

12

Controlling Animations with the Legacy Animation System

In this chapter, we will cover the following:

- ▶ Animating with the Animation view
- ▶ Dividing animation into clips
- ▶ Controlling playback and speed
- ▶ Setting up an animated Mixamo character
- ▶ Adding rigid props to animated characters
- ▶ Making an animated character throw an object
- ▶ Applying ragdoll physics to a character
- ▶ Rotating the character's torso to aim
- ▶ Blending, mixing, and cross-fading animation states
- ▶ Using animation events to play audio clips

Introduction

For quick and simple animation work, sometimes the pre-Mecanim **Legacy animation system** can be a good solution. In this chapter, we will learn a diverse set of techniques which will enable you to optimize the use of your game's animated models using the Legacy Animation System available in Unity 4.x.

Recipes dealing with externally-animated characters will make use of Mixamo's suite of characters, motion packs, and scripts. **Mixamo** is a complete solution for character rigging and animation, and it can even provide you with rigged 3D characters at low or no cost. You can find out more about it at Unity's Asset Store (u3d.as/content/mixamo/mixamo-animation-store/1At) or their website at www.mixamo.com.

Animating with the Animation view

Whether you don't have access to a 3D software package or just don't want to leave the Unity editor, you can create simple animation clips quickly and effectively through the **Animation** view. In this recipe, we will illustrate this process by animating a carousel.

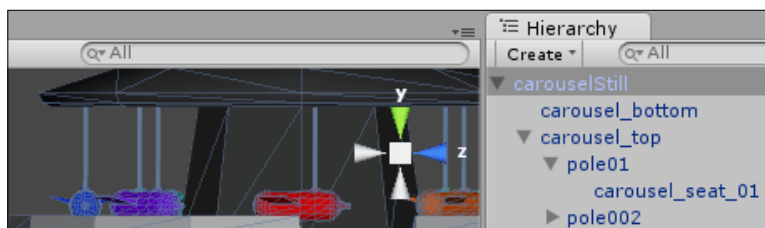
Getting ready

For this recipe, we have included a model named `carouselStill.FBX`, available in the `0423_12_01` folder.

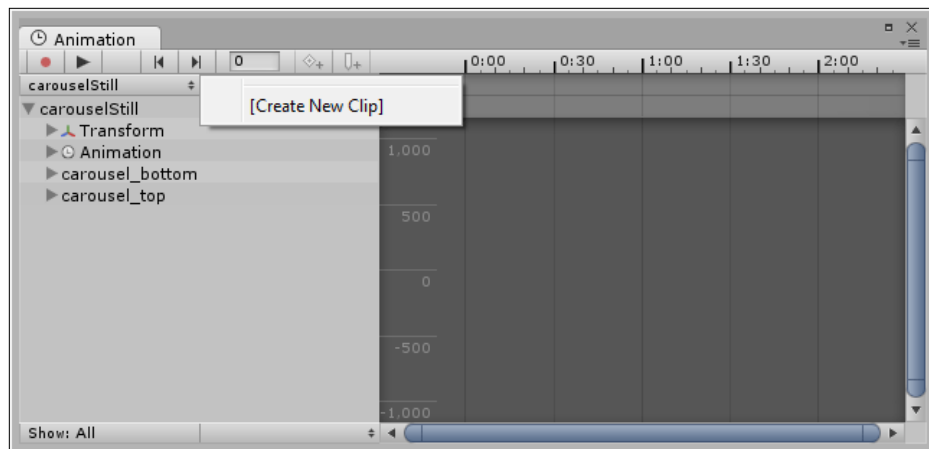
How to do it...

To create animation clips with the **Animation** view, perform the following steps:

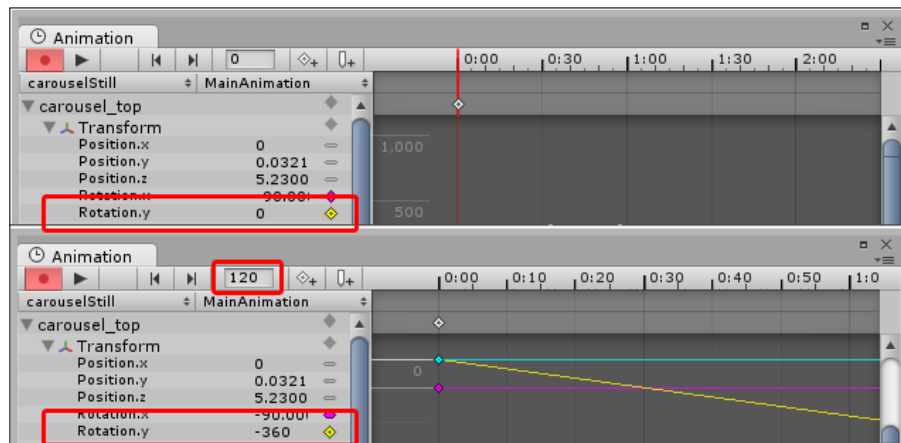
1. Import the FBX model `carouselStill` into your project and add it to your scene.
2. In the **Hierarchy** view, observe how the `carouselStill` is structured and how some elements are parented by others. The **Inspector** window should indicate that there is an **Animator** component attached to the game object. Remove that component and add the **Animation** component (**Component | Miscellaneous | Animation**).



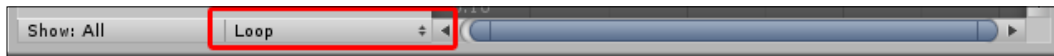
3. With the `carouselStill` game object selected, open the **Animation** view (**Window | Animation**).
4. Click on the button on the right-hand side to the one with the object's name to create a new clip. Save it as `MainAnimation.anim` inside the **Assets** folder.



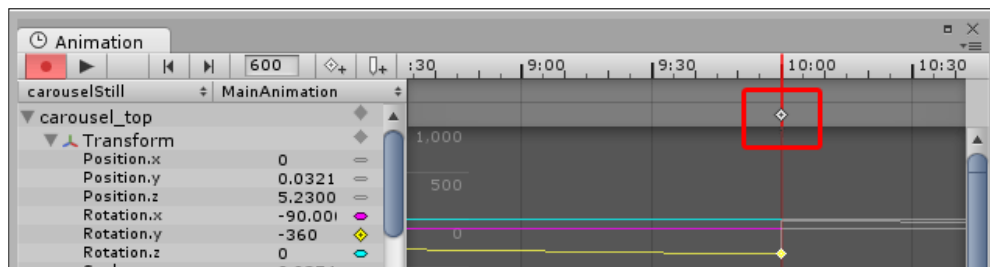
5. In the **Hierarchy** view, select the child named **carousel_top**.
6. Now, in the **Animation** view, click on the Timeline and drag the time indicator (represented by a red vertical line) to **0:00**. Also, notice how the **Record** button is switched on.
7. In the **Rotation.y** field, type in 0 and press the *Enter* key. That should create a keyframe.
8. Drag the time indicator to **2:00** (or type 120 in the **Counter** field located to the right-hand side of the recording/playback buttons, and press the *Enter* key). Then type -360 into the **Rotation.y** field and press the *Enter* key. A new keyframe should be created.



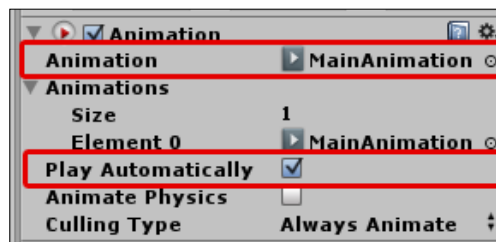
9. In the bottom part of the **Animation** view, click on the appropriate button to change the **Wrap Mode** to **Loop**.



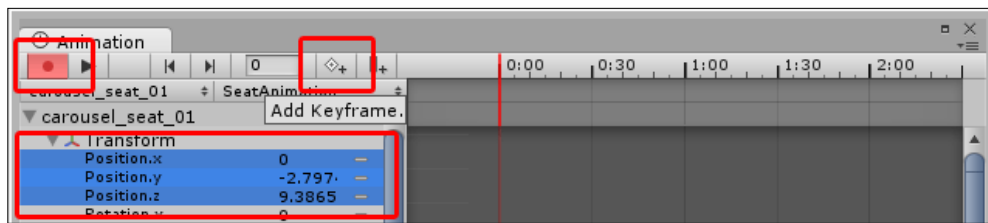
10. Click on the **Play** button of the **Animation** view to see your clip running. As it feels too fast, let's drag our second keyframe from **2:00** to **10:00** seconds.



