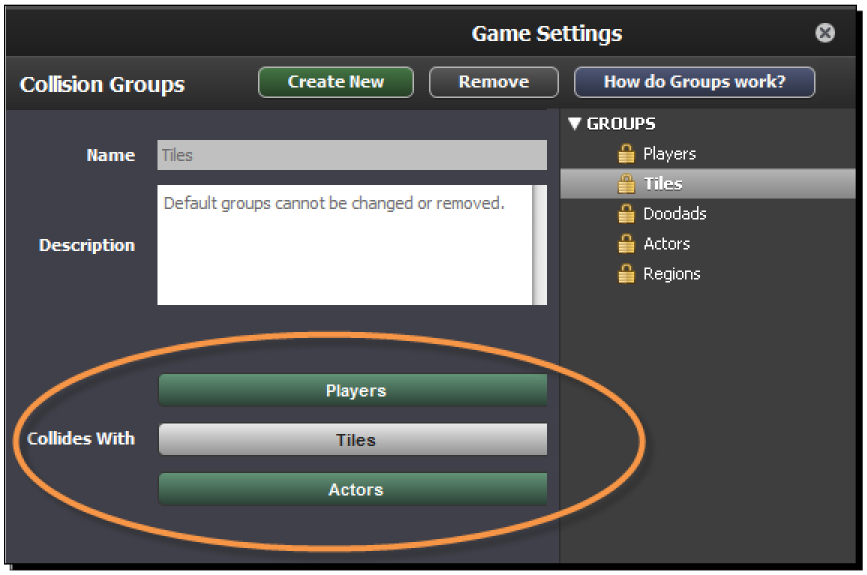
The game file to import and load for this session is `5961_03_07.stencyl`.

We need to open up the **Collision Groups** window in the Stencyl **Settings** dialog box:

1. Click the **Settings** icon on the main Stencyl toolbar.
2. Look down the list of icons that are shown in the left panel of the **Game Settings** dialog box and click on **Groups**.
3. In the **Collision Groups** window that is now being displayed, click on **Tiles** under the **GROUPS** heading at the top of the right-hand panel.

What just happened?

We have displayed the current collision settings for the **Tiles** group as shown in the following screenshot:



Name: Tiles

Description: Default groups cannot be changed or removed.

The **Collides With** section in the previous screenshot, which has been highlighted with an oval shape, shows which other collision groups have been configured to collide with the **Tiles** collision group. Currently, both the **Players** and the **Actors** collision groups are configured to collide with the **Tiles** group (remember, the **Tiles** group is the one we selected in the right-hand panel). Note that the **Players** button and the **Actors** button are shaded darker than the **Tiles** button, which shows us that they are currently selected.

This means that any actor that is placed in the **Players** group or the **Actors** group will react to a collision with a tile. Note that, if we configure a group such as **Tiles** to collide with **Players**, then the **Players** group will automatically be updated to reflect this change!

Viewing the actors' collision groups

We now know that tiles are configured to react to collisions with any actor which is in either the **Players** or **Actors** group, so let's see which groups **Players** and **Actors** are configured to detect collisions with.

Time for action – examining the Players and Actors groups

The game file to import and load for this session is `5961_03_07.stencyl`.

1. Click on **Players** in the right-hand panel (under the **GROUPS** heading).
2. Examine the buttons at the bottom of the main panel in the **Collides With** section; make a note of which group or groups are selected.
3. Click on **Actors** in the right-hand panel (under the **GROUPS** heading).
4. Examine the buttons at the bottom of the main panel in the **Collides With** section; make a note of which groups are selected.
5. Click on the **OK** button to hide the **Game Setting** dialog box.

What just happened?

We examined the collision settings for the **Players** group and the **Actors** group, and we should have noted that any actor in the **Players** group can detect collisions with tiles only. We should also have noted that actors in the **Actors** group can detect collisions with both **Tiles** and other actors in the **Actors** group.

It's useful to know that actors do not have to detect collisions with other actors in the same group.

The ability to configure collisions between actors in different groups gives us an amazing amount of flexibility in our game, when we need to manage collisions between actors and tiles.

Let's put this new found knowledge into practice!

> The collision groups entitled **Doodads** and **Regions** have special features that we will not be using. However, it's useful to note that actors in the **Doodads** group will never cause collision events to occur, and that **Regions** is a special type of group that enables us to specify arbitrary areas of a scene that can cause collision events.

Creating a new collision group

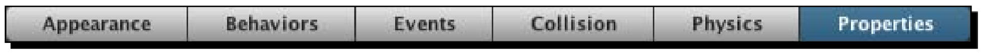
We know from our previous game test, that the monkey completely ignored the enemy actors that it ran past. If we need to remind ourselves what happened, we can test the game again; the monkey is able to run straight through the enemy actors without a collision occurring!

Time for action – creating a collision group for enemy actors

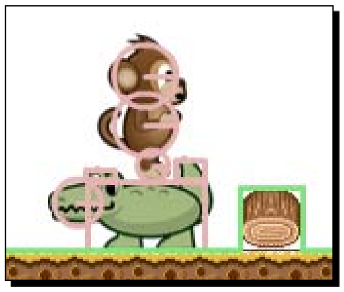
The game file to import and load for this session is `5961_03_07.stencyl`.

We're now going to configure the collision detection, so that the monkey will collide with the enemy Croc.

1. Display the **Stencyl Book Croc** actor in the Animation Editor (click on the **Dashboard** tab, then **Actor Types** and double-click on the **Stencyl Book Croc** thumbnail).
2. Click on the **Properties** button in the row of buttons at the upper-center of the screen.



3. Click the **Edit Groups** button in the **Choose Group** section to display the **Collisions Groups** window in the **Game Settings** dialog box.
4. Click on the **Create New** button at the top of the dialog box.
5. In the **Create a New Group…** dialog box that appears, type `Enemies` into the **Name** textbox and click on the **Create** button.
6. Look at the bottom of the list in the right panel, and we can now see that the new group has been added to the list.
7. In the main panel, click on the **Players** button and the **Tiles** button.
8. Click on **OK** to return to the Stencyl Book Croc **Properties** window.
9. In the **Choose Group** section, click on the **Group** drop-down list and select **Enemies**.
10. Test the game; make the monkey collide with the Croc!



What just happened?

We have created a new collision group called `Enemies`, which will contain actors that collide with the players and tiles, and we have added the Croc actor to this new collision group.

Creating a new collision group requires us to view the **Collision Group** window in the **Game Settings** dialog box, and then either select the group that we want to work with, or create a new group. We can then click on the buttons for the other group names to specify which other collision groups our currently selected group will collide with.

Have a go hero

The game file to import and load for this session is `5961_03_08.stencyl`.

We have already created an **Enemies** collision group and added the Croc actor to this group, so we know how to add the Snake actor to the **Enemies** group by changing its collision group in the actor's **Properties** window.

- Add the Snake actor to the **Enemies** group.
- Don't forget to test the game to ensure that the monkey collides with the snake!

Configuring collisions for the fruit actors

The game file to import and load for this session is `5961_03_09.stencyl`.

The enemy actors are now configured so that they are part of the **Enemy** collision group, but we haven't set up a collision group for the collectible fruit.

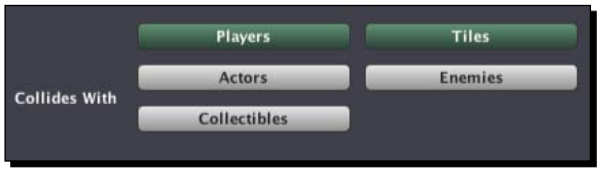
We have already learned the skills required for creating collision groups and for adding actors into a collision group, so our next task is to put those skills into practice and configure the collision group for the fruit.

In order to ensure that our game can detect collisions between the monkey and the fruit, we must complete the following steps:

1. Create a `Collectibles` collision group.
2. Configure the **Collectibles** group so that it detects collisions with **Tiles** and **Players**.
3. Add each fruit actor to the **Collectibles** collision group.

Remember, to create a new group, we can either click on the **Settings** icon on the toolbar, and then click on the **Groups** option in the left panel, or we can display one of the fruit actors in the Animation Editor, and click on the **Properties** button for that actor, which will enable us to see the **Edit Groups…** button.

The buttons determining the collisions for the **Collectibles** group should be configured as shown in the following screenshot:



When the **Collectibles** collision group has been created, remember to view the **Properties** window for each of the pieces of fruit, and change the collision group to **Collectibles**.

It's vital to test the game at this stage to check that all the collisions are working as expected. Make sure that the monkey collides with all four types of fruit to ensure that the collision group is configured correctly.

If, when the game is tested, we find that the collisions with the fruit actors aren't working as expected, review the *Time for action – creating a collision group for enemy actors section*, which contains the steps for creating a new group, specifying which groups collide, and adding an actor to the newly created group.

Using collision sensors

In most platform games, when the player's character passes a collectible item, it doesn't normally bump into it and stop. The player will run straight through the item, collecting points as it passes, and the collectible item will disappear so that it can't be collected more than once.

We know from the previous game test that when our monkey collides with a piece of collectible fruit, it crashes into it and stops, or lands on top of the fruit if the monkey is jumping. We need the collision to be detected so that our game can reward the player with a bonus, but we don't want to stop the flow of the game by having to stop and start the monkey each time a piece of fruit is collected.

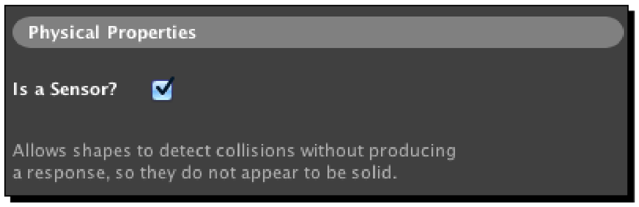
When we want a collision to be detected, but not cause a physical reaction in the game, we need to specify that the colliding actor is a **sensor**.

Time for action – configuring the fruit as a sensor

The game file to import and load for this session is `5961_03_10.stencyl`.

We'll start by setting up the banana actor as a sensor:

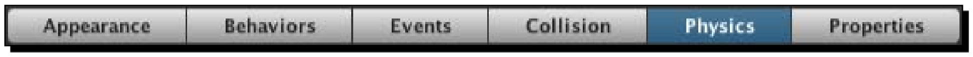
1. Display the banana actor in the Animation Editor (click on the **Dashboard** tab, then **Actor Types**, and double-click on the **Stencyl Book Banana** thumbnail).
2. Ensure that the **Collision** button is selected in the row of buttons at the upper-center of the screen.
3. Click on the left collision circle on the banana so that it is selected.
4. In the right panel, under the heading **Physical Properties**, click on the **Is a Sensor?** option so that a checkmark is shown as follows:


Physical Properties

Is a Sensor? ✓

Allows shapes to detect collisions without producing
a response, so they do not appear to be solid.

5. Click on the right collision circle on the banana so that it is selected.
6. Click on the **Is a Sensor?** option so that a checkmark is shown.
7. Click on the **Physics** button in the row of buttons at the upper-center of the screen.



8. Change the **Affected by Gravity?** setting to **No**.
9. Test the game; make the monkey collide with the banana (it will run through it)!

What just happened?

We have changed the banana actor into a sensor; our game will still detect collisions between the banana and the monkey and cause a collision event, but the collision will not cause a physical reaction in the game. It's important to note that, because the banana has two collision shapes (two circles), we had to configure both of those shapes as sensors.

Because the banana is now a sensor, we also needed to change a physics setting so that it will no longer be affected by the scene's gravity. If we do not tell sensors to ignore gravity, they will fall straight through the tiles!

Don't these changes put us right back where we started? Before we created the **Collectible** collision group, the monkey ran straight through the banana actor, and that's what's happening now!

Although it looks like we have taken a step backwards, behind the scenes Stencyl is still detecting the collision, but instead of making the monkey bump into the banana actor, it is just sensing when the collision occurs and creating a collision event. This means that we can use or create behaviors that depend upon a collision being sensed between the monkey and the banana actors, and that's exactly what we need in our game!

