

3

2D Animation

In this chapter, we will cover:

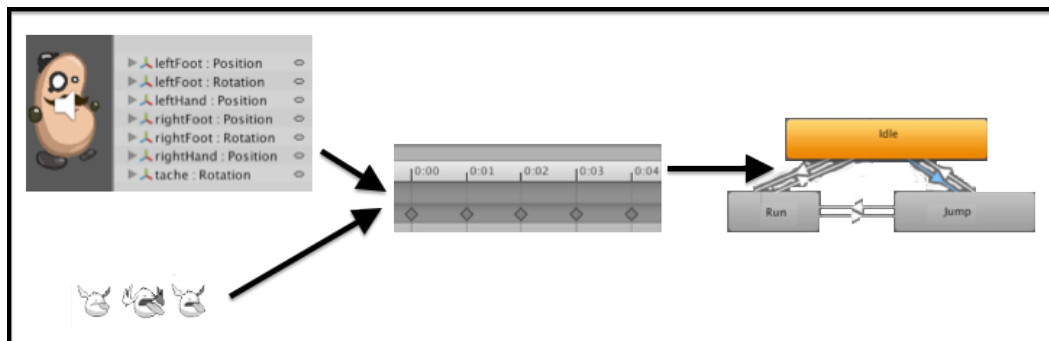
- Flipping a sprite horizontally
- Animating body parts for character movement events
- Creating a 3-frame animation clip to make a platform continually animate
- Making a platform start falling once stepped on using a Trigger to move animation from one state to another
- Creating animation clips from sprite sheet sequences

Introduction

Unity 5 builds on the introduction of powerful 2D features in the Mecanim animation system and the 2D physics system that were introduced in Unity 4.6 late 2014. In this chapter, we present a range of recipes to introduce the basics of 2D animation in Unity 5, and help you understand the relationships between the different animation elements in Unity.

The big picture

In Unity 2D animations can be created in several different ways – one way is to create many images, each slightly different, which frame-by-frame give the appearance of movement. A second way to create animations is by defining keyframe positions for individual parts of an object (for example, the arms, legs, feet, head, eyes, and so on), and getting Unity to calculate all the in-between positions when the game is running.



Insert image 1362OT_03_21.png

Both sources of animations become **Animation Clips** in the Animation panel. Each **Animation Clip** then becomes a **State** in the **Animator Controller State Machine**. We then define under what conditions a gameObject will **Transition** from one animation state (clip) to another.

Flipping a sprite horizontally

Perhaps the simplest 2D animation is a simple flip, from facing left to facing right, or facing up to facing down, and so on. In this recipe we'll add a cute bug sprite to the scene, and write a short script to flip its horizontal direction when the *Left* and *Right* arrow keys are pressed.



Insert image 1362OT_03_04.png

Getting ready

For this recipe, we have prepared the image you need in a folder named `sprites` in folder `1362_03_01`.

How to do it...

To flip an object horizontally with arrow key presses, follow these steps:

1. Create a new Unity 2D project.
2. Import the provided image `Enemy Bug`.

3. Drag an instance of the red **Enemy Bug** image from the **Project | Sprites** folder into the scene. Position this gameObject at (0,0,0) and scale to (2,2,2).
4. Add an instance of C# script class **BugFlip** as a component to your **Enemy Bug** gameObject:

```
using UnityEngine;
using System.Collections;

public class BugFlip : MonoBehaviour {
    private bool facingRight = true;

    void Update() {
        if (Input.GetKeyDown(KeyCode.LeftArrow) &&
            facingRight) Flip ();
        if (Input.GetKeyDown(KeyCode.RightArrow) &&
            !facingRight) Flip ();
    }

    void Flip () {
        // Switch the way the player is labelled as facing.
        facingRight = !facingRight;

        // Multiply the player's x local scale by -1.
        Vector3 theScale = transform.localScale;
        theScale.x *= -1;
        transform.localScale = theScale;
    }
}
```
5. When you run your scene, pressing the *Left* and *Right* arrow keys should make the bug face left or right correspondingly.

How it works...

The C# class defines a Boolean variable `facingRight`, which stores a true/false value corresponding to whether or not the bug is facing right or not. Since our bug sprite is initially facing right, then we set the initial value of `facingRight` to true to match this.

Method `Update()`, every frame, checks to see if the *Left* or *Right* arrow keys have been pressed. If the *Left* arrow key is pressed and the bug is facing right, then method `Flip()` is called, likewise if the *Right* arrow key is pressed and the bug is facing left (not facing right!), again method `Flip()` is called.

Method `Flip()` performs two actions, the first simply reverses the true/false value in variable `facingRight`. The second action changes the +/- sign of the X-value of the `localScale` property of the transform. Reversing the sign of the `localScale` results in

the 2D flip that we desire. Look inside the `PlayerControl` script for the **BeanMan** character in the next recipe – you’ll see exactly the same `Flip()` method being used.

Animating body parts for character movement events

In this recipe, we’ll learn to animate the hat of the Unity BeanMan character in response to a jump event.