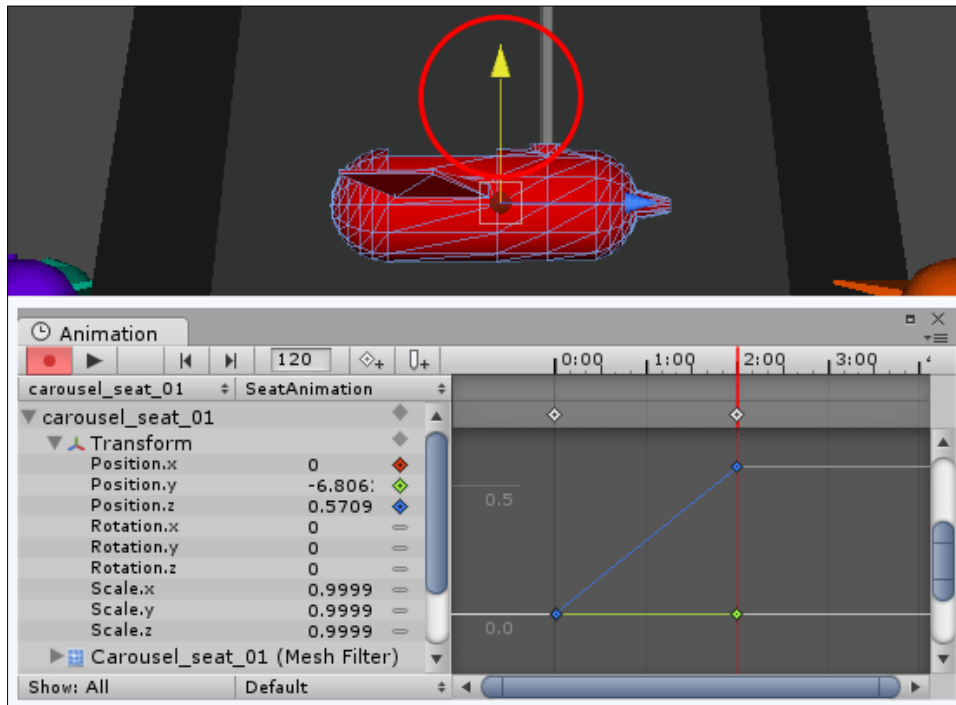
11. Select the **carouselStill** game object and, in the **Inspector** view, check out its **Animation** component. Make sure the option **Play Automatically** is selected and then choose **MainAnimation** as its default clip.



12. Now, we will create a separate animation clip for one of the carousel seats. In the **Hierarchy** window, select the object named **carousel_seat_01**. Then, attach an animation component to it (**Component** | **Miscellaneous** | **Animation**).
13. In the **Animation** view, create a new clip and save it as `SeatAnimation.anim` inside the **Assets** folder.
14. Click on the **Record** button and, at **0:00**, highlight all **Transform/Position** fields (x, y, and z) and click on the **Add Keyframe** button to add a new keyframe.

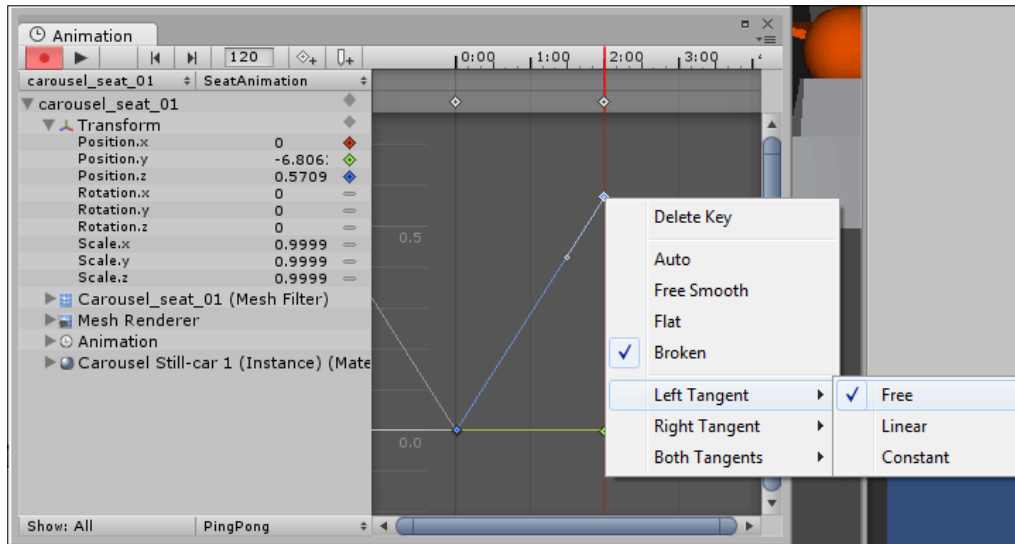


15. Advance the time indicator to **2:00** (or type 120 in the **Counter** field and press the **Enter** key) and add another keyframe.
16. Focus the **carousel_seat_01** in a **Scene** view and move it up in the indicated axis.
A line should indicate the position change over time in the **Animation** view.

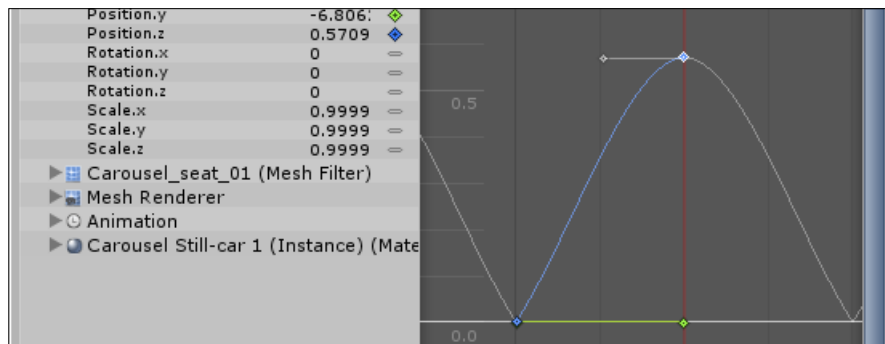


17. In the bottom part of the **Animation** view, click on the appropriate button to change the **Wrap Mode** to **PingPong**.

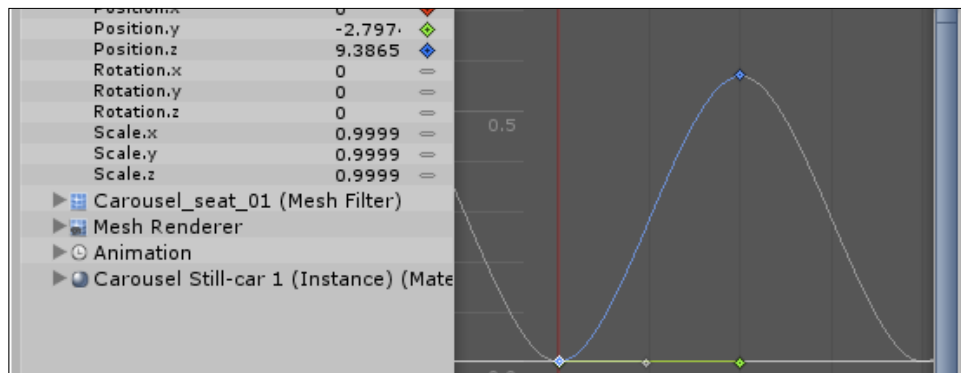
18. Right-click on the keyframe at **2:00** and change its **Left Tangent** to **Free**.



19. Use the handle to change the corner into a curve, as if drawing a bell.



20. Do the same to the **Right Tangent** of the Keyframe at **0:00**. That should ease the animation, making it smoother.



21. Turn off the **Record** button and select the **carousel_seat_01** game object in the **Hierarchy** view. Then, in the **Inspector** view, apply **SeatAnimation** as its default animation clip and make sure the option **Play Automatically** is selected.
22. Play your scene. The **carousel_seat_01** should be animated in its y axis, while still respecting the scene hierarchy by orbiting the main structure.
23. If so you wish, attach the **Animation** component to the other objects named **carousel_seat** and chose **SeatAnimation** as their default clip.

How it works...

Although the animation process itself can be quite intuitive, it might be worthwhile making two points clear:

The **Wrap Mode** sets the behavior for the animation clip once it has finished playing. **Loop** means it will be repeated indefinitely (from start to end); **PingPong**, means that it will be played back and forth (also indefinitely); **Once**, as the name suggests, means it will be played just once. Finally, **Clamp Forever** means that its last frame will be repeated forever.

Also, attaching an animation component to **carousel_seat_01** enables it to be independently animated, a feature that, in our case, made it easier to animate vertically while still retaining its parenting relationship to the main structure.

There's more...

To read more information about the **Animation** view and how to use it, check out Unity's documentation at <http://docs.unity3d.com/Documentation/Components/AnimationEditorGuide.html>.

Dividing animation into clips

Exporting all animations for a 3D model in a single file is very practical in terms of asset management and art production workflow. However, once it has been imported, those individual animations, now merged into a single file, have to be divided into clips. In this recipe, we will learn how to split all animations imported into separate clips, ensuring that they remain suitable for the Legacy Animation System.

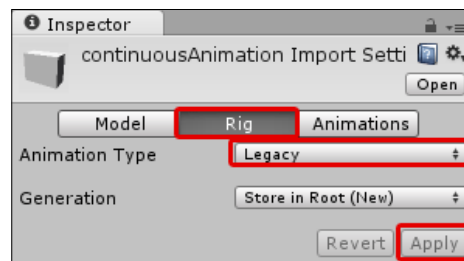
Getting ready

For this recipe, we have prepared an animated 3D model named `continuousAnimation`, available in the `0423_12_02` folder.