Have a go hero

The game file to import and load for this session is `5961_03_11.stencyl`.

We have only configured the banana to be a sensor, but we also have grapes, a melon, and a pineapple, which should also be configured in the same way, so go ahead and configure each of the fruit actors as sensors. Remember to change each actor's **Affected by Gravity**? setting to **No** so the actors don't fall through the tiles.

When this task has been completed, our monkey should be able to run through each piece of fruit without bumping into it!

Implementing terrain collision shapes

We've already learned about setting up collisions for tilesets and actors, but there is a third type of collision that we can use in the design of our games.

Terrain collision areas enable us to set up arbitrary areas within our scene that actors can collide with. This feature can be helpful if there are large areas of tiles that do not have collisions defined, but it is also very useful when a scene has not been designed using tiles; for example, a background may consist of a single artistic image onto which collision areas can be placed using the terrain feature.

Let's implement a practical example, so we can see how it works.

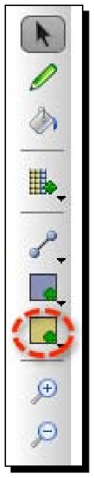
Currently, our monkey can run all the way to the edges of the scene, and when it gets there, it just bumps against nothing but the imaginary edge of the scene. Rather than modify the tiles' collision shapes in the Tileset Editor, we're going to draw custom terrain shapes over the green pillars that we placed at the extreme ends of the scene, to prevent the monkey from running through them.

Time for action – adding a terrain collision area to the scene

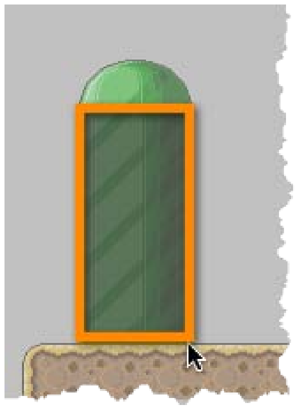
The game file to import and load for this session is `5961_03_12.stencyl`.

We're going to be working in the Scene Designer.

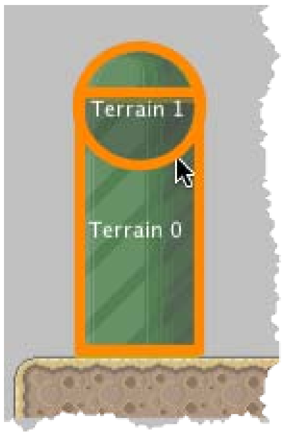
1. Go to the **Dashboard** tab, click on **Scenes**, and double-click on the thumbnail for the jungle scene.
2. Ensure that we can see the leftmost part of the scene (scroll all the way to the left).
3. Click on the **Add Terrain** tool in the vertical toolbar which can be found at the left edge of the Scene Designer (highlighted by the dotted oval in the following screenshot):



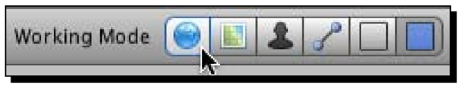
4. Referring to the following screenshot, click-and-drag a rectangle over the green pillar at the left of the scene, as shown:



5. Right-click on the same **Add Terrain** tool on the **Scene Designer** toolbar and select **Add Terrain (Circle)** from the pop-up menu.
6. Click-and-drag a circle over the curved section of the green pillar, as shown:



7. If necessary, it is easy to adjust the position of the new terrain shape by clicking on the shape and using the arrow keys. If the shapes need redrawing, they can be deleted by clicking on them and pressing the *Delete* key on the keyboard.
8. On the **Working Mode** section of the horizontal toolbar at the top of the Scene Designer, click on the **Work with Tiles and Actors** (globe) icon.



9. Test the game and try to make the monkey run past the pillar at the left of the scene.

What just happened?

We have created two custom terrain collision shapes within our scene: one rectangle and one circle. The shapes were positioned to overlay scenery that did not have its own collision bounds, but we could have placed our custom terrain shapes absolutely anywhere within the scene, if required.

When testing the game, we can see that the monkey can no longer run past the green pillar at the left side of the scene; it just bumps into the pillar as though it were another actor or a collection of tiles with the appropriate collision bounds.

Have a go hero

The game file to import and load for this session is `5961_03_13.stencyl`.

There is a second pillar at the rightmost edge of the scene, and this should also have custom terrain objects drawn over it so that the monkey cannot run past the pillar, to the edge of the screen.

- Use the skills that we have just learned to draw custom terrain shapes over the pillar at the rightmost edge of the scene.

Remember to test the game regularly during the design process to ensure that the monkey can traverse the scene as planned.

What else can we improve?

When testing the latest changes, if we had controlled the monkey so that it jumped onto the various platforms and then onto the pillar, we would have found that we could make the monkey land on top of the pillar! Is that something that we want the monkey to do? If not, we'll need to make the pillars taller by moving and adding tiles in the Scene Designer. If we modify the tiles that are used to construct the pillar, then we'll also need to redraw the custom terrain shapes to cover the modified pillars.

Again, it's up to us as game designers to determine a scene layout that works, is fun and challenging (but not too challenging) for players to explore, and which does not exhibit any strange behavior, such as being able to make actors jump onto edge-of-scene markers (for example, our pillars).

Summary

The development of our game and our game development skills are progressing at a great pace!

In this chapter, we have learned about three different types of **collision bounds** that can be implemented within Stencyl games:

- Actor collision shapes
- Tile collision shapes
- Custom terrain shapes

We've also seen how we can create very accurate collision shapes in our game, and we have learned about setting up **collision groups**, so that collisions are only detected when necessary, thereby improving the performance of our game.

Another useful feature that we have examined is the ability to configure an actor's collision shapes as sensors, so that although a physical collision does not take place, our game-code will still be able to react when a collision with a sensor occurs.

Although collisions are now being detected in our game, we still have some work to do! We need to make the game more interesting by providing some feedback to the player when these collisions occur. What should happen when the monkey bumps into an enemy actor? What should happen when the monkey bumps into the collectible fruit actors?

We'll find out the answers to these questions in *Chapter 4*, *Creating Behaviors*.

4
Creating Behaviors

In Chapter 3, Detecting Collisions, we configured collision shapes for the actors and tiles so that collisions are detected when required, and our game now allows the player to traverse the scene with some basic, predictable interactivity. However, although the monkey is now colliding with other actors, there are currently no responses to these collisions, other than a bump when some of the actors collide.

We need to take control of the game, so in this chapter, we're going to learn how to create instructions that will carry out specific actions when collisions and other events occur within our game.

In this chapter we will be:

- Creating custom behaviors
- Understanding the instruction block palette
- Creating a timed event
- Examining screen size and scene size
- Introducing randomness into our game
- Implementing our first special effect
- Understanding active actors
- Creating a countdown timer
- Implementing decision making in our game
- Repositioning an actor during gameplay
- Triggering custom events in our behaviors

Creating custom behaviors

When an event such as a collision occurs during gameplay, we would like to be able to decide exactly what happens, rather than being limited to using the built-in behaviors that have been provided for us. We're going to create our own custom behaviors that;

- Respond to collisions and other events
- Use a random number to make gameplay less predictable
- Utilize a timer to carry out actions at regular or delayed intervals
- Implement custom events that can be triggered by other behaviors

Creating our first custom behavior

As we discovered in the *Using behaviors to interact with our game* section in *Chapter 2*, *Let's Make a Game!*, behaviors are the instructions and rules that our game will use. We've already used some behaviors that have been built for us and are provided with Stencyl:

- Camera Follow
- Cannot Exit Scene
- Jump and Run Movement

We found the previously mentioned behaviors in the built-in behavior library, and we attached them to the monkey actor so that it would follow the specified instructions. Up until now, we haven't concerned ourselves with how these behaviors work. We've just accepted that they do the required job, but now it's time to learn about what is going on in the background, and to create a custom behavior of our own.

Currently, when our monkey runs past a piece of fruit, nothing happens, not even a bump, because in *Chapter 2*, *Let's Make a Game!,* we configured all the fruit actors to be **sensors**.

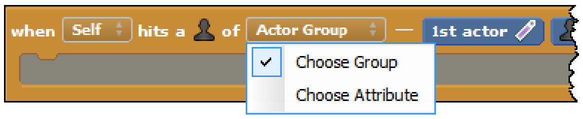
However, we would like something to happen. In this case, we want the fruit to disappear so it can't be collected twice.

Time for action – creating a behavior

The game file that needs to be imported and loaded for this session is `5961_04_01.stencyl`.

We are going to use Stencyl's **kill...** instruction block to remove the collected fruit from the game, remembering that each fruit actor is a member of the `Collectibles` group. Execute the following steps in order to create a behavior:

1. On the **Dashboard**, under the **LOGIC** heading, click on **Actor Behaviors**.
2. Click on the green **Create New** button situated at the top of the right-hand panel.
3. In the **Create New...** dialog box, enter the words `Collect Fruit` into the **Name** textbox and click on the **Create** button.
4. The behavior editor will be displayed.
5. Click on the **+ Add Event** button at the top of the left-hand panel.
6. In the pop-up menu that appears, move the mouse over **Collisions**.
7. In the next pop-up menu, in the **Any actor collides with...** section, click on **Member of Group**.
8. Double-click on the **Actor – Group** event label towards the top of the left-hand panel (immediately under the **+ Add Event** button).
9. Change the text to **Collides with Collectibles** and press *Enter* on the keyboard.
10. In the orange **when Actor hits a ...** block, displayed in the center panel, click on the **Actor Group** selector and click on **Choose Group** in the pop-up menu, as shown in the following screenshot:



11. In the **Choose an Actor Group** dialog box, select **Collectibles** and click on **OK**.
12. Save the game. It's not ready for testing yet!

What just happened?

We have created a new actor behavior called `Collect Fruit`, which has a single empty event called **Collides with Collectibles**, but we haven't provided any instructions yet.

The event was initially called **Actor – Group** and although that is quite descriptive (we're responding to a collision between an actor and a group), we might have several **Actor – Group** collision events, for example, we will also need to check for collisions with actors in the **Enemies** group. To make it easier to identify what each event does, it's a good practice to give each event a descriptive name and, in this case, **Collides with Collectibles** is an appropriate name, because it describes the event that we're listening for.

[Note that this name is not used by Stencyl for referencing the event. It is simply a human-readable label, with the purpose of helping the developer to easily identify the events that we have created.]

Stencyl behaviors can listen out for many different types of events, but we have chosen to listen for a very specific event, which is triggered when the actor collides with any actor in the specified group. When we clicked on the **Actor Group** option in the orange block, and specified the `Collectibles` group, we were telling the behavior to listen out for a collision event between this actor and any actor that is a member of the `Collectibles` group.

The orange **when** block in our new `Collect Fruit` behavior will listen out for a collision between **Self** and the `Collectibles` group, but which actor is the **Self** instruction referring to? The answer to this question demonstrates the beauty of Stencyl behaviors, because **Self** is whichever actor we attach the behavior to and we can attach it to any type of actor that we want. In other words, behaviors are reusable, they are not limited to being attached to a single type of actor. They can be attached to as many different actor types as we wish.

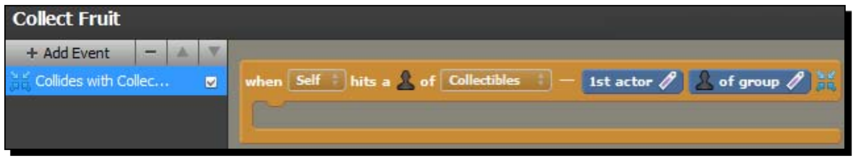
[The reusability of behaviors within Stencyl can be compared to **Object Oriented Programming** (**OOP**) methodologies in traditional programming languages.]

Currently, the behavior is not attached to an actor and we have not specified what action we want to take place when the collision event occurs, so let's do that now.