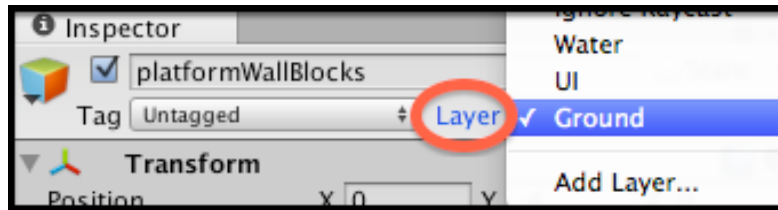
Getting ready

For this recipe, we have prepared the files you need in folder 1362_03_02.

How to do it...

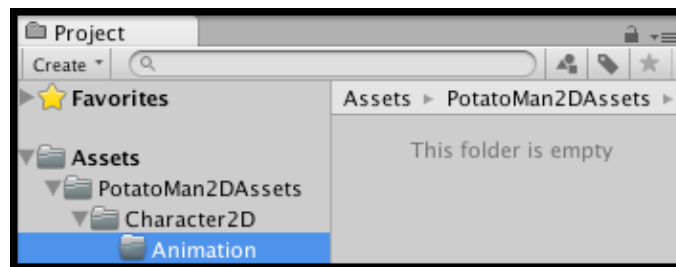
To animate body parts for character movement events, follow these steps:

1. Create a new Unity 2D project.
2. Import the provided package **BeanManAssets**, by choosing menu: **Assets | Import Package | Custom Package ...**, and then click the **Import** button to import all these assets into your **Project** panel.
3. Increase the size of the **Main Camera** to 10.
4. Let’s setup the 2D gravity setting for this project – we’ll use the same setting as from Unity’s 2D platform tutorial, a setting of $Y = -30$. Set 2D gravity to this value by choosing menu: **Edit | Project Settings | Physics 2D**, and then at the top change the Y value to -30.
5. Drag an instance of the bean-man **character2D** from the **Project | Prefabs** folder into the scene. Position this gameObject at (0,3,0).
6. Drag an instance of the sprite **platformWallBlocks** from the **Project | Sprites** folder into the scene. Position this gameObject at (0,-4,0).
7. Add a **Box Collider 2D** component to gameObject **platformWallBlocks** by choosing menu: **Add Component | Physics 2D | Box Collider 2D**.
8. We now have a stationary platform that the player can land upon, and walk left and right on. Create a new **Layer** named **Ground**, and assign gameObject **platformWallBlocks** to this new layer. Pressing the *Space* key when the character is on the platform will now make him jump.



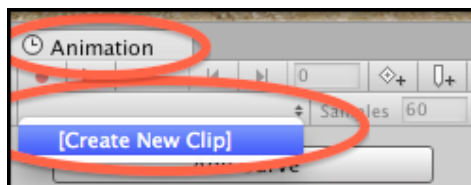
Insert image 1362OT_03_05.png

9. Currently the BeanMan character is animated (arms and legs moving) when we make him jump. Let's remove the Animation clips and Animator controller and create our own from scratch. Delete folders Clips and Controllers from **Project | Assets | PotatoMan2DAssets | Character2D | Animation**.



Insert image 1362OT_03_22.png

10. Let's create an Animation clip (and its associated Animator controller) for our hero character. In the Hierarchy panel select gameObject hero. Ensuring gameObject **character2D** is selected in the **Hierarchy**, open the **Animation** panel, and ensure it is in **Dope Sheet** view (this is the default).
11. Click the empty dropdown menu in the **Animation** panel (next to the greyed out word 'Samples'), and choose menu item **[Create New Clip]**.

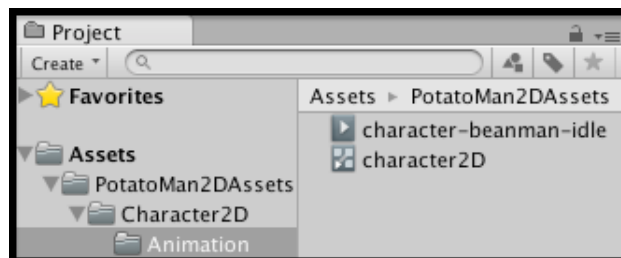


Insert image 1362OT_03_06.png

12. Save the new clip in the **Character2D | Animation** folder, naming it '**character-beanman-idle**'. You've now created an Animation clip for the 'idle' character state (wish is not animated).

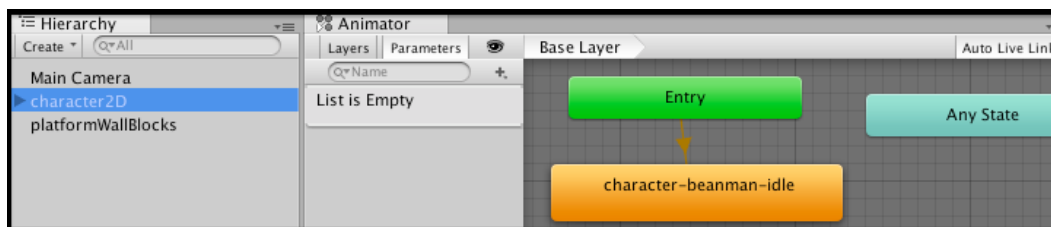
Note. Your final game may end up with tens, or even hundreds, of animation clips. Make things easy to search by prefixing the names of clips with object type, name, and then description of the animation clip.

13. Looking at the **Character2D | Animation** folder in the Project panel you should now see both the Animation clip you have just created (**character-beanman-idle**) and also a new Animator controller, which has defaulted to the name of your gameObject **character2D**.