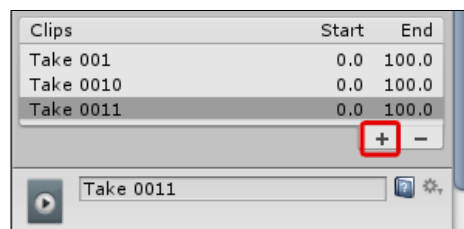
How to do it...

To divide your animation into several clips, perform the following steps:

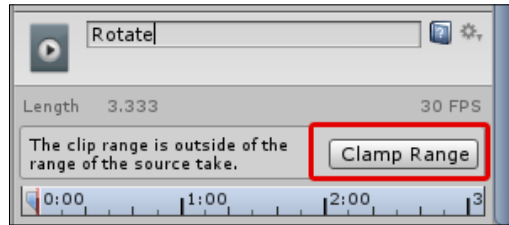
1. Import **continuousAnimation.FBX** into your project.
2. In the **Project** view, select the **continuousAnimation**. Then, in the **Inspector** view, change the active tab to **Rig**. Then set **Animation Type** as **Legacy** and **Generation** as **Store in Root (New)**. Click on **Apply**.



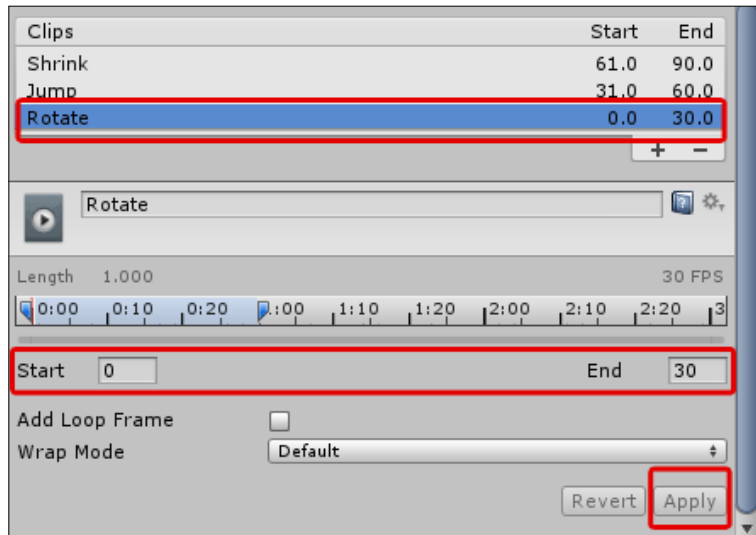
3. Now change the active tab to **Animations**. Watch the animation by hitting the **Play** button in the **Preview** area. There are three clear different movements we will create into separate clips: rotate, jump, and shrink.
4. From the **Clips** list, click twice on the **+** sign to create two more clips.



5. Select **Take 001** and click on the **Clamp Range** button (as shown in the following screenshot). Then, change its name to `Rotate` using the appropriate **Name** field. Do the same to **Take 0010** and **Take 0011**, renaming them as `Jump` and `Shrink`.

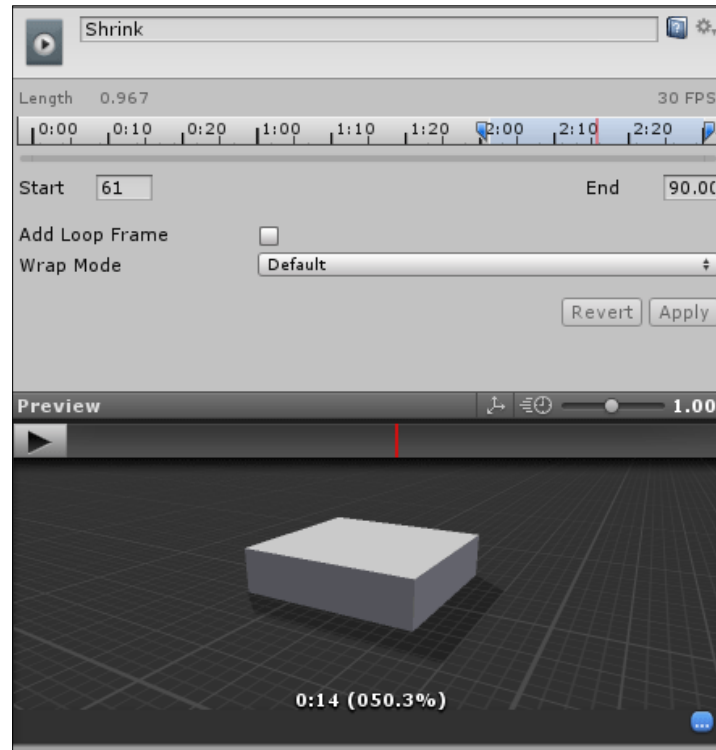


6. Select the **Rotate** clip, set its **Start** as 0 and its **End** as 30. Then click on the **Apply** button.



7. Now select **Jump** and set **Start** as 31 and **End** as 60. Click on **Apply**.
8. Finally, select **Shrink** and set **Start** as 61 and **End** as 90, and click on **Apply**.

9. Your animation is now divided into clips and can be previewed independently.



How it works...

The actual division of the animation into separate clips is made in the **Inspector** view through the object's **Animations** tab, where we have established the beginning and end of each clip.

There's more...

More information on importing and setting animation clips can be found inside Unity's documentation at <http://docs.unity3d.com/Documentation/Manual/Animations.html#ImportFile>.

Importing multiple files as animation clips

Alternatively, instead of exporting a single file containing every animation, you could export multiple files using the `charactername@animationname.fbx` convention (for example, `hero@walk.fbx` and `hero@run.fbx`), and Unity would interpret them as separate animation clips for the same model.

Understanding Wrap Mode and Loop Frame

From the **Import Settings**, you can also select the animation clip's **Wrap Mode**, which is basically an instruction on how the clip should behave after it reaches its last frame. It can be stopped (**Once**), repeated indefinitely (**Loop**), go back and forth indefinitely (**PingPong**), or have its last frame played forever (**ClampForever**).

Also, if you select the **Loop** checkbox, Unity will create an extra frame, identical to the first one, at the end of the animation clip. This should be used if you have selected the **Loop** wrap mode on an animation where the first and last frame are not identical, then it's obvious when it has restarted.

Controlling playback and speed

Playing a specific animation clip is a very common and useful feature—even more if we can also control the playback speed and change it according to the player's input. In this recipe, we will learn how to achieve such results.

Getting ready

For this recipe, we have prepared a package named `shipScene` containing a playable scene including a basic spaceship, a camera that follows it, and some basic geometry. The package is in the `0423_12_03` folder.

How to do it...

To control the playback and animation speed of a clip, perform the following steps:

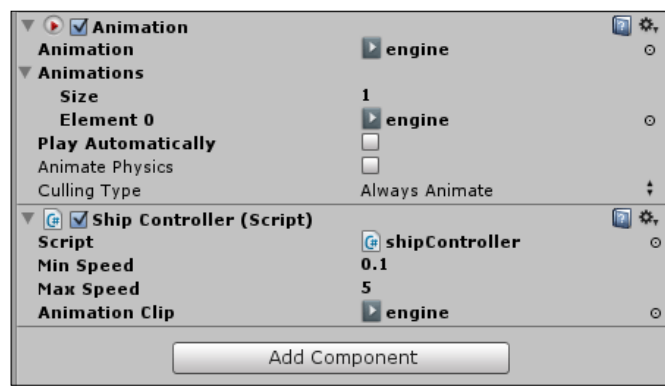
1. Import the **shipScene** package into your project.
2. Open the scene named **11_03** and play it. It's worth noting that the ship can be controlled with the keyboard, although its propeller doesn't move.
3. Stop the scene. In the **Project** view, locate the C# script named `shipController.cs`.
4. Open the script. Add the following code immediately below the line `private float speed = 0.0f;`:

```
public AnimationClip animationClip;

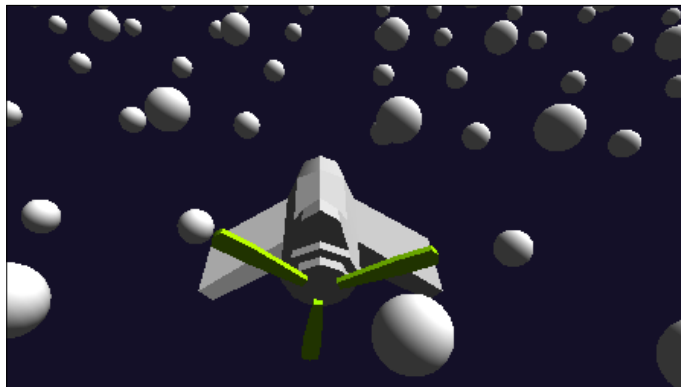
void Start() {
    animation.wrapMode = WrapMode.Loop;
    animation.Play (animationClip.name);
}
```

5. Then immediately below the line `transform.Rotate (0, rotationY, 0);` add the following:

```
animation[animationClip.name].speed = speed;
```
6. Save and close the script.
7. In the **Hierarchy** view, select the **ship** game object. Then, in the **Inspector** view, locate the **Ship Controller** component.
8. In the **Ship Controller** component, change the newly created **Animation Clip** parameter to **engine** by either dragging it from the **Project** view or selecting it from the list.