Time for action – adding an action and attaching to it an actor

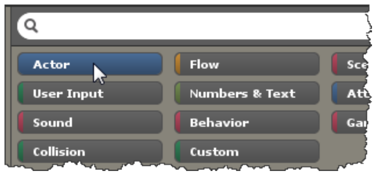
The game file that needs to be imported and loaded for this session is `5961_04_02.stencyl`.

Ensure that the Behavior editor is currently displaying the orange **when** block for the **Collides with Collectibles** event in the `Collect Fruit` behavior, as shown in the following screenshot:

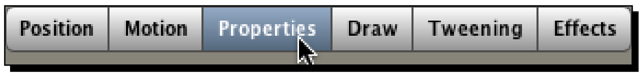


When following the given steps, carefully refer to the screenshots, as they will assist you in correctly placing the blocks.

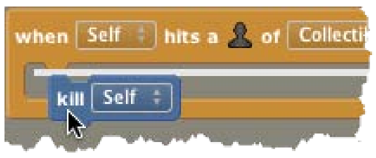
1. Click on the **Actor** button above the instruction block palette at the top of the right-hand panel, to ensure that it is selected, as shown in the following screenshot:



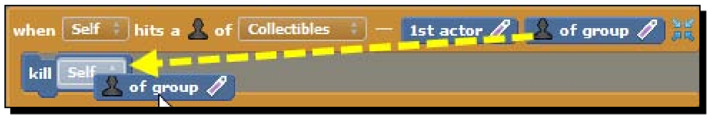
2. Click on the **Properties** category button in the row of buttons that is displayed immediately below the top buttons in the instruction block palette, as shown in the following screenshot:



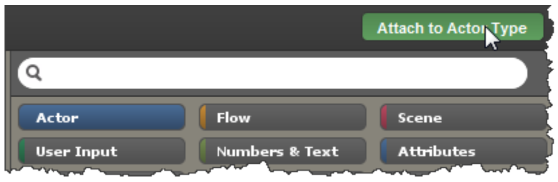
3. Locate the blue **kill Self** block in the **Alive / Dead** subcategory of the right-hand panel.
4. Click and drag the blue **kill Self** block, and drop it into the blank area of the orange **when Self hits a...** block in the center panel as shown in the following screenshot. When the white snap-line appears, let go of the mouse button. This will allow the block to snap into place:



5. Locate the blue **of group** block at the top-right of the orange **Self hits a ... of Collectibles** block, and drag it onto the word **Self**, which is inside the blue **kill Self** block that we just added, as shown in the following screenshot:

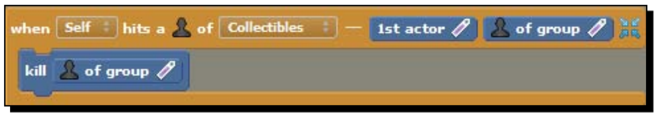


6. Click on the green **Attach to Actor Type** button at the top-right of the block palette as per the following screenshot:



7. In the **Choose an Actor Type** dialog box, double-click on the **Stencyl Book Monkey** icon, the behavior list for the monkey will be displayed.

8. Click on the `Collect Fruit` tab to return to the completed event, which should now look like the following screenshot:



9. Test the game. Make the monkey collide with a piece of fruit.

What just happened?

When the monkey collides with a piece of fruit, the fruit disappears!

We had already created the `Fruit Collected` behavior, and had specified that it must listen out for a collision event between `Self` (the monkey) and any member of the `Collectibles` group. However, we had not specified which action should take place when the collision occurred, so we added the `kill Self` instruction block to the `when Self hits a... of Collectibles` event, and configured it to remove the actor that we just collided with. The blue `of group` block is a label that provides a reference to the other actor involved in the collision, that is the actor we want to kill.

Finally, we attached the behavior to the monkey actor. Now, in our game, when the behavior runs, any references to `Self` in the behavior blocks will be considered to mean this monkey actor, and any references to `of group` will refer to the member of the `Collectibles` group that the monkey collided with.

[A very common mistake is to forget the final step, attaching the behavior to the appropriate actor! If a behavior appears not to run at all, this is the first mistake to check for.]

Sometimes when developing or reviewing the design of a behavior, it can be helpful to express it in English, rather than in technical terms. If we were to write our new event in English, it could be stated as follows:

When this actor hits a member of Collectibles, kill the Collectibles actor that we just collided with.

In the previous sentence, the phrase before the comma is the event that is being listened to in the orange `when self hits a ... of Collectibles:` block. The phrase after the comma is the action that is to be carried out when the event is triggered, that is the instruction in the blue `kill Self` block.

Almost all behaviors in Stencyl are designed with this basic principle—what am I listening for, and what should I do when it happens? Behaviors can be far more complicated than this, but this is the essence of most behaviors.

The actor's behavior screen

When we attached the `Collect Fruit` behavior to the monkey actor, the actor's behavior screen was displayed. However, we switched back to the `Collides with Collectibles` event, so that we could compare it with the screenshot. Let's return to the monkey's behavior screen by opening the monkey actor from the **Dashboard** option, and by clicking on the **Behaviors** button at the top of the right-hand panel.

The actor's behavior page provides some information that is useful for us to know. If we look at the left-hand panel, we can see a list of behaviors that are currently attached to this actor:

- Jump and Run Movement
- Cannot Exit Scene
- Camera Follow
- Collect Fruit

Also, at the top of the right-hand panel, there is a red **Deactivate Behavior** button, which can be used to disable the currently selected behavior. This feature can be useful for debugging because we can disable behaviors during the process of tracking down a problem. It's also useful to know that behaviors can be deactivated and reactivated within instruction blocks. This is a more advanced feature but it's a feature that's worth remembering for future use.

Adding an additional event to a behavior

We have created an event that makes fruit disappear when the monkey collects it, and we now need to add a new event that will kill the monkey when it collides with an enemy actor.

It is quite possible to create a brand new behavior that will do this job, but we're going to add an additional event to the behavior that we have just created. This means that the behavior will have two events to listen out for, namely:

- A collision between the monkey and a member of the `Collectibles` group
- A collision between the monkey and a member of the `Enemies` group

Originally, we created the behavior to do one job, and we gave it a sensible name `Collect Fruit`. However, as we're going to add a second event that is not related to fruit collection, this name won't make sense any more, so we need to give the behavior a more relevant name.

Time for action – adding an event and renaming the behavior

The game file that needs to be imported and loaded for this session is `5961_04_03.stencyl`.

If we're following on immediately from the previous instructions, we should be looking at the monkey actor's behavior page. However, we need to return to the behavior editor so, click on the `Collect Fruit` tab or, if the tab is not visible, go to the **Dashboard**, click on **Actor Behaviors** (under **LOGIC** in the left-hand panel) and double-click on the **Collect Fruit** behavior, which is currently represented by an image of a hammer.

We can now see the orange **when …** block that we created earlier. Now perform the following steps:

1. Click on the **+ Add Event** button at the top of the left-hand panel.
2. In the pop-up menu, move the mouse over the **Collisions** option, and select **Member of Group**.
3. Double-click the new **Actor – Group** item in the left panel.
4. Change the text to `Collides with Enemies` and press *Enter* on the keyboard.
5. On the orange **when Self hits a…** block, change the **Actor Group** option to **Enemies** by clicking on it, selecting **Choose Group** and double-clicking **Enemies** in the **Choose an Actor Group** dialog box.
6. In the right-hand panel, ensure that the **Actor** button is selected at the top of the panel, click on **Properties** from the row of silver buttons in the right-panel, and drag the blue **kill Self** block into the orange **when Self hits a…** block.

7. In the row of buttons shown at the upper-center of the screen, click on the **Properties** button, as shown in the following screenshot:

8. In the **Edit Properties** dialog box, change the name from `Collect Fruit` to `Manage Player Collisions`.
9. In the **Description** box, type `Manages all collisions for the player's character`.

[Note that we could also change the icon for this behavior from the default hammer to any image of our choice, although we're going to leave it as it is.]

10. Click on the **Apply Changes** button.
11. Test the game. First, make the monkey run into a piece of fruit, then make it bump into one of the enemy characters.

What just happened?

We have added a new event to the behavior that we created earlier, and we renamed the behavior.

The procedure for creating the `Collides with Enemies` event is very similar to the one we followed when creating the `Collides with Collectibles` event. We specified what type of event we wanted to create. In this case it is another group collision. Then, we specified which group we wanted to detect the collisions with, by changing the `Actor Group` to `Enemies` in the orange `when Self hits a…` block.

The next step was to tell the event what to do when a collision occurs with the `Enemies` group—we need to kill the actor—so we dragged the `kill Self` block into the orange `when Self hits a…` block.

We might ask ourselves the question, "Which actor is going to be killed?" To find the answer, we first need to know that the `kill Self` block is going to kill `Self`. Remembering that this behavior is attached to the monkey actor, we can determine that `Self` refers to the monkey. So, the monkey will be killed when the collision occurs.

This is an important difference when compared to the `Collides with Collectibles` event because, in that event, it is the actor `of group`—the member of the `kill Self` group—that was killed.

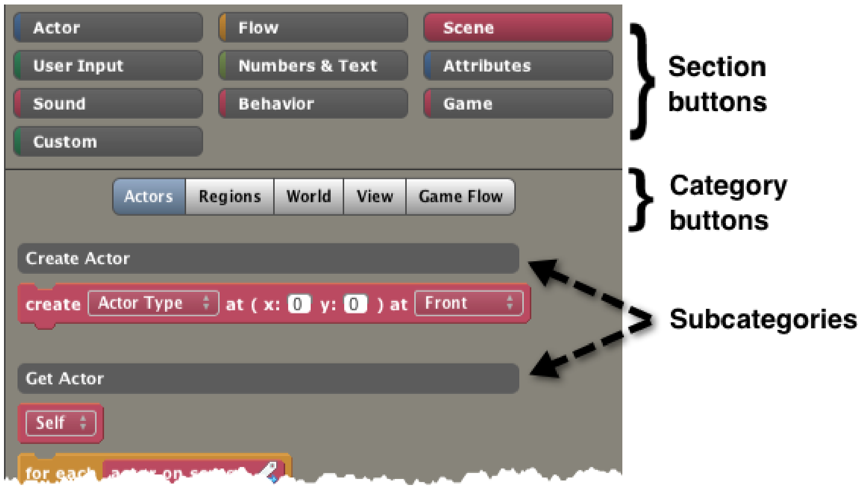
The final step was to give the behavior a more relevant name, it doesn't just manage collisions with the fruit anymore. It manages all of the monkey's collisions so the name `Manage Player Collisions` seems more appropriate, and will help us to more easily identify the behavior if we need to make changes in the future.

When we test the game, we will find that both events are doing their jobs—when our monkey touches a piece of fruit, the fruit disappears but, when the monkey runs into one of the enemy actors, the monkey is killed.

Understanding the instruction block palette

When we are creating or editing behaviors, we have access to the block palette, the rightmost panel from which we have been selecting the instruction blocks.

The palette has been arranged so that it is easy to locate the blocks that we require, and it is organized as shown in the following screenshot:



We have already used the section buttons to select blocks from the **Actor** and **Scene** sections, but it is worth noting that each section has different category buttons. For example, in the previous screenshot, we can see that the currently selected **Scene** section has five categories namely **Actors**, **Regions**, **World**, **View** and **Game Flow**. Underneath the category buttons, the blocks are separated into subcategories so, for example, the **Actors** category, in the **Scene** section, has the subcategories **Create Actor** and **Get Actor**.

If we click on the **Actor** section button, we will have a different selection of categories from which we can choose—**Position**, **Motion**, **Properties**, **Draw**, **Tweening**, and **Effects**. If we then click on the **Position** category button, we can see that the subcategories are **Position** and **Direction**.

As we learn more about Stencyl behaviors, we will become familiar with the sections, categories and subcategories, but one of the keys to mastering Stencyl is to understand which all instruction blocks are available to us, and where to find them.

To learn more about how individual instruction blocks work, right-click on the block in the palette, and select **View Help**. This will open up the relevant help page on the `stencyl.com` website.