Insert image 1362OT_03_23.png

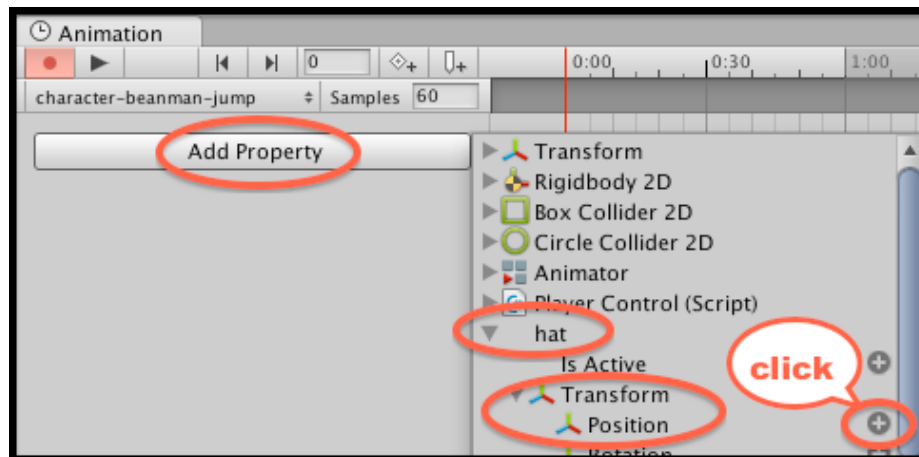
14. Ensuring gameObject **character2D** is selected in the **Hierarchy**, open the **Animator** panel and you'll see the State Machine for controlling the animation of our character. Since we only have one Animation clip (**character-beanman-idle**) then upon entry the State Machine immediately enters this state.



Insert image 1362OT_03_24.png

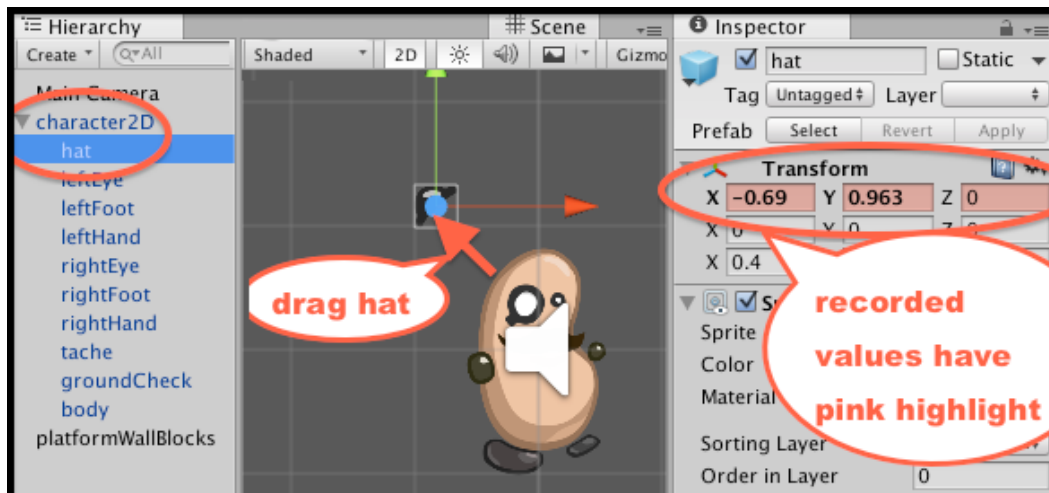
15. Run your scene – since the character is always in the 'idle' state, we see no animation yet when we make it jump.

16. Now we'll create a 'jump' Animation clip which animates the hat. Click the empty dropdown menu in the **Animation** panel (next to the greyed out word 'Samples'), and create a new clip in your **Animation** folder, naming it **character-beanman-jump**.
17. Click button **Add Property**, and chose **Transform|Position** of the **hat** child object, by clicking its '+' plus-sign button. We are now recording changes to the (X,Y,Z) position of gameObject **hat** in this animation clip.



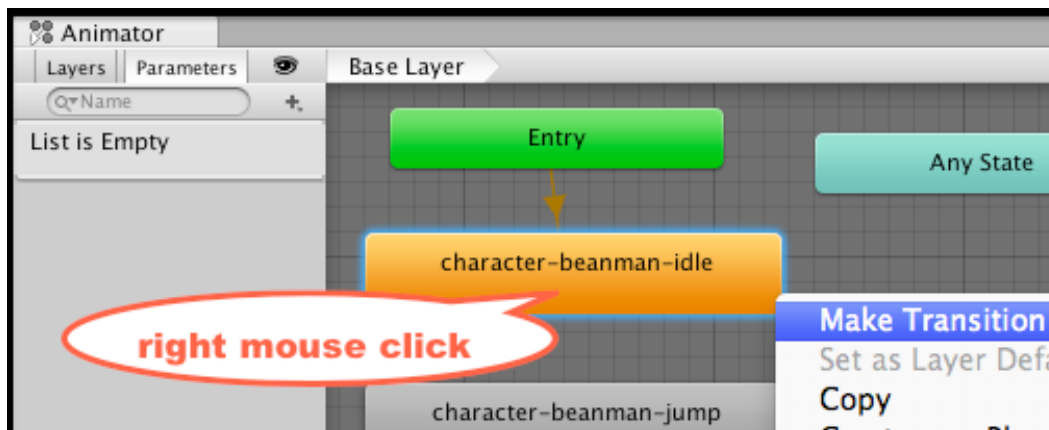
Insert image 1362OT_03_25.png

18. You should now see 2 'keyframes' at 0.0 and at 1.0. These are indicated by diamonds in the **Timeline** area in the right-hand-section of the **Animation** panel.
19. Click to select the first keyframe (at time 0.0). Now in the Scene panel move the hat up and left a little, away from the head. You should see that all three X, Y, Z values have a red background in the **Inspector** – this is to inform you that the values of the **Transform** component are being recorded in the animation clip.