9. Play the scene. The propeller should be animated and its playback speed should reflect the actual speed of the ship.



How it works...

As you have probably noticed, the whole process is very straightforward; all we needed to do was the following:

- ▶ Creating the variable `animationClip` to hold our clip
- ▶ Setting the animation's **Wrap Mode** to **Loop** and playing it at `Start()`
- ▶ Adjusting its speed in the `Update()` function to reflect the ship's `Translation speed`

There's more...

If you want to play your animation in reverse, just make the speed a negative value (in our case, something similar to `animation[animationClip.name].speed = - speed;`).

Setting up an animated Mixamo character

Since we will be using a character provided by Mixamo in our character-based recipes, a good place to start is learning how to set them up. As you will learn from this recipe, setting up a Mixamo character involves one or two particular steps.

This recipe is based on the instructions provided by Mixamo, available as a `README.txt` file within the provided project folder.

Getting ready

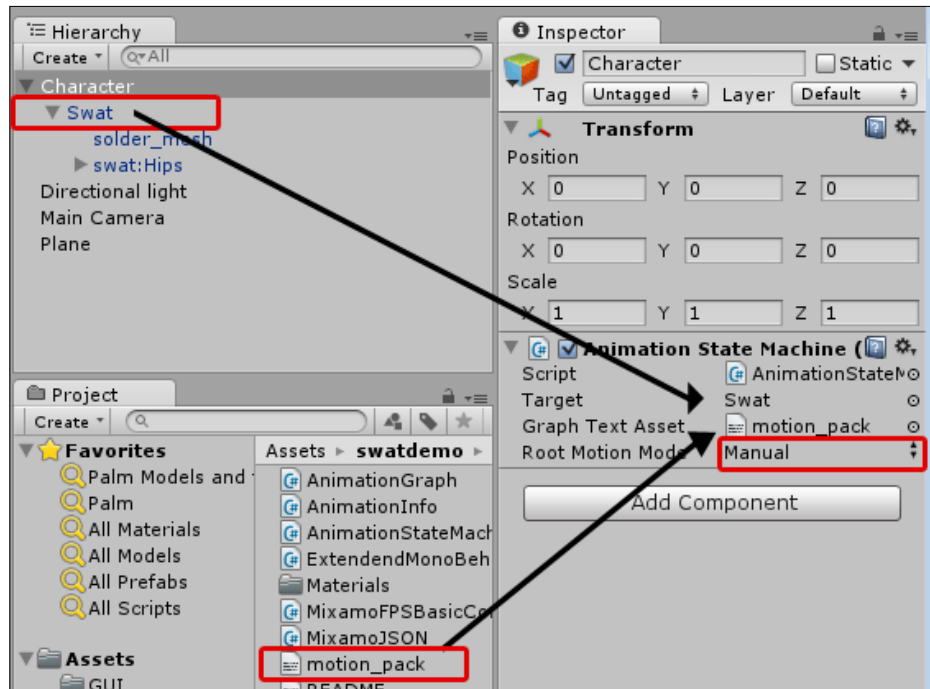
For this chapter, we have prepared a project folder named `MixamoProject_11` containing several assets such as levels, an animated character named `Swat`, and some props. You can find it in the `0423_12_codes` folder.

How to do it...

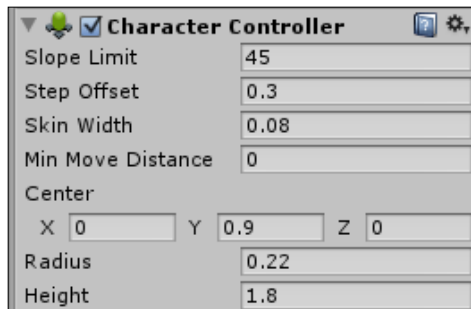
To set up a Mixamo animated character, perform the following steps:

1. Open the **MixamoProject** project and open the level named **11_04** (under the **Levels** folder). This is a level containing a plane, a camera, and a directional light.
2. Create an empty game object (by navigating to **GameObject | Create Empty**). Rename it to `Character` and in the **Inspector** view, set its **Position** to **X: 0, Y: 0, and Z: 0**.
3. From the **Project** view, locate the model named **Swat** (inside the folder **swatdemo**) and drag it into the **Character** game object, in the **Hierarchy** view. It should be added to the scene as a child of the **Character**.

4. In the project folder, locate the script named **AnimationStateMachine.cs** located inside the **swatdemo** folder, and attach it to the **Character** game object.
5. Select the **Character** game object and in the **Inspector** view, access its **Animation State Machine** component. Then, drag **Swat** from the **Hierarchy** view into the **Target** field and the **motion_pack** from the **Project** view into the **Graph text Asset** field. Finally, set its **Root Motion Mode** to **Manual**.



6. Now, add a **Character Controller** to the **Character** game object (by navigating to **Components | Physics | Character Controller**). Then, in the **Inspector** view, change the **Y** value of the field **Center** to 0.9, the value of **Radius** to 0.22, and the value of **Height** to 1.8.



7. Locate the script named **MixamoFPSBasicControlScript** (inside the folder **swatdemo**) and attach it to the **Character** game object.
8. Play the scene. Use the WASD keyboard control scheme to control your animated character. You can also jump using the Space bar, rotate by pressing the *Q* and *E* keys, reload the weapon by pressing the *R* key, and toss a grenade by pressing the *F* key.

How it works...

Mixamo scripts attached to the **Character** game object work in conjunction with Unity's **Character Controller** physics component. While the **Animation State Machine** organizes how the animations included in the **Swat** model should be played, the **MixamoFPSBasicControlScript** designates a control scheme to our character, triggering the animation clips (as previously organized) according to the player's input.

There's more...

The following is a bit more information on Mixamo characters and Unity's controller:

Finding more Mixamo tutorials for Unity

You can learn more about integrating your Mixamo character with Unity at http://www.mixamo.com/c/help/workflows_unity.

Adding rigid props to animated characters

In case you haven't added a sufficient number of props to your character when modeling and animating it, you might want to give him the chance of collecting new ones at runtime. In this recipe, we will learn how to instantiate a game object and assign it to a character, respecting the animation hierarchy.

Getting ready

For this recipe, we have prepared a project folder named **MixamoProject_11** containing several assets such as levels, animated characters, and props. You can find it in the **0423_12_codes** folder.

How to do it...

To add props at runtime to your character, perform the following steps:

1. Open the **MixamoProject** project and open the level named **11_05** (under the **Levels** folder).

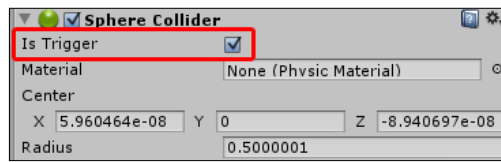
2. You should see Mixamo's animated S.W.A.T. soldier and two spheres. Those spheres will trigger the addition of new props into the character.
3. In the **Project** view, create a new C# script named `AddProp.cs`.
4. Open the script and add the following code:

```
using UnityEngine;
using System.Collections;

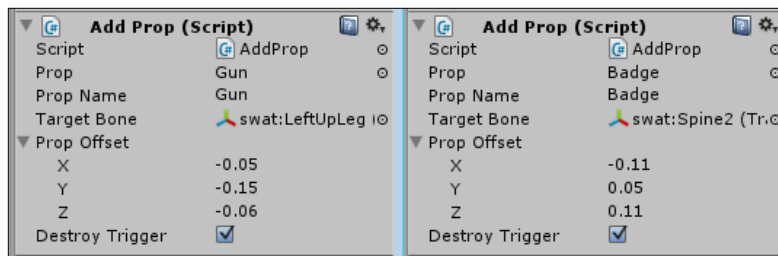
public class AddProp : MonoBehaviour {
    public GameObject prop;
    public string propName;
    public Transform targetBone;
    public Vector3 propOffset;
    public bool destroyTrigger = true;

    void OnTriggerEnter ( Collider collision ) {
        if (targetBone.IsChildOf(collision.transform)) {
            bool checkProp = false;
            foreach(Transform child in targetBone){
                if (child.name == propName)
                    checkProp = true;
            }
            if(!checkProp){
                GameObject newprop;
                newprop = Instantiate(prop, targetBone.position,
targetBone.rotation) as GameObject;
                newprop.name = propName;
                newprop.transform.parent = targetBone;
                newprop.transform.localPosition += propOffset;
                if(destroyTrigger)
                    Destroy(gameObject);
            }
        }
    }
}
```

5. Save and close the script.
6. Attach the script `AddProp.cs` by dragging it from the **Project** view into the objects named **Sphere Blue** and **Sphere Red**, in the **Hierarchy** view.
7. Select each sphere, and in the **Sphere Collider** component (located in the **Inspector** view), select the **Is Trigger** option.