Considering future refinements

We've created a new behavior for the monkey, and it correctly manages the monkey's collisions, but we still have some work to do in order to refine the actions that take place when the collisions occur.

For example, rather than fruit disappearing instantly when it is collected, it might be more visually appealing if a special effect occurred. Perhaps the fruit could fade away gently or, for a more dramatic effect, it might explode, with the pieces fading away.

When the monkey collides with an enemy actor, the monkey vanishes without warning, which isn't very elegant, so we'll need to make some decisions about how that process is handled.

However, rather than holding ourselves back with the technicalities of special effects, we're going to carry on building the important mechanics of the game, and we'll consider options for some of the refinements and implement them later in the development process.

A review of the gameplay

We are making fantastic progress with our game; we're only one quarter of the way through this chapter, and we have implemented several features and learned several new skills.

However, in its current state, the gameplay is not going to be very exciting for players of Monkey Run. The game will be exactly the same every time it is played and, as a designer, our job is to create a game that people will want to come back to play time and time again.

There is an important element missing from our game, and that is the element of chance, our game needs some randomness!

Random gameplay is very difficult to get just right. If a game has too many random elements, it will be too difficult to play, our players will give up very quickly and move on to more interesting games. For this reason, even when we, as designers are happy with the gameplay, it is vital that we find the opportunity to allow independent, unbiased game testers to provide us with feedback, so we can fine-tune the game and make it enjoyable to play.

Introducing a new challenge

We're going to introduce a new actor into our game—a mysterious Aztec statue that will fall from the sky at random locations and block the monkey's path. Our first task is to find the statue on StencylForge, download it and configure it for our game.

Because we are going to be building a more complex behavior in this section of the book, rather than jumping in and trying to create the whole behavior in one session, we'll break down the process so we can be sure that each step is working correctly.

As we create the behavior step-by-step, we'll notice that some things aren't working quite as they should. Don't worry we'll fix the problems as we work our way through the development process.

Building behaviors step-by-step is a good practice, jumping in and trying to create a complex behavior without regular testing can be a recipe for disaster. So, it's a good idea to get into the habit of working through the development process methodically, rather than rushing ahead and introducing difficult to fix bugs.

Have a go hero – downloading and configuring the statue

The game file that needs to be imported and loaded for this session is `5961_04_04.stencyl`.

The skills required to complete the following tasks were covered in *Chapter 3*, *Detecting Collisions*, so if required, refer to the relevant sections and review the detailed steps to complete the tasks listed.

The tasks we need to achieve here are:

- Downloading the actor called **Stencyl Book Statue** from StencylForge
- Creating a new collision group called `Droppers`
- Configuring the new `Droppers` group to collide with `Players`, `Tiles`, `Enemies`, and `Droppers`
- Adding the statue to the `Droppers` collision group—this is a vital step, so don't forget to complete it
- Saving the game, so we're ready for the next section, *Time for action – creating a behavior to drop the statues*

We now have a statue actor that is a member of a new group called `Droppers`, and we have configured it to collide with actors that are members of the specified groups.

Creating a timed event

We've already decided that we are going to drop the Aztec statues into the jungle at random locations but, first we need to decide how often our game is going to drop them. So the first step is to automatically create the actor inside the jungle scene every few seconds.

Time for action – creating a behavior to drop the statues

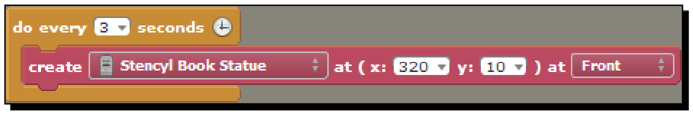
The game file that needs to be imported and loaded for this session is `5961_04_05.stencyl`.

We're going to start by creating a behavior to drop the statues into a single location in the scene, and then we'll refine the behavior as we progress. Execute the following steps in order to create a behavior to drop the statue:

1. On the **Dashboard**, under the **LOGIC** heading, click on **Scene Behaviors**.
2. Click on the message **This game contains no Logic. Click here to create one.** in the main panel.
3. In the **Name** textbox, type `Drop Actors Randomly` and click on the **Create** button to display the behavior editor.
4. At the top of the left-hand panel, click on **+ Add Event**.
5. In the pop-up menu, move the mouse over **Time**, then click **Every N seconds** (*not* **After N seconds**).
6. Double-click on the **Every N secs** item in the left-hand panel, and change the name of the event to `Drop statues`.
7. In the orange **do every … seconds** block, enter the digit `3` into the white textbox.
8. Click on the **Scene** button at the top of the instruction block palette, as shown in the following screenshot:

9. Drag the red **create Actor Type …** block into the orange **do every 3 seconds** block.
10. On the new red **create Actor Type at …** block, click on the **Actor Type** option and select **Choose Actor Type** from the drop-down menu.
11. In the **Choose an Actor Type** dialog box, double-click on the icon for the **Stencyl Book Statue**. The red block will change to **create Stencyl Book Statue at …**.

12. In the red **create Stencyl Book Statue at …** block, enter the number `320` into the white textbox to the right of **x:**.
13. Enter the number `10` into the white textbox to the right of **y:**, as shown in the following screenshot:



```
do every 3 seconds
    create Stencyl Book Statue at ( x: 320 y: 10 ) at Front
```

14. Click on the green **Attach to Scene** button above the instruction block palette.
15. Double-click the icon for the **Jungle** scene.
16. Test the game.

What just happened?

We've created a new scene behavior that drops the statue actor into the jungle scene every `3` seconds. Rather than being attached to an actor, this behavior is attached to the scene, because the scene is going to create the statues for us.

We needed to tell the `create Actor Type at` … block which actor to create, and where to create it, so we selected the statue actor type in the **Choose Actor Type** dialog box. The numbers that we entered in the **x:** and **y:** textboxes represent the location on the screen where the statue will be created. Although we have some work to do to position the statue correctly, the numbers we provided are adequate for testing if the behavior is carrying out its main task correctly.

Note that we don't need to manually add the statue actor to the jungle scene using the Scene Designer, as we have done with the monkey, fruit, and enemy actors, the `create Stencyl Book Statue at` … block will do all the work for us.

When we test the game, we can see that a new statue actor appears at the top-center of the screen every three seconds. However, we can quickly determine that things are not quite right. If we move the monkey, we can see that the statue is always dropping into the scene at exactly the same position, which is probably not what we're hoping for, because it won't present much of a challenge for players of our game when the monkey progresses to the other end of the scene.

Identifying and resolving problems

We've already identified one unexpected issue—the statues always fall in the same location—but it's now time to give the game a really good testing and make a note about what happens when, for example, many statues are allowed to fall into the scene. Try it.

There may be other problems to be found, but it's important to understand that these bugs aren't necessarily faults with Stencyl, they may be system limitations (that is, known features) or they could be caused by the way that we have designed the game—for example, the player's character may become stuck at a certain point in a scene if we haven't carefully considered the placing of tiles.

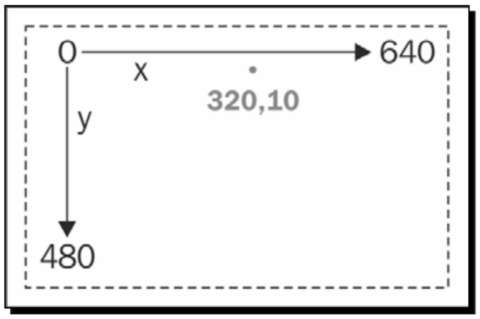
Sometimes these effects are desirable in a game but, if they are not, we will need to change our game so that it works differently, so it's a good idea to note any problems as early as possible (always keep a notepad and pencil at hand). It's all part of the learning process.

We'll fine-tune the behavior that drops the statues as we progress through this chapter but, before we can do that, we need to understand a little more about the relationship between screen and scene sizes.

Examining screen size and scene size

In our newly created behavior, we specified the **x** position as `320` and the **y** position as `10`. It's a good idea to examine what these numbers mean so that we have a good understanding of why the statue actor is appearing exactly where it does.

The coordinates on a computer screen are measured from the upper left-hand corner of the screen, which has an x position of zero and a y position of zero. As we move from the left of the screen to the right, the x coordinate increases in value. As we move from the top of the screen to the bottom, the y coordinate increases in value, as shown in the following figure, in which the dotted lines represent the edges of our screen:



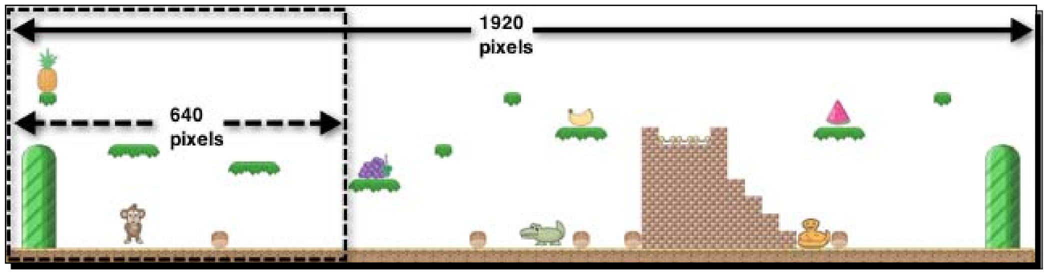
When we initially created our game, at the beginning of *Chapter 2*, *Let's Make a Game!* we accepted the default screen width of 640 pixels and screen height of 480 pixels. The screen size determines the area of the game that we can see on the computer screen at any one time. We can see this configuration by clicking on the **Settings** icon on the Stencyl toolbar, and clicking on **Setting**, and then on **Display** in the **Game Settings** dialog box.

A pixel is the smallest controllable point on a computer display. In Stencyl, although it is easy to do, it is fairly unusual to draw on the screen at pixel level. We usually create actors and tiles, or draw shapes that are constructed for us, without having to worry about individual pixels.

The point on the previous figure, labeled `320, 10` represents the x and y coordinates at which our statue is being created, that's 320 pixels horizontally from the left of the screen, and 10 pixels vertically from the top of the screen.

Later on in *Chapter 2*, *Let's Make a Game!*, when we made the jungle scene scrollable, we changed the scene width from 20 tiles to 60 tiles. In our game, each tile is 32 pixels wide, which means that we changed the width of the scene from 640 pixels wide (20 tiles x 32 pixels) right up to 1,920 pixels wide (60 tiles x 32 pixels). We can see this information about the scene width by looking at the properties dialog box for the Jungle scene. Open the Jungle scene and click on the **Properties** button at the top of the screen.

The following screenshot shows how our screen size (the area that is displayed) relates to the scene size (the whole area in which our game is played).



In the previous screenshot, the dotted rectangle, which is `640 pixels` wide, represents the size of the screen, that is, the area of the game that we can see at any one time while the game is being played, and this is also known as the **viewport**. The solid line, which is `1,920 pixels` wide, represents the width of our scene. As the monkey runs towards the right-hand side of the screen, we can imagine that the dotted black rectangle will slide along the scene in order to prevent the monkey disappearing off the side. It's the camera that we set up back in *Chapter 2*, *Let's Make a Game!*, that manages this process for us.

The reason that the statues always fall at the same location, is that we have specified a fixed x coordinate of 320 pixels (that's the horizontal location) for the statue. This means that, as our monkey runs towards the right edge of the screen, the statues will always fall at 320 pixels from the left-hand edge of the scene.

We need the statues to be dropped in the center of the screen (or viewport), so let's resolve that issue.

Time for action – adjusting the drop-location of the statue

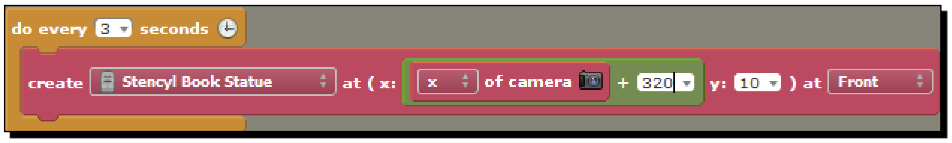
The game file that needs to be imported and loaded for this session is `5961_04_06.stencyl`.