Insert image 1362OT_03_26.png

20. Since 1 second is perhaps too long for our jump animation, drag the second keyframe diamond to the left to a time of 0.5.
21. We now need to define when the character should Transition from the 'idle' state to the 'jump' state. In the **Animator** panel select state **character-beanman-idle**, and create a transition to the state **character-beanman-jump** by right-mouse-clicking and choosing menu Make Transition, then drag the transition arrow to state **character-beanman-jump**.



Insert image 1362OT_03_27.png

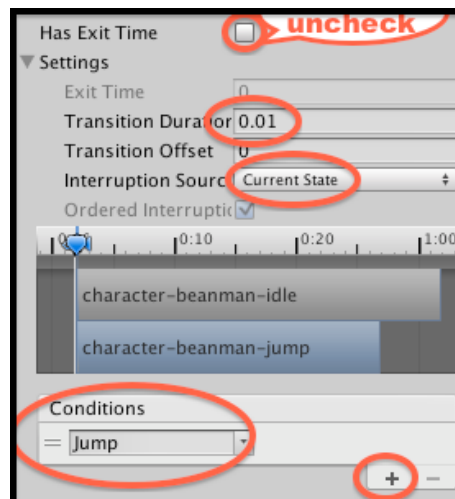
22. Now let's add a Trigger parameter named 'Jump', by clicking on the add parameter plus-sign "+" button at the top-left of the **Animator** panel, choosing **Trigger**, and typing the name **Jump**.



Insert image 1362OT_03_28.png

23. We can now define the properties for when our character should **Transition** from idle to jump. Click the Transition arrow to select it, and set the following 4 properties in the **Inspector** panel:

- **Has Exit Time**: uncheck
- **Transition Duration**: 0.01
- **Interruption State**: Current State
- **Conditions**: Add **Jump** (click plus-sign "+" button at bottom)



Insert image 1362OT_03_29.png

24. Save and run your scene. Once the character has landed on the platform and you press the **SPACE** key to jump, you'll now see the character's hat jump away from his head, and slowly move back. Since we haven't added any transition to ever leave the Jump state, this Animation clip will loop, so the hat keeps on moving even when the jump is completed.
25. In the Animator panel select state **character-beanman-jump** and add a new Transition back to state **character-beanman-idle**. Select this Transition arrow and in the Inspector panel sets its properties as follows:
 - **Has Exit Time:** (leave as checked)
 - **Exit time:** 0.5 (this needs to be the same time value as the second keyframe of our Jump animation clip)
 - **Transition Duration:** 0.01
 - **Interruption State:** Current State
26. Save and run your scene. Now when you jump the hat should animate once, after which the character immediately returns to its Idle state.

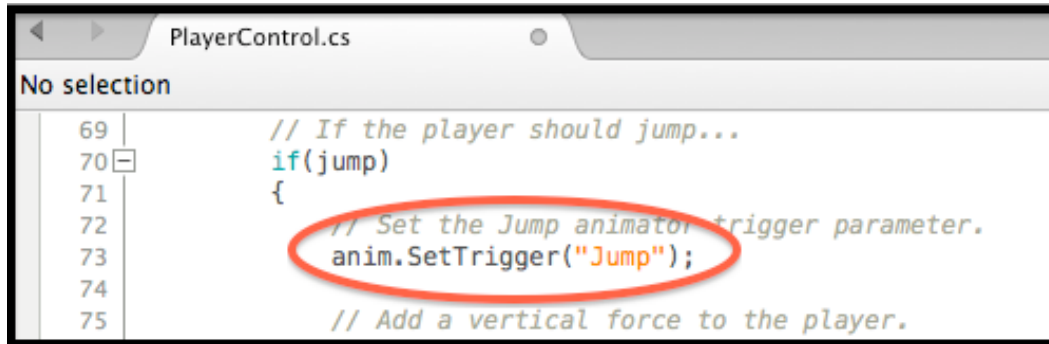
How it works...

You have added an Animation controller State Machine to gameObject **character2D**. The 2 Animation clips you created (idle and jump) appear as States in the Animator panel. You created a Transition from Idle to Jump when the 'Jump' Trigger parameter is received by the State Machine. You created a second Transition, which transitions back to the Idle state after waiting 0.5 seconds (the same duration between the 2 keyframes in our Jump Animation clip).

Note that the key to everything working for the bean-man character is that when we make the character jump with the **SPACE** key, then code in the **PlayerControl** C# scripted component of gameObject **character2D**, as well as making the sprite move upwards on screen, also sends a **SetTrigger(...)** message to the Animator controller component, for the **Trigger** named **Jump**.