8. Select the **Sphere Blue** object and check out its **Add Prop** component.
9. First, let's assign a Prefab to the **Prop** field. In the **Project** view, expand the **Props** folder. Then, drag the Prefab named **Gun** into the **Prop** field.
10. In the **Prop Name** field, type Gun.
11. In the **Target Bone** field, select the **swat:LeftUpLeg** transform from the list (or drag it from the **Hierarchy** view: it is a child of the **Character** game object).
12. Expand the **Prop Offset** field and type in the values of **X**, **Y**, and **Z** as -0.05 , -0.15 , and -0.06 respectively. Finally, deselect the **Destroy Trigger** option.
13. Select the **Sphere Red** object and fill in the variables as indicated: **Prop: Badge**; **Prop Name: Badge**; **Target Bone: swat:Spine2**; **Prop Offset: X = -0.11, Y = 0.05, and Z = 0.11**. Also, deselect **Destroy Trigger**.



14. Play the scene. Using the WASD keyboard control scheme, direct the character to **Spheres**. Colliding with them will add a new handgun holster (to be attached to the character's left leg) and a badge (to be attached to the left-hand side of the character's chest).



How it works...

Once it's been triggered by the character, the script attached to the spheres instantiate the assigned Prefabs, making them children of the bones they have been "placed into". The **Prop Offset** option can be used to fine-tune the exact position of the prop (relative to its parent transform). As the props become parented by the bones of the animated character, they will follow and respect its hierarchy and animation. Note that the script checks for pre-existing props of the same name before actually instantiating a new one.

There's more...

You could make a similar script to remove props. In that case, the `OnTriggerEnter()` function would contain only the following code:

```
if (targetBone.IsChildOf(collision.transform)) {
    foreach(Transform child in targetBone){
        if (child.name == propName)
            Destroy (child.gameObject);
    }
}
```

Making an animated character throw an object

Now that your animated character is ready, you might want to coordinate his actions with his animation states. In this recipe, we will exemplify this by making the character throw an object when the player presses the appropriate button. Also, we will make sure the action corresponds to the character's animation.

Getting ready

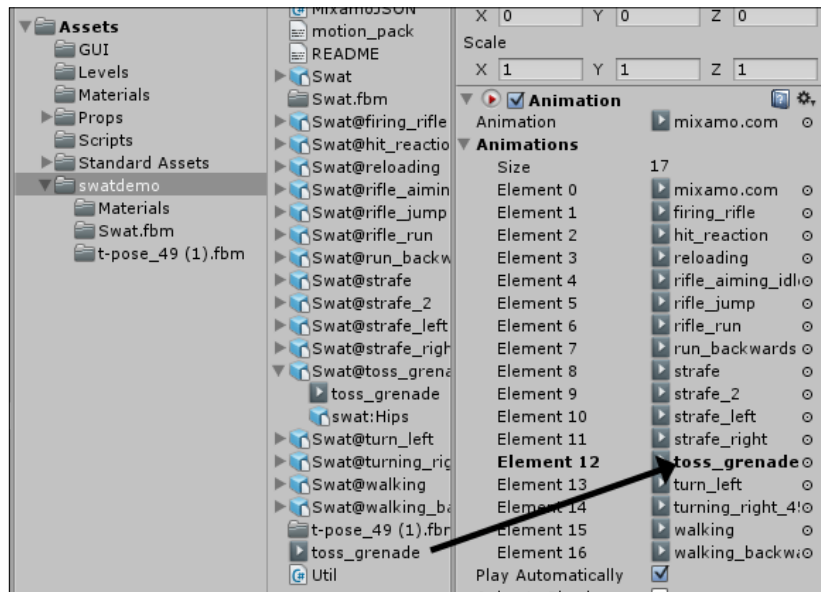
For this recipe, we have prepared a project folder named `MixamoProject_11` containing several assets such as levels, animated characters, and props. You can find it in the `0423_12_codes` folder.


How to do it...

To make your character throw an Easter egg(!), perform the following steps:

1. Open the **MixamoProject** project and open the level named **11_06** (under the **Levels** folder).

2. Play the level and press the *F* key on your keyboard. The character will move as if throwing something with his right hand.
3. Stop the level. In the **Project** view, locate the animation clip named **toss_grenade** and duplicate it.
4. Expand the **Character** game object and select its child **Swat**. Now, in the **Inspector** window, drag the newly-created **toss_grenade** clip into the slot occupied by the old animation clip of the same name.



 We need to duplicate externally created animation clips, otherwise we won't be able to add animation events later.

5. Attach the following C# script to the game object named **Swat** (child of the game object named **Character**).

```
// file: ThrowObject.cs
using UnityEngine;
using System.Collections;

public class ThrowObject : MonoBehaviour {
    public GameObject projectile;
    public Vector3 projectileOffset;
    public Vector3 projectileForce;
    public Transform character;
```

```

public Transform charactersHand;

public void Prepare () {
    projectile = Instantiate(projectile, charactersHand.position,
        charactersHand.rotation) as GameObject;
    if (projectile.GetComponent<Rigidbody>())
        Destroy(projectile.rigidbody);

    projectile.GetComponent<SphereCollider>().enabled = false;
    projectile.name = "projectile";
    projectile.transform.parent = charactersHand;
    projectile.transform.localPosition = projectileOffset;
    projectile.transform.localEulerAngles = Vector3.zero;
}

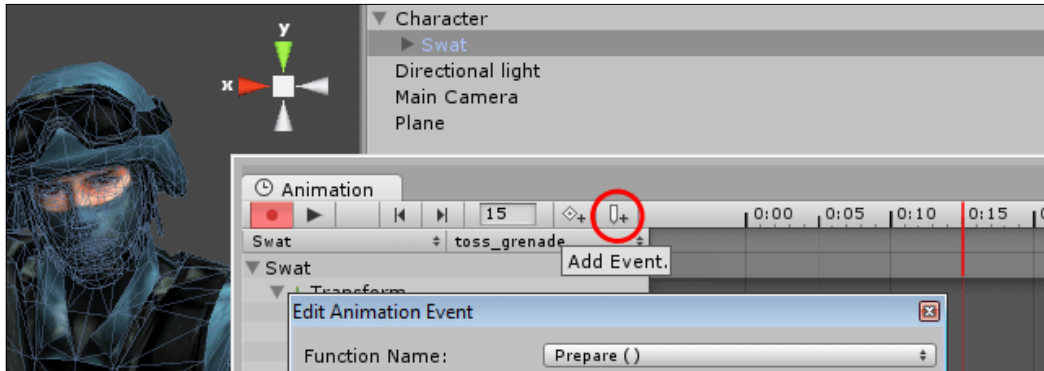
public void Throw () {
    projectile.transform.rotation = character.rotation;
    projectile.transform.parent = null;
    projectile.GetComponent<SphereCollider>().enabled = true;
    projectile.AddComponent<Rigidbody>();
    projectile.rigidbody.AddRelativeForce(projectileForce);
}
}

```

6. Select the **Swat** object. In the **Inspector** view, check out its **Throw Object** component. From the **Project** view, drag the Prefab named **EasterEgg** (available in the **Props** folder) into the **Projectile** field.
7. From the **Hierarchy** view, drag the **Character** game object into the **Character** field of the component. Also, in the field **Characters Hand**, select the **swat:RightHand** transform.
8. As we are still on the **Throw Object** component, fill in the variables as indicated: **Projectile Offset: X = 0.07, Y = -0.04, Z = 0; Projectile Force: X = 0, Y = 200, Z = 500** (as shown in the following screenshot).



9. Select the **Swat** game object and access the **Animation** view (**Window | Animation**). Now, select **toss_grenade** from the animation list.
10. Drag the time indicator to **0:15** and click on the **Add Event** button. From the dialog box that pops up, select the **Prepare()** function.



11. Now, drag the time indicator to **1:25** and click on the **Add Event** button. This time, select the **Throw()** function.
12. Play your scene. Your character should now be able to throw an Easter egg when you press the **F** key.

How it works...

Once the **toss_grenade** animation reaches **0:15**, it calls the function named **Prepare()** inside the **ThrowObject** script. This function instantiates a Prefab, now named **projectile**, into the character's hand, also making it respect the character's hierarchy. In addition, it disables the Prefab's collider and destroys its **Rigidbody** component, provided it has one.

Later, when the **toss_grenade** animation reaches **1:25** it calls the **Throw()** function, which enables the projectile's collider, adds a **Rigidbody** component to it, and makes it independent from the **Character** object. Finally, it adds a relative force to the projectile's **Rigidbody** component, so it will behave as if it were thrown by the character.

There's more...