Ensure that the `Drop Actors Randomly` behavior is displayed onscreen. Execute the following steps:

1. Click on the drop-down arrow to right of the **x** textbox (it currently contains the number `320`).
2. Move the mouse cursor over **Math** in the drop-down menu.
3. In the **Arithmetic** section of the pop-up menu, click on the **0 + 0** block.

Note that the zeroes in the new green instruction block do not appear in our event blocks, they have been replaced by empty textboxes.

4. In the left-hand blank textbox of the new **of camera + …** block, click on the drop-down arrow.
5. Select **Scene** from the drop-down menu.
6. In the **Basics** section of the pop-up menu, click on the red **x of camera** block.
7. In the right-hand blank textbox of the **of camera + …** block, enter the number `320`, as shown in the following screenshot:


do every 3 seconds
create Stencyl Book Statue at (x: x of camera + 320 y: 10) at Front

8. Test the game. Make the monkey run to the right of the scene.

What just happened?

Instead of the statues dropping at a location that is always 320 pixels from the left-hand edge of the scene, they will now always drop at a location 320 pixels from the left-hand edge of the screen.

To achieve this result, we used the `x of camera` block and a little bit of basic math (addition). The position `x of camera` is always the left-hand edge of the screen, which is zero pixels from the left of the screen. The screen width is 640 pixels, so if we add half the screen width (half of 640 = 320) to the camera's x position (0 pixels), we will always be specifying an x position of 320 pixels from the left of the screen. No matter where our monkey is, the
statue will always be dropped into the center of the viewport.

The statues will now drop from the center of the screen as required, but there are still some refinements to be made to this behavior, and we'll work on them as we progress through this chapter.

Examining the scene instruction blocks

We have been learning how to use the `create Actor Type at` … block, which we found under the **Actors** category in the **Scene** section of the block palette.

Take this opportunity to examine the other instruction blocks that can be found in the **Scene** section of the block palette—particularly those in the **View** and **Game Flow** subcategories. Many of the more advanced blocks won't necessarily make sense yet, but there are some very interesting instruction blocks for which their purpose is immediately apparent. For example, in the **View** category, there is a `shake screen for 0 sec with intensity 0 %` block which, perhaps unsurprisingly, does exactly what it says, it shakes the screen for the specified time with the specified intensity.

Make a note of any interesting blocks, and consider how they might be useful in our game. Perhaps the scene could shake each time a statue hits the ground.

Preparing for future changes

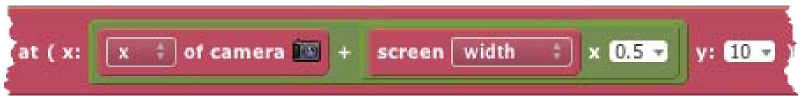
We've created a problem for the future! What if we decide to change the screen size for our game? Perhaps we decide to target a different device for our game—the screen size of 640 pixels wide and 480 pixels high might not be appropriate. We've already learned that we can easily change these parameters in the **Game Settings** dialog box, but if we change the screen width, our game isn't going to work as planned. We've specified that the statue should be dropped at exactly 320 pixels from the left-hand edge of the screen, which is half the current screen width of 640. If, for example, we change the screen width to 800 pixels, then the statue won't drop in the center of the screen because half that width is 400 pixels. It will be dropped somewhere towards left of the center, which is not the desired location! Let's fix that problem.

Time for action – making the behavior more flexible

The game file that needs to be imported and loaded for this session is `5961_06_07.stencyl`.

Ensure that the `Drop Actors Randomly` behavior is being displayed. Execute the following steps in order to make the behavior more flexible

1. Delete the number `320` from the right-hand textbox in the green addition block (it's not strictly necessary to delete this, but it keeps everything neater).
2. Click on the drop-down arrow in the same textbox from which we just deleted the number.
3. In the **Math** section of the pop-up menu, click on the **0 x 0** block.
4. Click on the drop-down arrow in the left-hand textbox of the newly added green multiplication block.
5. Select **Scene** from the drop-down menu and then, in the **Basics** section of the next pop-up menu, click on the red **screen width** block.
6. In the right-hand textbox of the green multiplication block enter the number `0.5`. Your screen should like the following screenshot:



7. Test the game.

What just happened?

Nothing appears to have changed in the game because the statues are still dropping in exactly the same location. However, we have ensured that, whatever the width of our screen, the statue will always drop in the center of the screen rather than at the fixed location that we originally entered. We have calculated the x position of the statue by adding half the width of the screen; that is, `screen width x 0.5`, to `x of camera`.

This may seem a trivial change to make, but there is another benefit. Imagine that we had 20 different behaviors that referred to the width of the screen, and that the numbers had been manually typed, as they were in our original version of the behavior. If we now imagine that we have decided to change the screen width in the **Game Settings** dialog box, then we'd have to work our way through all 20 behaviors and manually adjust the numbers! This would be a very time-consuming process and, even worse, it would increase the risk of introducing errors into our game.

Why did we multiply the screen width by 0.5, when we could have divided the screen width by 2, in order to achieve the same result? In theory, computer code can often multiply faster than it can divide, so some developers consider it to be a more optimized calculation. However, in practice, it's unlikely to have any effect on a game developed with Stencyl. The choice of whether to multiply or divide is entirely down to the developer.

There is a potential disadvantage to calculating the x coordinate for the center of the screen, rather than manually typing the number into the instruction block. The computer has to calculate the x coordinate based on the width of the screen every time a statue is dropped, which uses processing power. This is unlikely to cause a problem for us, but it does demonstrate that game designers have a lot of issues to consider when building a game!

Introducing randomness into our game

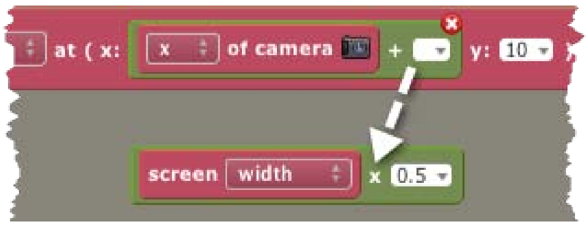
We've now learned how to do calculations with Stencyl's instruction blocks, and we have also learned about the **screen width** block, which can be used to dynamically retrieve the width of the screen. However, earlier in this chapter, the issue of randomness was raised, but we haven't actually introduced that element of gameplay, so let's do that now.

Time for action – introducing randomness to our behavior

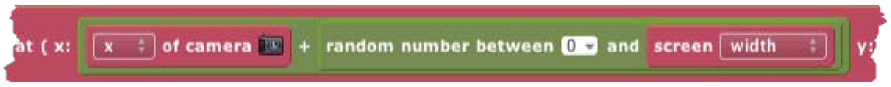
The game file that needs to be imported and loaded for this session is `5961_04_08.stencyl`.

We're going to modify our instructions so that the statues are dropped at a random location, so ensure that the `Drop Actors Randomly` behavior is being displayed. Execute the following steps in order to introduce randomness to our behavior:

1. Click and drag the green multiplication block out of the addition block, and drop it onto the gray background, below the orange **do every 3 seconds** block, as shown in the following screenshot:



2. Click on the drop-down arrow in the now empty textbox in the green **x of camera + … block**.
3. Select **Math** from the drop-down menu and then, in the **Operations** section of the pop-up menu, click on the **random number between …** block.
4. Enter the number `0` into the textbox immediately after the word **between**.
5. Click on the drop-down arrow in the textbox immediately after the word **and**.
6. From the drop-down menu, select **Scene**, and then click on the **screen width** block:



7. Press *Ctrl* + *K* or, on Mac OS X press *Command* + *K*.
8. Test the game. Make the monkey run left and right, and watch as several statues fall from the sky.

What just happened?

The actors are now falling at a random location somewhere between the leftmost edge of the screen and the rightmost edge of the screen.

The first step was to drag the **screen width x 0.5** blocks out of the behavior because we no longer needed them. We then added the **random number between … and …** block, which allows us to specify a low number and high number, and then uses this information to generate a random number within that range. We specified a lower number of `0`, which is the x coordinate for the left-hand side of the screen, and a higher number of **screen width**, which is the x coordinate for right-hand side of the screen. When our game is running, the **random number between 0 and screen width** instruction will automatically generate for us a random x position between zero and 640. The random number is then added to the `x of camera` to ensure that the statues always fall within the viewport.

Finally, we pressed the key combination of *Ctrl* + *K*, or *Command* + *K* on Mac OS X. This shortcut removes any unused blocks from the behavior designer, such as the ones that we dragged out of the behavior in the first step.

Unused blocks don't have to be removed from the behavior designer, they are ignored when the game is compiled, and they don't use any extra memory in our compiled game. However, it's a good idea to tidy up the behavior editor when the blocks are no longer required.

Optimizing the number of actors

Let's solve two more problems. The first problem is a technical one; depending on the speed of the computer's processor, after approximately 10 to 20 statues have been dropped, the game starts to lag, it slows down to an unacceptable speed because there are too many active actors on screen. Computers have limited resources and it is our job, as the game developer, to use these resources effectively.

The second problem is a gameplay issue; if too many statues have fallen, it can become impossible to complete the level, because the statues might block the monkey's path through the jungle. In some games, this may be the desired gameplay, but we want to create a game that is a little less stressful for our player on the first level, perhaps gameplay could become more difficult in later levels, but not just yet.

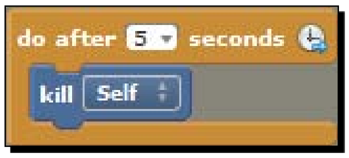
There are many ways to resolve such problems in a game, and it's usually down to the imagination of the developer to find a solution. We're going to solve both problems simultaneously, with a new behavior.

Time for action – making the statues disappear after a delay

The game file that needs to be imported and loaded for this session is `5961_04_09.stencyl`.

Let's create a timer that will remove each statue from the scene after a specified delay, Now perform the following instructions:

1. If the **Drop Actors Randomly** tab is open, then close the tab.
2. On the **Dashboard**, click on **Actor Behaviors**.
3. Create a new actor behavior called **Manage Statues**.
4. Click on **+Add Event** and select **Time** from the menu.
5. Click on the **After N seconds** icon.
6. Change the name of the **After N Seconds** event to `Kill statues`.
7. In the **do after … seconds** block, enter the number `5` into the textbox.
8. The **Actor** section button in the instruction block palette should already be selected, so click on the gray **Properties** category button in the palette.
9. Drag the blue **kill Self** block onto the orange **do after 5 seconds** block, as shown in the following screenshot:



10. Click on the green **Attach to Actor Type** button above the instruction block palette, and double-click on the **Stencyl Book Statue** in the **Choose an Actor Type** dialog box.
11. Test the game. Watch several statues fall and wait for at least five seconds.

What just happened?

We created a new actor behavior, which automatically kills the actor five seconds after it is created, and we attached the new behavior to the statue actor.

It was an easy behavior to create and it does a very straightforward job, however, the result is not very elegant—the statues just vanish! As with other aspects of the game, this may be the desired effect, but instantly vanishing actors might be rather confusing for players of a game. Perhaps we can implement a more elegant solution.

Implementing our first special effect

Rather than have our statues disappear instantly, we're going to make them gently fade away.

Time for action – making the statues disappear after a delay

The game file that needs to be imported and loaded for this session is `5961_04_10.stencyl`.

Ensure that the `Manage Statues` behavior is displayed on screen.