Note. The difference between a **Boolean** Parameter and a **Trigger** is that a **Trigger** is temporality set to **True** and once the **SetTrigger(...)** event has been 'consumed' by a state transition it automatically returns to being **False**. So Triggers are useful for actions we wish to do once and then revert to a previous state. A **Boolean** Parameter is a variable, which can have its value set to true/or **False** at different times during the game, and so different Transitions can be created to fire depending on the value of the variable at any time. Note that **Boolean** parameters have to have their values explicitly set back to **False** with a **SetBool(...)**.

The screenshot highlights the line of code that sends the `SetTrigger(...)` message.



```
69 // If the player should jump...
70 if(jump)
71 {
72     // Set the Jump animator trigger parameter.
73     anim.SetTrigger("Jump");
74 }
75 // Add a vertical force to the player.
```

Insert image 1362OT_03_30.png

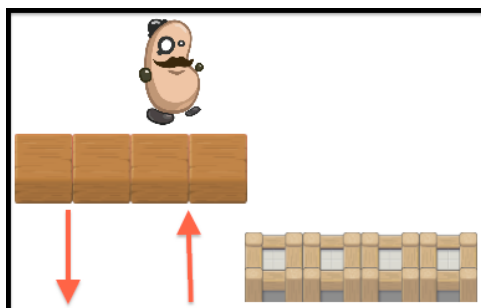
State Machines for animations of a range of motions (running / walking / jumping / falling / dying etc.) will have more states and transitions. The Unity provided bean-man character has a more complex **State Machine**, and more complex animations (of hands and feet, and eyes and hat etc. for each **Animation** clip), which you may find useful to explore.

Learn more about the Animation view on the Unity Manual web pages:

<http://docs.unity3d.com/Manual/AnimationEditorGuide.html>.

Creating a 3-frame animation clip to make a platform continually animate up and down

In this recipe, we'll make a wooden-looking platform continually animate, moving upwards and downwards. This can be achieved with a single, 3-frame, animation clip (starting at top, position at bottom, top position again).



Insert image 1362OT_03_02.png

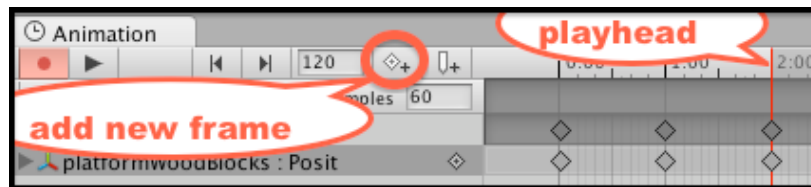
Getting ready

This recipe builds on the previous one, so make a copy of that project, and work on the copy for this recipe.

How to do it...

To create a continually moving animated platform, follow these steps:

1. Drag an instance of the sprite **platformWoodBlocks** from the **Project | Sprites** folder into the scene. Position this gameObject at (-4,-5,0), so that these wood blocks are neatly to left, and slightly below, the wall blocks platform.
2. Add a Box Collider 2D component to gameObject **platformWoodBlocks** so that the player's character can stand on this platform too. Choose menu: **Add Component | Physics 2D | Box Collider 2D**.
3. Create a new folder named **Animations**, in which to store the animation clip and controller we'll create next.
4. Ensuring gameObject **platformWoodBlocks** is still selected in the **Hierarchy**, open an **Animation** panel, and ensure it is in **Dope Sheet** view (this is the default).
5. Click the empty dropdown menu in the **Animation** panel (next to the greyed out word 'Samples'), and choose menu item **[Create New Clip]**.
6. Save the new clip in your **Animations** folder, naming it '**platform-wood-moving-up-down**'.
7. Click button **Add Curve**, and chose **Transform** and the click the '+' plus-sign by **Position**. We are now recording changes to the (X,Y,Z) position of gameObject **platformWoodBlocks** in this animation clip.
8. You should now see 2 'keyframes' at 0.0 and at 1.0. These are indicated by diamonds in the **Timeline** area in the right-hand-section of the **Animation** panel.
9. We need 3 keyframes, with the new one at 2:00 seconds. Click at 2:00 in the Timeline along the top of the **Animation** panel, so that the red line for the current playhead time is at time 2:00. Then click diamond+ button to create a new keyframe at the current playhead time.



Insert image 1362OT_03_08.png

10. The first and third keyframes are fine – they record the current height of the wood platform at $Y = -5$. We need to make the middle keyframe record the height of the platform at the top of its motion, and Unity in-betweening will do all the rest of the animation work for us. Select the middle keyframe (at time 1:00), by clicking on either diamond at time 1:00 (they should both turn blue, and the red playhead vertical line should move to 1:00, to indicate the middle keyframe is being edited).
11. Now in the **Inspector** change the Y position of the platform to 0. You should see that all three X, Y, Z values have a red background in the **Inspector** – this is to inform you that the values of the **Transform** component are being recorded in the animation clip.
12. Save and run your scene. The wooden platform should now be animating continuously, moving smoothly up and down the positions we setup.

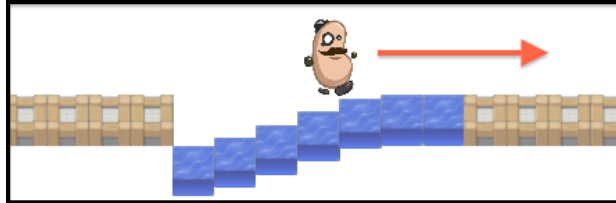
How it works...