Instead of using the **Animation** view, we could have created animation events via **Script**. For more information, check Unity's documentation at <http://docs.unity3d.com/Documentation/ScriptReference/AnimationEvent.html>.

Applying ragdoll physics to a character

Action games often make use of ragdoll physics to simulate the character's body reaction to being unconsciously under the effect of a hit or explosion. In this recipe, we will learn how to set up and activate ragdoll physics to our character whenever he steps on a landmine object. We will also use the opportunity for reset the character's position and animations a number of seconds after that event.

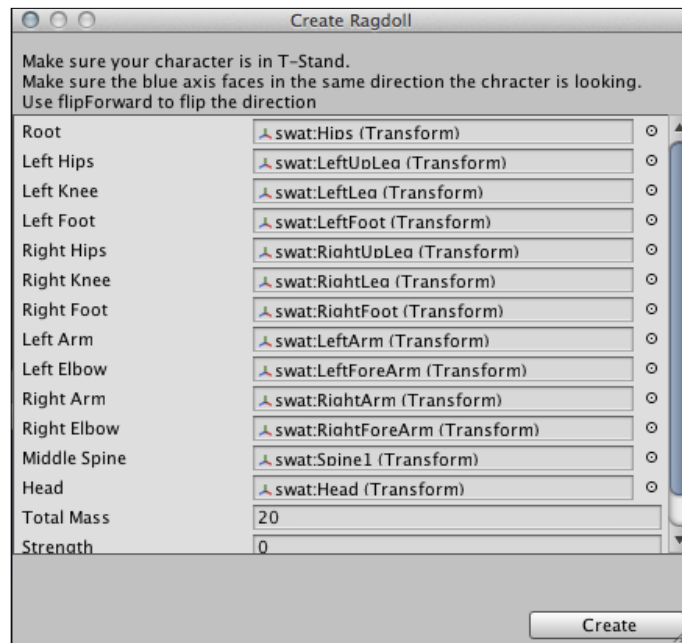
Getting ready

For this recipe, we have prepared a project folder named `MixamoProject_11` containing several assets such as levels, animated characters, and props. You can find it in the `0423_12_codes` folder.

How to do it...

To implement ragdoll physics into your character, perform the following steps:

1. Open the **MixamoProject** project and open the level named **11_07** (under the **Levels** folder).
2. You should see Mixamo's animated S.W.A.T soldier and two cylinders: the **Landmine** and the **Spawnpoint**.
3. First, let's set up our ragdoll. Navigate to **GameObject | Create Other | Ragdoll**. The Ragdoll Wizard should pop-up.
4. Assign values to the transforms as follows:
 - ❑ **Root:** `swat:Hips`
 - ❑ **Left Hips:** `swat:LeftUpLeg`
 - ❑ **Left Knee:** `swat:LeftLeg`
 - ❑ **Left Foot:** `swat:LeftFoot`
 - ❑ **Right Hips:** `swat:RightUpLeg`
 - ❑ **Right Knee:** `swat:RightLeg`
 - ❑ **Right Foot:** `swat:RightFoot`
 - ❑ **Left Arm:** `swat:LeftArm`
 - ❑ **Left Elbow:** `swat:LeftForeArm`
 - ❑ **Right Arm:** `swat:RightArm`
 - ❑ **Right Elbow:** `swat:RightForeArm`
 - ❑ **Middle Spine:** `swat:Spine1`
 - ❑ **Head:** `swat:Head`



5. In the **Project** view, create a new C# script named `RagdollCharacter.cs`.
6. Open the script and add the following code:

```
using UnityEngine;
using System.Collections;

public class RagdollCharacter : MonoBehaviour {
    private float hitTime;
    private bool wasHit = false;

    void Start () {
        DeactivateRagdoll();
    }

    void Update () {
        public void ActivateRagdoll(){
            this.GetComponent<AnimationStateMachine>().enabled = false;
            this.GetComponent<MixamoFPSBasicControlScript>().enabled =
false;
```

```

        foreach(Rigidbody bone in GetComponentsInChildren<Rigidbody>()) {
            bone.isKinematic = false;
            bone.detectCollisions = true;
        }
        foreach(Animation anim in GetComponentsInChildren<Animation>()) {
            anim.GetComponent<Animation>().enabled = false;
        }
        wasHit = true;
        hitTime = Time.time;
    }

    public void DeactivateRagdoll() {
        this.GetComponent<AnimationStateMachine>().enabled = true;
        this.GetComponent<MixamoFPSBasicControlScript>().enabled = true;
        foreach(Rigidbody bone in GetComponentsInChildren<Rigidbody>()) {
            bone.isKinematic = true;
            bone.detectCollisions = false;
        }
        foreach(Animation anim in GetComponentsInChildren<Animation>()) {
            anim.GetComponent<Animation>().enabled = true;
        }
        transform.position = GameObject.Find("Spawnpoint").transform.position;
        transform.rotation = GameObject.Find("Spawnpoint").transform.rotation;
        wasHit = false;
    }
}

```

7. Save and close the script.
8. Attach the **RagdollCharacter.cs** script to the **Character** game object.
9. Select the **Character** and change its **Tag** to **Player**.
10. In the **Project** view, create a new C# script named **Landmine.cs**.
11. Open the script and add the following code:

```

using UnityEngine;
using System.Collections;

public class Landmine : MonoBehaviour {
    public float range = 50.0f;
}

```



```

public float force = 2000.0f;

void OnTriggerEnter ( Collider collision ) {
    if (collision.gameObject.tag == "Player") {
        collision.GetComponent<RagdollCharacter>().ActivateRagdoll();
        Vector3 explosionPos = transform.position;
        Collider[] colliders = Physics.
        OverlapSphere(explosionPos, range);
        foreach (Collider hit in colliders) {
            if (hit.rigidbody)
                hit.rigidbody.AddExplosionForce(force,
                explosionPos, range, 3.0F);
        }
    }
}

```

12. Save and close the script.
13. Attach the script to the **Landmine** game object.
14. Play the scene. Using the WASD keyboard control scheme, direct the character to the **Landmine** game object. Colliding with it will activate the character's ragdoll physics and apply an explosion force to it. As a result, the character should be thrown away to a considerable distance.

How it works...

Unity's Ragdoll Wizard assigns, to selected transforms, the components **Collider**, **Rigidbody**, and **Character Joint**. In conjunction, those components make ragdoll physics possible. However, those components must be disabled whenever we want our character to be animated and controlled by the player. In our case, we switch those components on and off using the **RagdollCharacter** script and its two functions: `ActivateRagdoll()` and `DeactivateRagdoll()`, the latter including instructions to respawn our character in the appropriate place.

For testing purposes, we have also created the **Landmine** script, which calls **RagdollCharacter** script's `ActivateRagdoll()` function. It also applies an explosion force to our ragdoll character, violently throwing him outside the explosion site.

There's more...

Instead of resetting the character's transform settings, you could have destroyed his `gameObject` and instantiated a new one over the respawn point using `Tags`. For more information on that subject, check Unity's documentation at <http://docs.unity3d.com/Documentation/ScriptReference/GameObject.FindGameObjectsWithTag.html>.

Rotating the character's torso to aim

When playing a third-person character, you might want him to aim his weapon at a target that is not directly in front of him without making him change his direction. In those cases, you will need to apply what is called **procedural animation**, which does not rely on pre-made animation clips, but rather on the processing of other data such as player input to animate the character. In this recipe, we will use this technique to rotate the character's torso.

Getting ready

For this recipe, we have prepared a project folder named `MixamoProject_11` containing several assets such as levels, animated characters, and props. You can find it in the `0423_12_codes` folder.

How to do it...

To rotate the character's torso using the mouse, perform the following steps:

1. Open the **MixamoProject** project and open the level named **11_08** (under the **Levels** folder).
2. You should see Mixamo's animated S.W.A.T. soldier. We have included three different cameras for you to experiment with (they are children of the **Character** game object).
3. In the **Project** view, create a new C# script named `MouseAim.cs`.
4. Open the script and add the following code:

```
using UnityEngine;
using System.Collections;

public class MouseAim : MonoBehaviour
{
    public Transform spine;
    public Transform armedHand;
    public bool lockY = false;
    public float compensationYAngle = 20.0f;
    public float minAngle = 308.0f;
    public float maxAngle = 31.0f;
    public Texture2D targetAim;
    private Vector2 aimLoc;
    private bool onTarget = false;

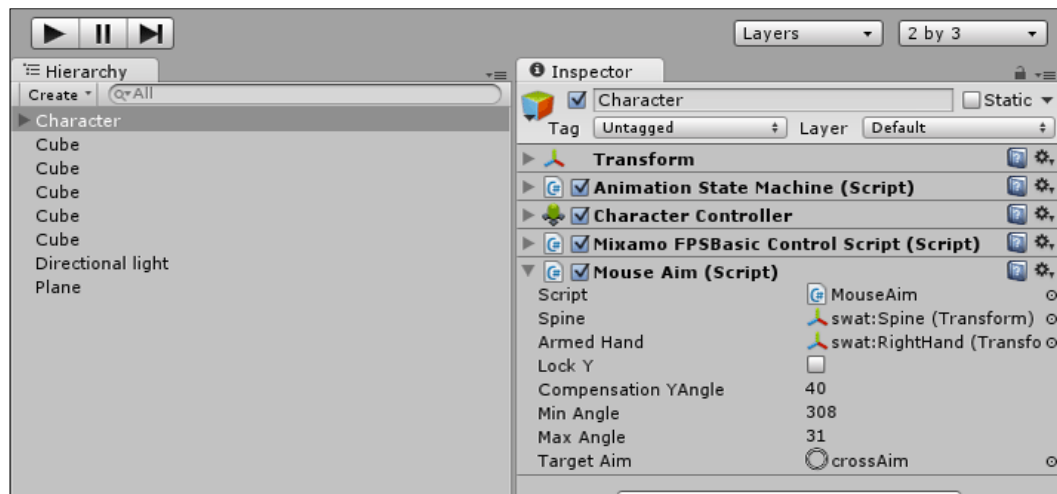
    public void Aim() {
        Vector3 point = Camera.main.ScreenToWorldPoint(new
        Vector3(Input.mousePosition.x, Input.mousePosition.y, Camera.main.
        farClipPlane));
```

```

        if (lockY)
            point.y = spine.position.y;
        Vector3 relativePoint = transform.
InverseTransformPoint(point.x, point.y, point.z);
        if (relativePoint.z < 0){
            Vector3 inverseZ = transform.InverseTransformPoint(rel
ativePoint.x, relativePoint.y, -relativePoint.z);
            point = inverseZ;
        }
        spine.LookAt(point, Vector3.up);
        Vector3 comp = spine.localEulerAngles;
        comp.y = spine.localEulerAngles.y + compensationYAngle;
        spine.localEulerAngles = comp;
        if (spine.localEulerAngles.y > maxAngle && spine.
localEulerAngles.y < minAngle){
            if (Mathf.Abs((spine.localEulerAngles.y - minAngle)) <
Mathf.Abs((spine.localEulerAngles.y - maxAngle))){
                Vector3 min = spine.localEulerAngles;
                min.y = minAngle;
                spine.localEulerAngles = min;
            }else{
                Vector3 max = spine.localEulerAngles;
                max.y = maxAngle;
                spine.localEulerAngles = max;
            }
        }
        RaycastHit hit;
        if (Physics.Raycast(armedHand.position, point, out hit)){
            onTarget = true;
            aimLoc = Camera.main.WorldToViewportPoint(hit.point);
        }else{
            onTarget = false;
            aimLoc = Camera.main.WorldToViewportPoint(point);
        }
        //Debug.DrawRay (armedHand.position, point, Color.red);
    }
    void OnGUI(){
        int sw = Screen.width;
        int sh = Screen.height;
        GUI.DrawTexture(new Rect(aimLoc.x * sw - 8, sh - (aimLoc.y
* sh) - 8, 16, 16), targetAim, ScaleMode.StretchToFill, true,
10.0F);
    }
}

```

5. Save and close the script.
6. Attach the **MouseAim.cs** script by dragging it from the **Project** view into the **Character** game object.
7. Select the **Character** game object, and in the **Mouse Aim** component (located in the **Inspector** view), fill in the values for variables as indicated: **Spine:** **swat:Spine**; **Armed Hand:** **swat:RightHand**, **Lock Y:** **deselected** (unless using the **Camera_top**); **Compensation YAngle:** **40**; **Min Angle:** **308**; **Max Angle:** **31**. Also, drag the **crossAim** texture from the **GUI** folder in the **Project** view into the **Target Aim** field.



8. Now, in the **Project** view, locate and open the script named **AnimationStateMachine.cs**.
9. Find the code `public class AnimationStateMachine : ExtendedMonoBehaviour {` and add the following line below it:


```
public MouseAim mouseAim;
```
10. Then, find the code `void Start () {` and add the following line below it:


```
mouseAim = GetComponent<MouseAim>();
```
11. Locate the **LateUpdate()** function and add the following lines just before it closes:


```
if (mouseAim)
    mouseAim.Aim();
```
12. Save the script and play the scene. You should now be able to rotate the character's torso by moving the mouse cursor around the screen. Even better, a GUI texture will be displayed wherever he is aiming at.



How it works...

Our **Aim()** function starts by converting our bi-dimensional mouse cursor screen's coordinates to 3-dimensional world space coordinates (stored in the **point** variable). Then, it rotates the character's torso towards the **point** location, using the **LookAt()** command to do so. Additionally, it makes sure the spine does not extrapolate **Min Angle** and **Max Angle**, which would cause distortions to the character model.

In the process, we convert the target point from its absolute world coordinates to its position in relation to the character, and vice-versa. This is done so we can detect whether the target point is located behind the character and, if it is, manipulate the point so the character keeps aiming forward.

Also, we have included a **Compensation YAngle** variable that makes it possible for us to fine-tune the character's alignment with the mouse cursor.

Please note that the **Aim()** function is called from the **LateUpdate()** function within the character's **AnimationStateMachine** script. This is done to make sure our transform manipulations will override the character's animation clips.

There's more...

In case you want to implement shooting, we have provided you with a place to start by casting a ray from the character's armed hand to the target point. To see it as you test the scene, uncomment the line `Debug.DrawRay (armedHand.position, point, Color.red);`.

The ray cast detects collisions with surrounding objects to which the `targetAim` texture, usually drawn over the mouse cursor location, should snap.

Blending, mixing, and cross-fading animation states

Instead of generating an astronomic number of animation clips for our characters, we can increase their complexity and completeness by combining their pre-existing animation states. In this recipe, we will use three techniques to achieve that goal. Those techniques are as follows:

- ▶ **Cross-fading:** Creating a smooth transition from an animated state to the next, instead of abruptly stopping an animation to start a new one. In our recipe, we will cross-fade the idle and walk states of the character.
- ▶ **Mixing:** Allowing specific parts of your model to play a different animation than the rest. In our recipe, the player will be able to control the character's antenna animation separately.
- ▶ **Additive blending:** Overlaying two different animations. In our recipe, we will combine the animations of the character leaning towards his sides with either idle or walk animation, depending on the character's current state.

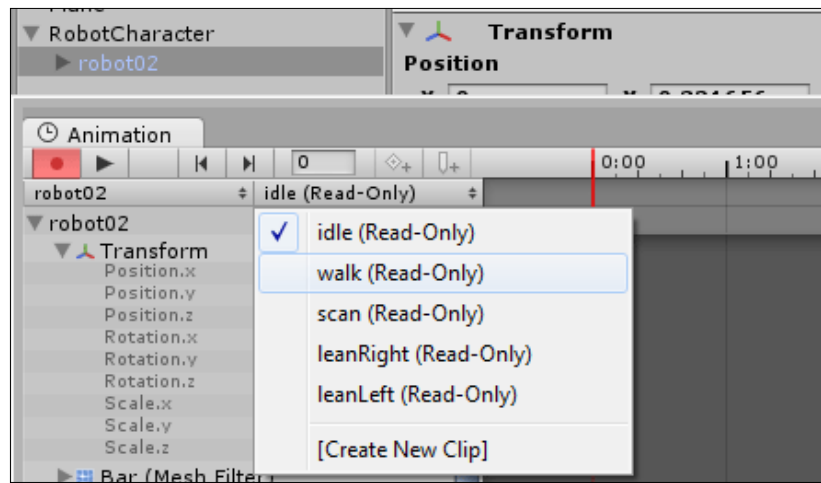
Getting ready

For this recipe, we have prepared a Prefab named `combiningStates` including a level that contains a robot character with basic input controls. You can find it in the `0423_12_09` folder.

How to do it...

To combine animation states, perform the following steps:

1. Import the **combiningStates** package into your project.
2. Open the level named **11_09** and play it. You should be able to control the **RobotCharacter** by using the left, right, up, and down keys to move or rotate it.
3. Stop the game. In the **Hierarchy** view, expand the **RobotCharacter** game object and select its child named **robot02**. Then, open the **Animation** view (**Window | Animation**). You can now preview each animation clip we are about to use.