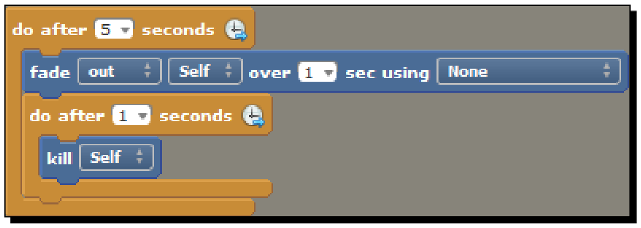
Towards the end of this *Time for action* section, there is a screenshot displaying the completed behavior, and this can be referred to while completing the following steps:

1. Drag the **kill Self** block out of the orange **do after 5 seconds** block, and drop it onto the gray background below the orange block.
2. Ensure that the **Actor** section button is selected at the top of the instruction block palette, and click on the gray **Tweening** category button.
3. Drag the **fade in Self over 0 sec using None** block into the **do after 5 seconds** block.
4. In the **fade in Self over ... sec using None** block, click on the **in** option and change it to **out**.
5. Enter the number `1` into the **over** textbox.
6. Display the flow instruction blocks by clicking on the **Flow** section button at the top of the instruction block palette, and click on the **Time** category button.

7. Drag the **do after 0 seconds** block from the instruction block palette and place it underneath the blue **fade out …** block.

8. In the new **do after … seconds** block, change the textbox to `1`.

9. Drag the blue **kill Self** block into the new **do after 1 seconds** block:



```
do after 5 seconds
    fade out Self over 1 sec using None
    do after 1 seconds
        kill Self
```

10. Test the game. Watch what happens to each statue after five seconds.

What just happened?

We have modified the behavior so that, instead of just killing the statue instantly, it will gently fade the statue out of view and then kill the actor.

It's important to note that when the statue has completely faded out of view, it still exists (it's just invisible!), which is why we need the **kill Self** block. We also need to understand that while the **fade out Self over 1 sec using None** block is doing its job, it does not wait for the fading to complete before the behavior moves on to the next instruction. That's why we have had to put the **kill Self** block inside an additional **do after 1 seconds** block. If we fail to delay the **kill Self** instruction, then the **fade out Self over 1 sec using None** block will start to fade the statue, but the **kill Self** block will instantly remove the statue after the fade process has started and before **the fade out Self over 1 sec using None** block has completed its work.

By introducing this new behavior, we have fixed two problems—the gameplay problem, whereby statues could block the monkey's path, and also the technical problem, which caused the game to lag when too many statues were on the screen. Lateral thinking such as this, will often help us to resolve gameplay and technical problems. So, bear this in mind in the cases when a difficult technical or gameplay issue occurs during the development process. Consider that there might be a straightforward solution that can also make the game more interesting and fun to play.

Experimenting with the timings

The game file that needs to be imported and loaded for this session is `5961_04_11.stencyl`.

In the `Kill statues` event, we have three different timings to consider:

1. The main `do after 5 seconds` event block, which determines how long the event should wait after the creation of the statue and before triggering the enclosed instruction blocks.
2. The `fade out Self over 1 sec using None` block, which determines over how long the fade will occur.
3. The additional `do after 1 seconds` block, which specifies how long to wait before the statue is removed from the scene. In practice, for this effect, the delay should always be the same as the delay for the `fade out Self over 1 sec using None` block.

Experiment with the timings of the blocks to get a feel of how it affects the gameplay, and decide upon a set of timings that works well in the game.

Another interesting experiment would be to edit the `Drop Actors Randomly` scene behavior, so the statues appear more quickly. Reducing the `do every … seconds` delay in the `Drop statues` event will make the statues appear more frequently, and, if the fade and kill delays are sped up, the game becomes far more manic. Consider how changes to these arguments (the numbers and textual information that affect the way that some instruction blocks work) can change the gameplay. Often, an increase in difficulty in later levels can easily be achieved, simply by changing the timings in a behavior.

It might be a good idea to ask someone else to test the game in its current state, and listen to what they have to say about their first impressions of the game. Make a note of these comments and keep them. There will always be an opportunity to come back to this behavior at a later stage of the development to make further adjustments.

Ensure that, after experimenting, the game is tested thoroughly and is working correctly.

Have a go hero – making the fruit fade when collected

The game file that needs to be imported and loaded for this session is `5961_04_11.stencyl`.

Earlier in this chapter, we created an actor behavior called `Manage Player Collisions`, in which the `Collides with Collectibles` event removes the fruit from the scene when the monkey collides with it.

Try modifying the `Collides with Collectibles` event so that it fades out the fruit rather than killing it instantly. Refer to the previous *Time for action – making the statues disappear after a delay* section, in which we changed the `Manage Statues` behavior to fade out the statues. The code is very similar.

> Right-clicking on an existing instruction block in a behavior, and selecting **Copy** from the drop-down menu, places the block into the Stencyl clipboard. The copied blocks can then be pasted (right-click, **Paste**) into the gray background area of a different behavior, and then dragged and dropped into the required location.

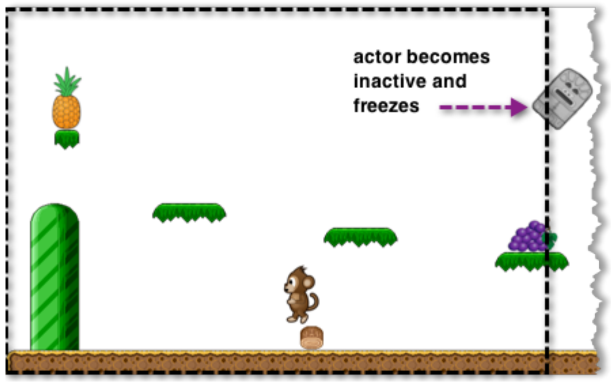
Try different timings for the fading of the fruit and note that timings do not have to be whole numbers. It is acceptable to specify fractions of a second, for example, `do after 0.5 seconds`.

Understanding active actors

Stencyl does its best to assist us in optimizing our games. For example, if we have a moving actor in our scene, but it leaves our screen, Stencyl will make the actor **inactive**, so that it isn't wasting our computer's processing resources on unnecessary physics calculations. The actor will be made **active** again when it reappears in our scene.

Although this optimization feature is very useful, it can cause problems with certain forms of gameplay. For example, we may want our offscreen actors to remain active, even though they have left the screen. In our game, we may occasionally see that this built-in optimization becomes a problem if a statue falls at an angle, or on its side, at the very edge of the screen. The statue can freeze in midair.

In the following screenshot, the dotted rectangle represents the screen that we can see during gameplay. The statue is falling such that it is far enough away from the edge of the screen to make the actor inactive, but close enough that we can still see the corner of the statue, frozen onscreen.



This phenomenon is difficult to see when testing our game. Our monkey has to be exactly in the right place on the screen, and running in the right direction, when the statue falls. While this might happen very infrequently, the chances are that it will happen eventually, and players of our game will not be impressed to see the game's actors freezing unexpectedly.

Experiencing a freezing statue

The game file that needs to be imported and loaded for this session is `5961_04_11_freeze_demo.stencyl`.

Rather than playing our game for an extended period, and hoping to see a statue freeze at the edge of the screen, we can see the problem by loading the demonstration file that has been provided.

The demonstration version of the game has been configured so that a frozen statue can be seen at the right-hand edge of the screen, as shown in the previous screenshot. Test the demonstration game, but do not press any keys. The corner of a statue will be seen hanging in midair.

If the monkey is then made to run to the right of the screen, so that the camera moves, the statue will become active again, and fall as expected.

After completing the previous experiment, do not use the demo game to continue development. The demo game has been configured specifically for the purpose of demonstrating how inactive actors can freeze and still be visible onscreen, and several other elements of the game have been deleted or modified in order to simplify the demonstration.

It is recommended that, once the experiment is complete, the demo game be closed and deleted.

The origin of the actors

Stencyl determines which actors are to become inactive based on the location of the origin of the actor. By default, the origin of an actor will be at its exact center; it's the central point around which the actor will rotate. Stencyl tries to take into account the size of the actor, so it can ensure that the actor is completely offscreen before making it inactive. However, if an actor is not in its original orientation, it may still be seen to freeze.

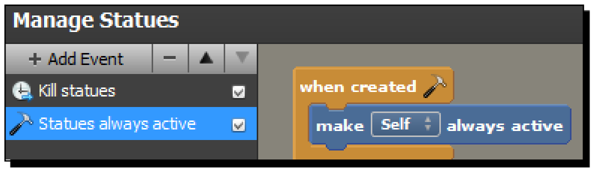
Our next task is to prevent the statue actors from becoming inactive.

Time for action – stopping the statues from becoming inactive

The game file that needs to be imported and loaded for this session is `5961_04_12.stencyl`.

Ensure that the `Manage Statues` actor behavior is on screen. We should be able to see the `Kill statues` event. Execute the following steps to stop the statue from becoming inactive:

1. Click on the **+ Add Event** button at the top of the left-hand panel.
2. Select **Basics**, then click on **When Creating**.
3. Rename the **Created** event to **Statues always active**.
4. Ensure that the **Actor** section button is selected in the instruction block palette.
5. Click on the **Properties** category button.
6. In the **Misc** subcategory, locate the blue **make Self always active** block, and drag it into the orange **when created** block, as shown in the following screenshot:



7. Test the game.

What just happened?

Although our game doesn't necessarily appear to behave any differently, the statues' physics properties are now still active, even when offscreen.

We have added a new **when created** event to the `Manage Statues` behavior, and we have renamed the new event to `Statues always active`. A **when created** event is triggered only once, when the actor is first created, and it's a very useful event into which one-off configuration blocks can be placed.

Because this behavior has been attached to the statue actor, the event will be carried out once for each statue actor as it is created, and then it will never be executed again for that actor.

Inside the **when created** event, we placed the `make Self always active` block, which ensures that the actor will never become inactive, even if it is not visible in the viewport. Now we can be sure that players of our game won't experience any unexpected frozen actors.

It's important to be aware that always active actors can have a serious impact on the performance of a game if they are not used carefully. Even when active actors are offscreen, they will still react to the game's physics requirements.

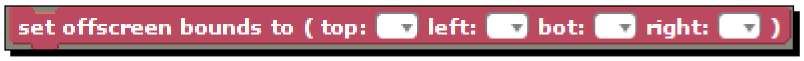
For this reason, care should be taken while deciding whether or not to make an actor always active. In our scene, we know that the statues are created every three seconds, and we know that the statues are killed after six seconds (a five second delay, then a one second fade-out). So, it is impossible for more than three statues to be active in our scene at any time.

Our earlier experiments have shown that we can have up to approximately 10 to 20 statues in a scene before the game's performance is adversely affected, and we must take this into consideration if we adjust the timings in the future. For example, if we add more playable levels to our game, which are more challenging, the number of active statues in the scene may increase.

There's more than one way

When developing games, there is almost always more than one way to achieve the same result and, very often, there are many ways! It's not necessarily a case of one method being better than the others, but sometimes different solutions can affect the gameplay or the performance of the game in different ways. For example, the game might lag, or there might be other consequences of using a particular solution.

Freezing actors is an example of a problem that has more than one solution. We chose to make the statue actors always active, which is a perfectly good solution for our game. However, there is at least one other equally good solution, which can be achieved using a Scene block called `set offscreen bounds to ...`, shown in the following screenshot:



This block can be found in the **Scene** section blocks under the **View** category.

Normally, the offscreen bounds are the same size as the screen. If the actor's origin moves outside the bounds of the screen, then the actor will become inactive. However, using a `set offscreen bounds to …` block allows us to specify bounds beyond the edges of the screen, in which our actors can remain active.

This could be perfect for our requirements because, if we set the offscreen bounds to an appropriate size, the statues would not freeze as they left the sides of the screen, but, if the monkey moved, leaving the statue far away and outside the edges of the screen where gameplay isn't affected, the statue would become inactive. This would free up resources without affecting the gameplay.

Have a go hero – using the offscreen bounds block

The game file that needs to be imported and loaded for this session is `5961_04_13.stencyl`.

Try replacing the `make Self always active` block with a `set offscreen bounds to …` block. Take care in deciding upon values for the arguments for this block.

> For detailed information relating to how to use a specific instruction block, right-click on the block and select **View Help** on the pop-up menu. This will open up a web browser at the relevant help page on the Stencyl website.

It will be helpful to know that the size of the statue is approximately 55 pixels wide and 80 pixels high.