You have added an animation to gameObject **platformWoodBlocks**. This animation contains three keyframes. A keyframe represents the values of properties of the object at a point in time. The first keyframe stores a Y-value of -4, the second keyframe a Y-value of 0, and the final keyframe -4 again. Unity calculates all the in-between values for us, and the result is a smooth animation of the Y-position of the platform.

Making a platform start falling once stepped on using a Trigger to move animation from one state to another

In many cases we don't wish an animation to begin until some condition has been met, or some event occurred. In these cases a good way to organize the Animator Controller is to have two animation states (clips) and a Trigger on the Transition between the clips. We use code to detect when we wish the animation to start playing, and at that time we send the Trigger message to the Animation Controller, causing the transition to start.

In this recipe we'll create a water platform block in our 2D platform game; such blocks will begin to slowly fall down the screen as soon as they have been stepped on, and so the player must keep on moving otherwise they'll fall down the screen with the blocks too!



Insert image 1362OT_03_01.png

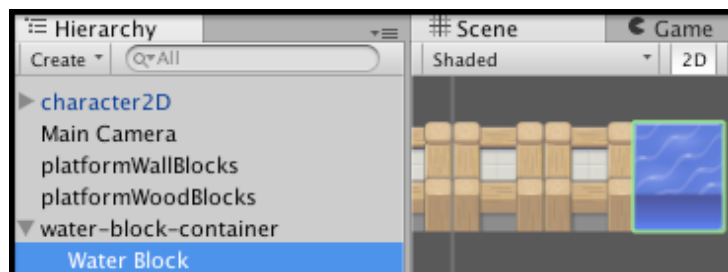
Getting ready

This recipe builds on the previous one, so make a copy of that project, and work on the copy for this recipe.

How to do it...

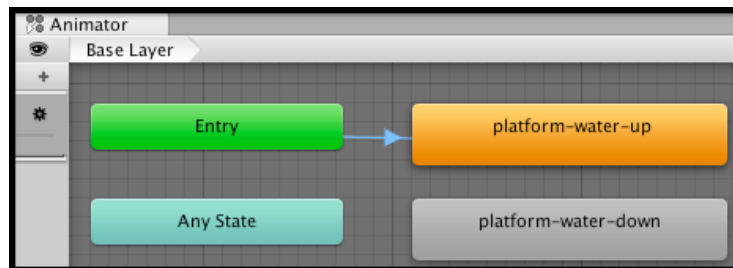
To construct an animation that only plays once a Trigger has been received, follow these steps:

1. In the **Hierarchy** create an **Empty** gameObject named **water-block-container**, positioned at (2.5, -4, 0). This empty gameObject will allow us to make duplicates of animated Water Blocks that will animate relative to their parent gameObject position.
2. Drag an instance of the sprite **Water Block** from the **Project | Sprites** folder into the scene and child it to gameObject **water-block-container**. Ensure the position of your new child gameObject **Water Block** is (0,0,0), so that it appears neatly to right of the wall blocks platform.



Insert image 1362OT_03_09.png

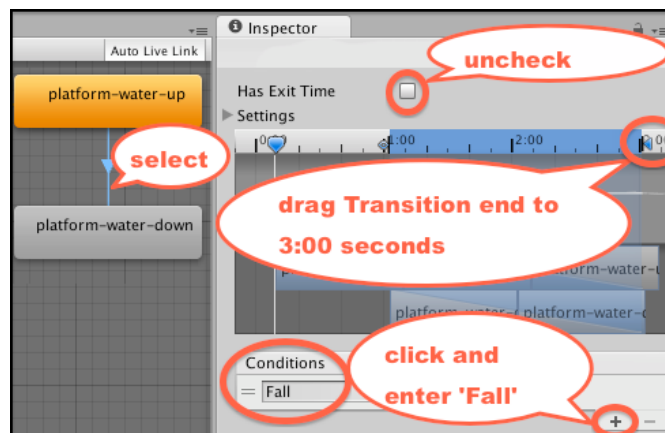
3. Add a **Box Collider 2D** component to child gameObject **Water Block**, and set the layer of this gameObject to **Ground**, so that the player's character can stand and jump on this water block platform.
4. Ensuring child gameObject **Water Block** is selected in the **Hierarchy**, open an **Animation** panel, then create a new clip named **platform-water-up**. saving it in your **Animations** folder.
5. Click button **Add Curve**, and chose **Transform** and **Position**.
6. Delete the second keyframe at time 1:00. You have now completed the creation of the water block up animation clip.
7. Create a second Animation clip, named **platform-water-down**. Again, click button **Add Curve**, and chose **Transform** and **Position**, and delete the second keyframe at time 1:00.
8. With the first keyframe at time 0:00 selected, set the Y-value of the gameObjects Transform Position to -5. You have now completed the creation of the water block down animation clip, so you can click the red record button to stop recording.
9. You may have noticed that as well as the up/down **Animation Clips** that you created, another file was created in your **Animations** folder, an **Animator Controller** named **Water Block**. Select this file and open the **Animator** panel, to see and edit the State Machine diagram.



Insert image 1362OT_03_10.png

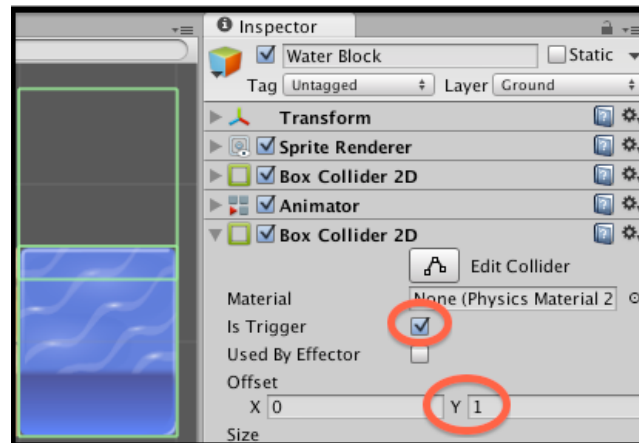
10. Currently, although we created 2 animation clips (states), only the **Up** state is ever active. This is because when the scene begins (Entry) the object will immediately go in state **platform-water-up**, but since there are no transition arrows from this state to **platform-water-down**, then at present the **Water Block** gameObject will always be in its **Up** state.
11. Ensure state **platform-water-up** is selected (it will have a blue border around it), and create a Transition (arrow) to state **platform-water-down**, by choosing **Make Transition** from the **mouse-right-click** menu.

12. If you run the scene now, the default **Transition** settings are that after 0.9 seconds the **Water Blocks** will transition into their **Down** state. We don't want this – we only want them to animate downwards after the player has walked onto them. So create a **Trigger** named **Fall**, by choosing the **Parameters** tab in the **Animator** panel, clicking the plus '+' button and selecting **Trigger**, and then selecting **Fall**.
13. Do the following to create our Trigger:
- in the **Animator** panel select the **Transition**, then
 - in the **Inspector** panel uncheck the **Has Exit Time** option,
 - in the **Inspector** panel drag the Transition end time to 2:00 seconds (so the Water Block slowly Transitions to its Down state over a period of 2 seconds)
 - in the **Inspector** panel click the plus '+' button to add a **Condition**, which should automatically suggest the only possible condition parameter, which is our **Trigger Fall**.



Insert image 1362OT_03_12.png

14. We now need to add a collider trigger just above the Water Block, and add C# script behavior to send the **Animator Controller Trigger** when the player enters the collider. Ensuring child gameObject **Water Block** is selected, add a (second) 2D Box Collider, with a Y-Offset of 1, and tick its **Is Trigger** checkbox.



Insert image 1362OT_03_13.png

15. Add an instance of C# script class `waterBlock` as a component to your **Water Block** child gameObject:

```
using UnityEngine;
using System.Collections;

public class waterBlock : MonoBehaviour {
    private Animator animatorController;

    void Start(){
        animatorController = GetComponent<Animator>();
    }

    void OnTriggerEnter2D(Collider2D hit){
        if(hit.CompareTag("Player")){
            animatorController.SetTrigger("Fall");
        }
    }
}
```

16. Make 6 more copies of gameObject **water-block-container**, with X positions increasing by 1 each time, that is, 3.5, 4.5, 5.5, and so on.
17. Run the scene, and as the player's character runs across each water block they will start falling down, so he had better keep running!

How it works...

You created a two-state **Animator Controller** state machine. Each state was an **Animation Clip**. You created a **Transition** from the **Water Block** Up state to its Down state that will take place when the Animator Controller received a Fall Trigger message. You created a **Box Collider 2D** with a **Trigger**, so that scripted component WaterBlock could be detected when the player (tagged **Player**) enters its collider, and at that point send the **Fall** Trigger message to make the **Water Block** gameObject start gently Transitioning into its Down state further down the screen.

Learn more about the Animator Controllers on the Unity Manual web pages:
<http://docs.unity3d.com/Manual/class-AnimatorController.html>.

Creating animation clips from sprite sheet sequences

The traditional method of animation involved hand drawing many images, each slightly different, which displayed quickly frame-by-frame to give the appearance of movement. For computer game animation, the term Sprite Sheet is given to the image file that contains one or more sequences of sprite frames. Unity provides tools to breakup individual sprite images in large sprite sheet files, so that individual frames, or sub-sequences of frames can be used to create Animation Clips that can become States in Animator Controller State Machines. In this recipe, we'll import and break up an open source monster sprite sheet into three animation clips for Idle, Attack, and Death.



Insert image 1362OT_03_17.png

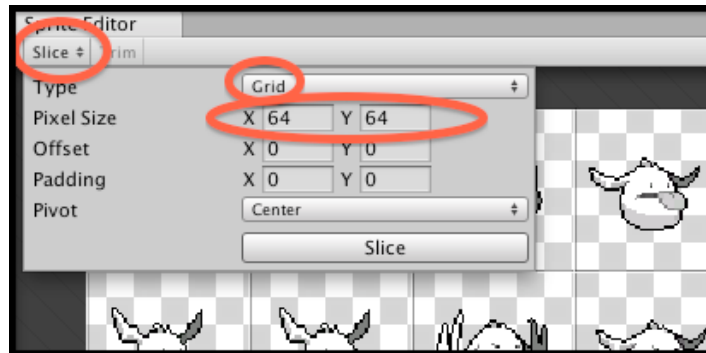
Getting ready

For all the recipes in this chapter, we have prepared the sprite images you need in folder 1362_03_05.

How to do it...