Creating a countdown timer

Our game already has several challenges for players such as:

- Collecting fruit
- Avoiding dangerous animals
- Dodging falling statues

However, we're going to give our players another challenge, which doesn't involve other actors.

Time for action – creating a countdown timer

The game file that needs to be imported and loaded for this session is `5961_04_14.stencyl`.

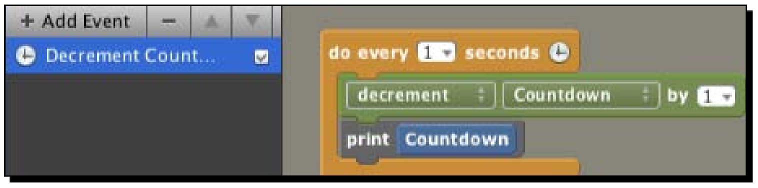
Ensure that the game has been saved, and close any open tabs, leaving only the **Dashboard** tab visible. Now execute the following instructions to create the countdown timer:

1. Create a new Scene Behavior called **Score Management**.
2. Click on the **Attributes** section button in the instruction block palette.
3. Click on the **Create an Attribute** button, situated immediately below the category buttons.
4. In the **Name** textbox, enter `Countdown`.
5. Select **Number** in the **Type** drop-down list.
6. Click on the **OK** button to close the dialog box.
7. In the left-hand panel, click on **+ Add Event**.
8. Select **Time**, then **Every N Seconds**.
9. Rename the **Every N Seconds** event to `Decrement Countdown`.
10. In the new **do every … seconds** event, enter the digit `1` into the empty textbox.
11. Click on the **Numbers & Text** section button at the top of the instruction block palette.
12. Drag the **increment number by 0** block into the **do every 1 seconds** event block.

Note that the **Number** option automatically changes to **Countdown**.

13. Click on the **increment** option item and change it to **decrement**.
14. Enter the digit `1` into the **decrement Countdown by …** textbox.
15. Click on the **Flow** section button at the top of the block palette.
16. Click on the **Debug** category button.
17. Locate the **print anything** block in the **Print to Console** subcategory, and drag it below the **decrement Countdown by 1** block.

18. Select the **Attributes** section button at the top of the block palette, and click on the **Getters** category button.
19. Drag the blue **Countdown** block into the empty **print ...** textbox, as shown in the following screenshot:



20. Click on the green **Attach to Scene** button situated above the block palette, and select the **Jungle** scene.
21. In the **Attributes** window that appears in the main panel, enter the digit `10` into the **Countdown** textbox.
22. On the main Stencyl menu, select the **Log Viewer** option from **View**.
23. Test the game.
24. When the game has compiled and is being displayed on the screen, examine the contents of the **Log Viewer** window. Watch what happens after 10 seconds.
25. Close the game.

What just happened?

We've created a `Score Management` behavior, and have displayed the value of the game's countdown in Stencyl's **Log Viewer** window. We should see that the countdown starts with a value of nine, and reduces by a value of one each second. After 10 seconds, the countdown reached zero, but then it continued to count backwards to -1, -2, -3, and it would continue to count backwards infinitely if we allowed it to (and if we had enough time).

The first step was to create an attribute to store the value of the countdown. An attribute is a changeable value that is used to store a specific piece of information, and, in this case, we need to store a number; therefore, we created a number attribute that represents the number of seconds that we want to count down from. Our new number attribute, called `Countdown`, can now be used anywhere within the behavior. It can be added to, subtracted from, and tested to see if it equals another value. We can manipulate attributes in almost every way imaginable!

Within the `Score Management` behavior, we added the `do every … seconds` event, which we renamed to `Decrement Countdown`. We then added the `increment number by …` block, but since we want our attribute to count down, we changed it from increment to decrement. Because our behavior only has one attribute, the name of that attribute automatically replaced the **Number** option. Finally, for this block, we specified the value we wanted to decrement by, which is `1`.

The `decrement Countdown by 1` block will be executed once every second, thus providing us with a second-by-second countdown. However, we haven't told the game what to do when the countdown reaches zero. So, it will just carry on counting down until we end the game.

When we attached the behavior to the jungle scene, we were presented with the behavior configuration screen. Because we created an attribute for this behavior, we were given the opportunity to provide a starting value for `Countdown`, and we entered the number `10`.

We don't yet have any way to display the value of the `Countdown` attribute within our game (that will be discussed in a later section). For now, we are temporarily using the `print …` instruction block to display the value in the Log Viewer.

The Log Viewer is a very useful tool—we can use a `print …` block to output information to the Log Viewer, so that we can see what is going on in our game at any time, without having to worry about displaying information on the game screen.

It's also a great tool for debugging because, if we have a problem and we're not sure why the problem is occurring, we can use a `print …` block to see the values of our attributes, or other information that Stencyl makes available to us, for example, the coordinates of actors.

During the testing of the game, we could see the value of `Countdown` in the Log Viewer, but the countdown started from nine, not 10. Why? When creating instructions, it's vital to fully understand the sequence of events that will occur. When our `Decrement Countdown` behavior starts, the process is as follows, remembering that the starting value of the `Countdown` attribute is `10`:

- Decrease the `Countdown` attribute by `1` (it's now equal to `9`)
- Print the value of `Countdown` to the Log Viewer

The first time the value of `Countdown` is printed to the Log Viewer, it has already been reduced by one. This may appear to be a trivial point to note, but such issues can have unexpected consequences in behaviors. So, is certainly worth being aware of what can go wrong. For example, before the game starts, we may have advised the player that they will have 10 seconds to complete the task. They may be annoyed when it appears that they only have 9 seconds. Game players can be very unforgiving.

In this instance, the problem can be resolved very easily, if we wish to do so—simply display the `Countdown` attribute before it is decreased. We can just drag the `print Countdown` block so that it is above the `decrement Countdown by 1` block. Try it!

Examining the debug blocks

We have used a `print …` debug block which, as we have discovered, is very useful for showing us values used in our game, without having to display them within the game itself.

There are several other debug blocks available to us. Take this opportunity to examine them, and consider how they might be useful to us as game developers.

To view the debug blocks, click on the **Flow** section button, then on the **Debug** category button in the block palette.

Remember that right-clicking any block will provide access to the **View Help** option.

Implementing decision making into our game

Now that we have the countdown working, we need to implement a solution that will kill the monkey when the counter reaches zero.

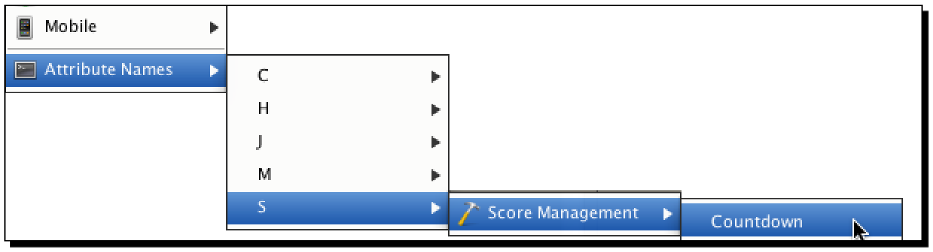
Time for action – listening for the countdown to reach zero

The game file that needs to be imported and loaded for this session is `5961_04_15.stencyl`.

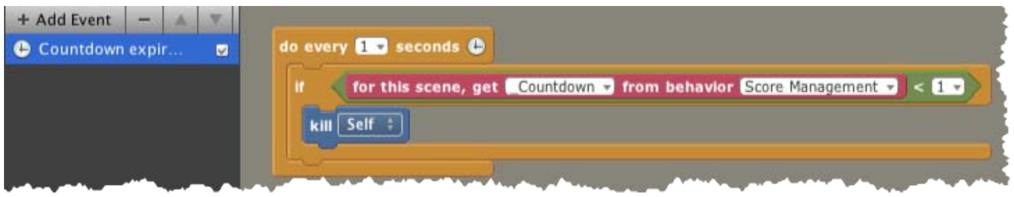
Rather than allowing the countdown to continue beyond zero and into negative numbers, in the following steps we're going to create a behavior with an event that will react when the player runs out of time:

1. Create a new Actor Behavior **called Health**.
2. Add a new **Time** event, **Every N Seconds**, and rename it to **Countdown expired**.
3. Change the **do every … seconds** textbox to `1` second.
4. Click on the **Flow** section button in the palette.

5. Ensure that the **Conditions** category button is selected.
6. Drag the orange **if** block into the **do every 1 seconds** event.
7. Click on the drop-down arrow in the **if** block, select **Comparison**, and, in the **Inequality** section, click on the **0 < 0** block.
8. In the right-hand textbox of the green **0 < 0** block, enter the number `1`.
9. In the block palette, click on the **Behavior** section button and the **Attributes** category button.
10. In the **For Scene** subcategory of the block palette, locate the red **for this scene**, **get text from behavior text** block and drag it into the left-hand textbox in the **0 < 0** block.
11. Click on the drop-down arrow in the left-hand textbox of the **for this scene, get … from behavior …** block and select **Attribute Names** in the drop-down menu.
12. In the pop-up menu, move the mouse over the letter **S**, then select **Score Management** and then **Countdown**, as shown in the following screenshot:



13. Locate the **kill Self** block in the palette, and drag it into the **if for this scene, get countdown from behavior Score Management** block, as shown in the following screenshot:


+ Add Event
Countdown expir...
do every 1 seconds
if for this scene, get Countdown from behavior Score Management < 1
kill Self

14. Attach the behavior to the monkey actor.
15. Test the game.
16. When the game is being displayed on the screen, examine the contents of the **Log Viewer** window. Watch what happens to the monkey after 10 seconds.
17. Close the game and close the **Log Viewer** window.

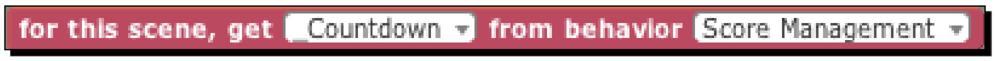
What just happened?

We've attached the newly created `Health` behavior to the monkey and we should now be familiar with the main event block in this behavior—it will do something every one second, because we created a `do every` … event block.

For the first time, we are introducing an `if …` block, which allows the instruction blocks to make a decision based on some information in our game. We want to know when our countdown timer is less than one, or in other words, when our player is out of time!

When we use an `if` … block, we must give it something to test and, in this case, we are comparing two numbers using the green less than block. The `<` symbol means less than and we're checking to see if something is less than the digit `1`, which we entered into the second textbox.

The something that we're checking is decided by the red `for this scene, get _ Countdown from behavior Score Management` block shown in the following screenshot, which, at first, might look a little complicated. So, let's break it down:


for this scene, get Countdown from behavior Score Management

This block will retrieve the value of an attribute that can be found in another behavior in the current scene. We used a very useful shortcut to populate the textboxes. We located the name of the attribute that we needed by using the drop-down menus. We could have typed the information into the boxes, and that works perfectly well—if we type all the information correctly! If we were to type any of the information incorrectly, then the instruction just won't work. So, why not just use the shortcut?

In plain English, our red `for this scene, get` … block could be phrased as follows:

In the current scene, retrieve the value of the Countdown attribute that is used in the Score Management behavior.

We created the `Score Management` behavior in the *Time for action – listening for the countdown to reach zero* section, and we know that that there is an attribute called `Countdown`, which is decreased every second.

If we now look at our new `Countdown expired` event as a whole, we can phrase it in English as:

Every one second, check if the value of Countdown is less than one and, if it is, kill Self.

Because we attached this behavior to the monkey actor, `Self` refers to the monkey.

We have a timer that will count down second-by-second, and we now have an event that checks when the value of the countdown is less than one and, if the countdown is less than one, the monkey is killed!

As we discovered earlier, a `kill` block is somewhat abrupt, and our monkey actor just disappears without warning. We'll be giving our game some polish later on but, remember, for the time being, we're just getting our gameplay mechanics working correctly.

There is a very important point to note when retrieving attribute values using the red `attribute` blocks. Note that the name of the attribute—in our case it is `Countdown`—is preceded with an underscore character so it appears as `_Countdown`. The underscore character is needed because these instruction blocks use Stencyl's **internal names** for the attributes, not the friendly names that we have given to them. For this reason, it is generally recommended, as we have done, to use the pop-up menus to populate the textboxes in these blocks.

What if? Otherwise...

We have used an `if` ... block to check for a specific condition, and to carry out actions if that condition is `True`**.** In our game, the actor is killed when the `Countdown` attribute has a value less than one.

However, what can we do if we need to respond when the condition is not `True`?

In the **Flow** section of the palette, in the **Conditions** category, there are two other instruction blocks alongside the **if ...** block in the **Conditionals** subcategory. Both of these alternative conditional blocks begin with the word **otherwise**.

Research what these blocks do, and consider how they might be useful when creating behaviors for a game.

Repositioning an actor during gameplay

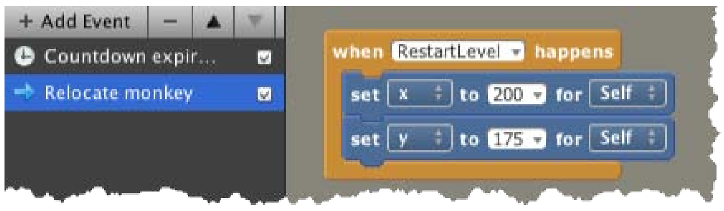
We're going to give our player more than one chance to complete the level. When the monkey dies after colliding with an enemy actor, we're going to move it back to the start of the level.

Time for action – creating an event to relocate the monkey

The game file that needs to be imported and loaded for this session is `5961_04_16.stencyl`.

Ensure that the `Health` behavior is being displayed. It currently contains a single event called `Countdown expired`. Now perform the following steps:

1. Click on **+ Add Event** | **Advanced** | **Custom Event**.
2. Rename the **Custom Event** to **Relocate monkey**.
3. In the new orange **when … happens** event block, enter the name `RestartLevel` (note the capitalization and that there no spaces).
4. In the block palette, under **Actor**, click on **Position**, locate the **set x to 0 for Self** block, and drag it into the **when RestartLevel happens** block.
5. Hold down the *Alt* key on the keyboard (*Option* on Mac OS X), and drag the new **set x to … for Self** block so that a copy of it appears below the first one.
6. In the top **set x to … for Self** block, enter the number `200` into the textbox.
7. In the lower **set x to 0 for self** block, click on the **x** option and change it to **y**, then enter `175` in the textbox, as shown in the following screenshot:



when RestartLevel happens
set x to 200 for Self
set y to 175 for Self

8. Save the game. It's not ready for testing yet.

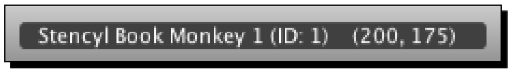
What just happened?

We've created a **Custom Event** called `Relocate monkey`. Custom events allow us to create a set of instructions that will be started by a unique trigger that we can specify.

The behaviors we have created so far rely on built-in events, such as when the value of an attribute is less than a specified number, or when actors belonging to the specified groups collide with each other. Our new custom event will listen out for the triggering of an event called `RestartLevel`, and we'll be creating an instruction to trigger that event in the next section, *Time for action – triggering the custom event*.

Once our custom event has been triggered, it will execute the instructions inside the `when RestartLevel happens` block. In this case, the instructions specify a new `x` and `y` position for the monkey, the actor will be placed right back where it started.

How did we know which numbers to specify for the `x` and `y` positions? If we open up the jungle scene in the Scene Designer, we can find out the start position of the monkey by clicking on the monkey actor in the scene and looking down to the lower left-hand corner of the **Stencyl** window, where we will see the coordinates of the currently selected actor, as shown in the following screenshot:


Stencyl Book Monkey 1 (ID: 1)   (200, 175)

The `x` and `y` coordinates depend on exactly where the monkey actor was placed in the jungle scene, so have a look at the coordinates, and adjust the numbers in the blue `set x to … for Self` and `set y to … for Self` blocks accordingly.

Triggering custom events in our behaviors

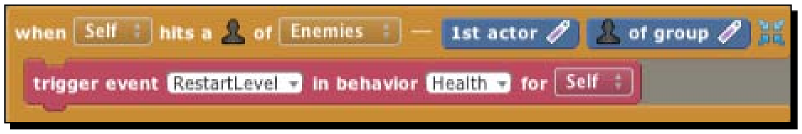
We've created a custom event that will react to the `RestartLevel` trigger, so our next task is to put together the instructions that will trigger the custom event when required.

Time for action – triggering a custom event

The game file that needs to be imported and loaded for this session is `5961_04_17.stencyl`. Execute the following instructions:

1. Open the **Manage Player Collisions** actor behavior.
2. In the left-hand panel, click on the **Collides with Enemies** event title.
3. Right-click on the **kill Self** block and select **Remove** in the pop-up menu.
4. In the block palette, click on the **Behavior** section button and ensure that the **Triggers** category is selected.
5. Drag the **trigger event text in behavior text for self** instruction block into the orange **when Self hits a …** block. Double-check that the correct instruction block has been selected (there are several blocks with similar names).
6. In the left-hand textbox of the **trigger event … in behavior … for Self** block, type `RestartLevel`—this must be typed so that it is identical to the name entered into the custom event block that we created previously, so check the capitalization.

7. In the right-hand textbox of the **trigger event RestartLevel in behavior … for Self** block, click on the drop-down arrow and select **Behavior Name | H | Health** from the drop-down menu, as shown in the following screenshot:


when Self hits a of Enemies — 1st actor of group
trigger event RestartLevel in behavior Health for Self

8. Test the game. Observe what happens when the monkey collides with an enemy actor, such as the croc.

What just happened?

We have created a trigger for our custom `RestartLevel` event! When the monkey collides with an enemy actor, it is no longer instantly killed. It is relocated to its starting position in the scene, so the player can have another attempt at completing the level.

The first step was to remove the **kill Self** block, as we no longer needed it. We then inserted the new **trigger event … in behavior … for Self** block, which needs two pieces of information—the name of the event to trigger, and the name of the behavior that holds the event.

It's absolutely vital to ensure that the event name (the name of the trigger) is typed exactly the same as it is in the custom event, otherwise the trigger just won't work.

Trigger names cannot have spaces, so it's often considered good practice to capitalize the first letter of each word in the trigger's name, so that they are easier to read.

Another useful tip is to copy the name of the trigger from the custom event block, and paste it into a **trigger …** block, so it can be guaranteed that the names match.

If we glance back to the block palette, where the various **trigger …** blocks are currently on view, we can see that that there are two **trigger …** blocks in the **For Actor** subcategory. We used the first block because we only have one custom event, called `RestartLevel`, and we know the name of the behavior that holds the event.

We could have been a little lazy and used the second block, which does not require us to specify the behavior name. There isn't really a problem with using that block; other than it having a trivial effect on the performance of the game, the player wouldn't notice anything slow down. However, it's a good idea to specify the behavior name when possible because, first, it helps us to understand what is happening right now and, secondly, if we need to review or modify the game in six months' time, it will remind us in which behavior to find the custom event.

Nevertheless, there is a very good reason for sometimes using the second **trigger …** block, which does not require the behavior name, and that is because we can create multiple custom events with the same trigger name. Imagine that we have four different behaviors that need to do something when an actor dies, we could create a custom event in each of those behaviors, and give each custom event the same trigger name, such as PlayerDied. We could trigger all those custom events with a single **trigger …** block, thus saving ourselves a lot of work.

Triggers and more triggers

In the section *Time for action – triggering the custom event*, we used the first **trigger …** block, which can be found in the **For Actor** subcategory of the **Behaviors** section of the palette, and we also discussed the use of the second block in the same subcategory.

Note that there are three other types of **trigger …** blocks: two in the **For Scene** subcategory, and one in the **For Both** subcategory.

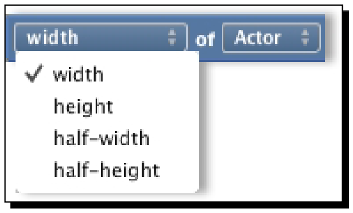
Consider how these additional **trigger …** blocks could be used, and particularly when it might be beneficial to use the **For Both** trigger, which can trigger actor and scene events.

Taking time to learn the available blocks

It's impossible to master the tools that are available to us, without knowing which tools are available.

In this chapter, we have created some very useful behaviors, but we have only used a small subset of the instruction blocks that are available to us.

Take some time to methodically work through the instruction block palette, and examine each block carefully, for example, does it have a drop-down option menu such as the block shown in the following screenshot?



If it does have a drop-down option menu, discover what the available options are, and consider how they might be useful in game design. Don't worry if the purpose of each block and the options available don't make sense immediately, some of the features are quite advanced, and are rarely used. However, knowing that these features are available will help to solve game design problems as they arise in the future.

For the moment, concentrate on the following sections and categories in the block palette:

- **Actor**: **Position** and **Properties**
- **Flow**: **Conditional**, **Looping**, and **Time**
- **Flow**: **Debug** and **Commenting**
- **Numbers & Text**: **Math**, **Trig / Exponents**, and **Text**

Just being aware of all the blocks in the previously mentioned Sections and Categories is a great start to being able to solve many of the problems that we face when designing and creating computer games.

If we're not sure what an instruction block is for, but we think it looks interesting, we can use the help system to learn more about the block. Just right-click on the block in the palette and select **View | Help**.

Learning from the provided behaviors

The behaviors that have been provided with Stencyl are an excellent source of information and inspiration. They can be edited and modified just like our own custom behaviors. In fact, the behaviors provided with Stencyl are just custom behaviors that have been created by someone else.

It's certainly a valuable exercise to examine the built-in behaviors that we have used in our game. The built-in behaviors that we have used are as follows:

- Cannot Exit Scene
- Camera Follow
- Jump and Run Movement

These behaviors can be located in the **Dashboard**, along with our behaviors that we have created—just double-click on the behavior's icon to view the instruction blocks that have been used to create the events.

Be aware that making changes to the built-in behaviors is very likely to cause problems in our game, so be careful to ensure they are not modified in any way.

Make a duplicate of the game, and experiment with it, rather than making experimental changes to our original game file. To make a duplicate of a game, save it first, to ensure that any recent changes have been saved to disk. Then, on the Stencyl menu, select **File** | **Save Game As**, and provide a name for the duplicate game. At this point, we can make changes without risk of affecting our original game.

Summary

We've reached the end of this chapter and, once again, we have made amazing progress. Over only three practical chapters, we have completed almost all of the gameplay-mechanics for our game.

We still have quite some way to go before we can say that our game is complete, there's a lot of polishing to do before we get there. However, we have, so far, covered many of the vital skills that we'll need to create numerous different types of games. It's important to remember that these skills are transferable to other genres of game. Many games will need the counters, decision making `if` … blocks, and triggers that we have learned about in this chapter!

The skills we have learned in this chapter include:

- Creating a custom behavior that carries out actions, and attaching it to an actor
- Using random numbers to introduce the element of surprise in our game
- Making a timer event in a behavior that is attached to a scene
- How and why actors should or shouldn't be always active
- Our first special effect—fading an actor until it disappears
- Positioning an actor at any desired location during gameplay
- Creating custom events and triggering them from other behaviors

We've also benefited from learning many game design skills and techniques, such as:

- The difference between screen size and scene size
- How to use the instruction block palette
- Basic optimization techniques, such as reducing the number of active actors onscreen at any one time
- Understanding that there is often more than one way to solve a problem, and how different solutions can be beneficial or disadvantageous to gameplay

We've reached the stage of development where the vast majority of gameplay is in place, and most of our future tasks will involve fine-tuning and polishing our game to make it more presentable to our players.

In *Chapter 5*, *Animation in Stencyl*, we're going to learn about some of the different types of animation that can be implemented within Stencyl, and we'll use these skills to add some interesting visual effects into our game.