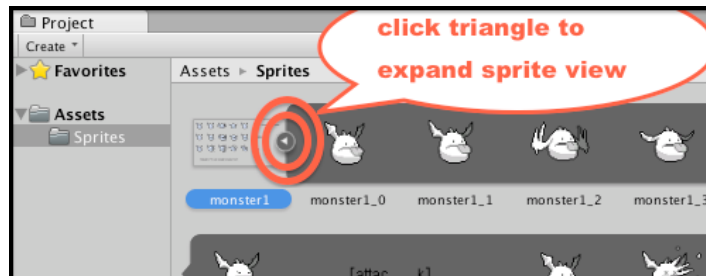
To create an animation from a sprite sheet of frame-by-frame animation images, follow these steps:

1. Create a new Unity 2D project.
2. Import the provided image `monster1`.
3. With image `monster1` selected in the **Project** panel, change its **Sprite** mode to **Multiple** in the **Inspector**, then open the **Sprite Editor** panel by clicking button **Sprite Editor**.
4. In the **Sprite Editor** open the **Slice** dropdown dialog, set the **Type** to **Grid**, set the grid **Pixel Size** to **64x64**, and then click the **Slice** button. Finally, click the **Apply** button in the bar at the top right of the **Sprite Editor** panel).



Insert image 1362OT_03_18.png

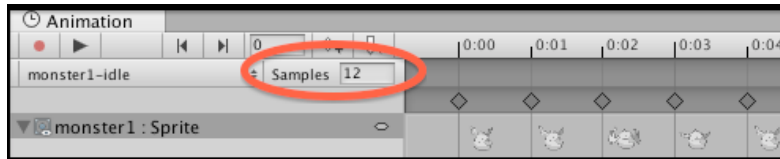
5. In the **Project** panel you can now click the expand triangle button center-right on the sprite, and you'll see all the different child frames for this sprite.



Insert image 1362OT_03_19.png

6. Create a folder named `Animations`.
7. In your new folder, create an **Animator Controller** named `monster-animator`.
8. In the scene create a new **Empty** gameObject named `monster1` (at position 0,0,0), and drag your `monster-animator` into this gameObject.

9. With gameObject **monster1** selected in the **Hierarchy**, open up the **Animation** panel, and create a new **Animation Clip** named **Idle**.
10. Select image **monster1** in the **Project** panel (in its expanded view), and select and drag the first 5 frames (frames 0-4) into the **Animation** panel. Change the sample rate to 12 (since this animation was created to run at 12-frames per second).



Insert image 1362OT_03_20.png

11. If you look at the State Chart for **monster-animator**, you'll see it has a default state (clip) named **monster-idle**.
12. When you run your scene you should now see the **monster1** gameObject animating in its **monster-idle** state. You may wish to make the Main Camera size a bit smaller, since these are quite small sprites.

How it works...

Unity's Sprite Editor knows about sprite sheets, and once the correct grid size has been entered it treats the items in each grid square inside the sprite sheet image as an individual image, or frame, of the animation. You selected sub-sequences of sprite animation frames and added them into several **Animation Clips**. You had added an **Animation Controller** to your gameObject, and so each **Animation Clip** appears as a state in the **Animation Controller State Machine**.

You can now repeat the process, creating an **Animation Clip** **monster-attack** with frames 8-12, and a third clip **monster-death** with frames 15-21. You would then create Triggers and **Transitions** to make the monster gameObject transition into the appropriate **states** as the game is played.

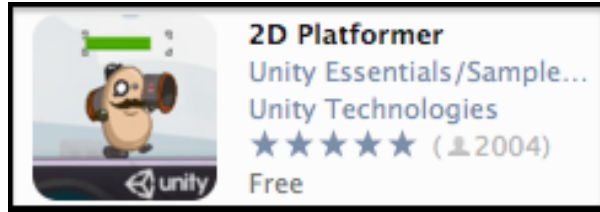
Learn more about the Unity Sprite Editor from the Unity video tutorials:
<https://unity3d.com/learn/tutorials/modules/beginner/2d/sprite-editor>.

Conclusion

In this chapter, we have introduced recipes demonstrating the animation system for 2D game elements. The bean-man 2D character is from the Unity 2D Platformer, which you

can download yourself from the Unity asset store. That project is a good place to see lots more examples of 2D game and animation techniques:

www.assetstore.unity3d.com/en/#!/content/11228.



Insert image 1362OT_03_03.png

Here are some links for useful resources and sources of information to explore these topics further:

- Unity 2D Platformer (where the BeanMan character came from):
<https://www.assetstore.unity3d.com/en/#!/content/11228>
- The platform sprites are from Daniel Cook's **Planet Cute** game resources:
www.lostgarden.com/2007/05/dancs-miraculously-flexible-game.html
- Creating a basic 2D platformer game:
unity3d.com/learn/tutorials/modules/beginner/live-training-archive/creating-a-basic-platformer-game
- Hat Catch 2D game tutorial:
unity3d.com/learn/tutorials/modules/beginner/live-training-archive/2d-catch-game-pt1
- Unity games from a 2D perspective video:
unity3d.com/learn/tutorials/modules/beginner/live-training-archive/introduction-to-unity-via-2d
- A fantastic set of modular 2D characters with a free Creative Commons licence from 'Kenny'. These assets would be perfect for animating body parts in a similar way to the bean-man example in this chapter and in the Unity 2D platformer demo:
<http://kenney.nl/assets/modular-characters>