4. In the **Project** view, open the C# script named **RobotController.cs**.
5. First, include the following code right before the **Update()** function:

```
public GameObject model;
public Transform antenna;
public string idle = "idle";
public string walk = "walk";
public string scan = "scan";
public string leanRight = "leanRight";
public string leanLeft = "leanLeft";

void Start()
{
    model.animation[idle].layer = 0;
    model.animation[walk].layer = 0;
    model.animation[scan].layer = 3;
    model.animation[scan].AddMixingTransform(antenna);
    model.animation[leanRight].layer = 4;
    model.animation[leanLeft].layer = 4;
    model.animation[leanRight].blendMode = AnimationBlendMode.
Additive;
    model.animation[leanLeft].blendMode = AnimationBlendMode.
Additive;
    model.animation[leanRight].enabled = true;
    model.animation[leanLeft].enabled = true;
    model.animation[leanRight].weight = 1.0f;
}
```

```
        model.animation[leanLeft].weight = 1.0f;
        model.animation[idle].wrapMode = WrapMode.Loop;
        model.animation[walk].wrapMode = WrapMode.Loop;
        model.animation[scan].wrapMode = WrapMode.Once;
        model.animation[leanRight].wrapMode = WrapMode.ClampForever;
        model.animation[leanLeft].wrapMode = WrapMode.ClampForever;
        model.animation.Stop();
    }
```

6. Then, include the following code inside the **Update()** function, just before it closes:

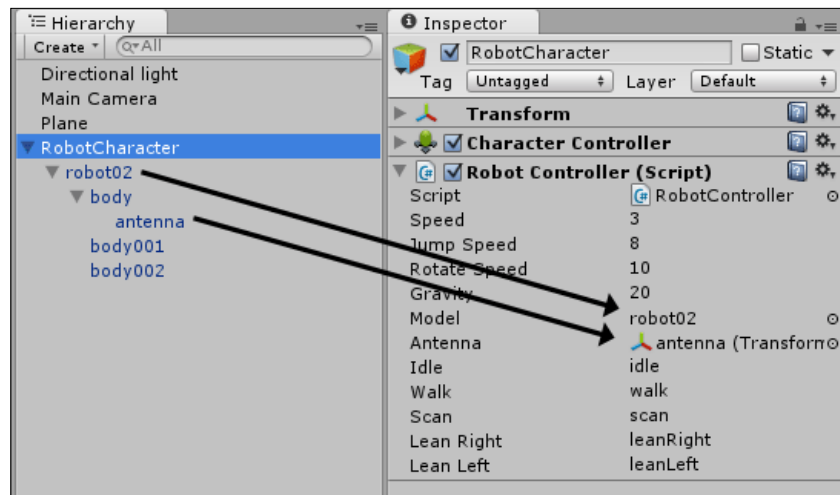
```
    if (Input.GetAxis("Vertical") >= 0) {
        model.animation[walk].speed = 1;
    } else {
        model.animation[walk].speed = -1;
    }

    if (controller.velocity.x != 0 || controller.velocity.z != 0) {
        model.animation.CrossFade(walk);
    } else {
        model.animation.CrossFade(idle);
    }

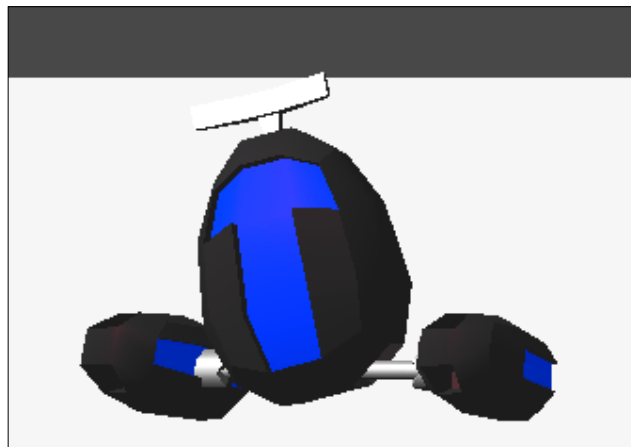
    var lean = Input.GetAxis("Horizontal");
    model.animation[leanLeft].normalizedTime = -lean;
    model.animation[leanRight].normalizedTime = lean;
    model.animation.Blend(leanLeft, 1f, 1f);
    model.animation.Blend(leanRight, 1f, 1f);

    if (Input.GetButton("Fire1"))
        model.animation.Play(scan);
```

7. Save and close the script.
8. Select the **RobotCharacter** game object and in the **Inspector** view, access the **Robot Controller** component. As most of our variables are already filled in, we just need to assign values to **Model** and **Antenna**.
9. In the **Model** field, select the **robot02** game object (child of **RobotCharacter**).
10. In the **Antenna** field, select the antenna transform (**RobotCharacter** | **robot02** | **body**).



11. Play the scene. The character should cross-fade between **idle** and **walk**, blend **leanRight** and **leanLeft** animations when the character is rotating, and whenever the Fire button is pressed, animate the antenna as in the **scan** animation clip.



How it works...

Cross-fading: This is quite straightforward. Instead of initiating a clip with **Play()**, just use **CrossFade()**. Note that both **idle** and **walk** wrap modes are set to **Loop**.



`CrossFade()` can also set the duration of fading and the `PlayMode`.

Mixing: First, we have set assigned **antenna** in the **AddMixingTransform()** command. Then, all we needed to do was to use the **Play()** command to initiate the **scan** animation (now restricted to the **antenna**) whenever the Fire button is pressed. Note that the **scan** wrap mode is set to **Once**.

Additive Blending: First we have set the blend modes for **leanRight** and **leanLeft** as **Additive**. Then, we get the **Vertical Axis** value (finding out whether the player is pressing the left or right arrow key) and use it as the **lean** variable. That variable is used to determine where, in its playing time, the **lean** animation will be when we combine it with the other animations being played by using the **Blend()** command.

There's more...

There's much more you can achieve with animation scripting. Additive blending, for instance, can be used to blend facial expressions into your characters.

Unity's page on the subject has some interesting examples (in fact, the code in our script is heavily based on their sample code). You can check it out at <http://docs.unity3d.com/Documentation/Manual/AnimationScripting40.html>.

Using animation events to play audio clips

When an animation component of a game object is being played, it is very straightforward to use animation events to send messages to scripted components of the same game object. One simple, yet effective, utilization of this technique is to use animation events to trigger the playing of audio clips at selected frames in the animation sequence.

Getting ready

You'll find two water droplet sounds in the 0423_12_10 folder.

How to do it...

To use animation events to play audio clips, perform the following steps:

1. Set up a new scene by moving the **Main Camera** to (0, 1, -3) and add a directional light.
2. Create a cube game object at (0, 0, 0) named `AnimatedCube`, and add an `AudioSource` component to this cube.



This cube should be visible at the bottom-center of the **Game** panel with current Unity default new scene settings—if it isn't visible in the **Game** panel, then move it so that it is!

3. Add the following script class to **AnimatedCube**:

```
// file: CubeAnimMethods.cs
using UnityEngine;
using System.Collections;

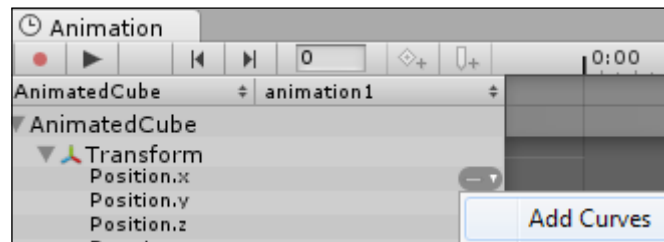
public class CubeAnimMethods : MonoBehaviour {
    public void PlayOneShot(AudioClip clip) {
        audio.PlayOneShot(clip);
    }
}
```

4. Ensure that an **Animation** panel is visible.



A common arrangement is **Window | Layout | Tall**, and then adding an **Animation** panel to the **Game** panel in the lower-left of the IDE window.

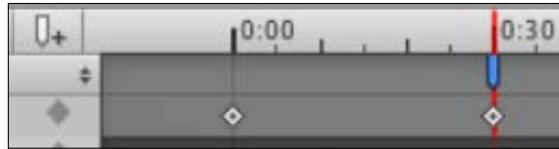
5. First ensure **AnimatedCube** is selected in the **Hierarchy** panel, then in the **Animation** panel, create a new animation clip named `animation1`.
6. In the **Animation** panel, select **Add Curves** for component: **AnimatedCube | Transform | Position.x**.



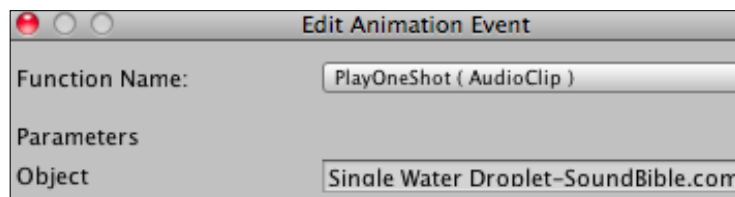
We need to duplicate externally-created animation clips; otherwise we won't be able to add animation events later.

7. Move the playhead to frame **30** (either drag the red line, or change the 0 to 30 in the **Frame Number** textbox), and add a key-frame at frame **30** by clicking on the diamond tool with the **+** sign (next to the **Frame Number** textbox).

8. Move the cube to position (1, 0, 0) (to the right-hand side of the camera view).
9. Ensuring the playhead is still at frame **30**, add an animation event by clicking on the pencil tool with the **+** sign (next to the **New Keyframe** tool).



10. Edit the blue animation event by clicking on it, and from the pop-up window, select the **PlayOneShot()** method. Once selected, drag your audio clip into the **Parameters** slot:



11. Change the **Repeat** setting of the animation from **Default** to **Ping Pong**.

How it works...

An animation in an animation clip of a game object can send a message to call methods from scripted components of the same game object. In this recipe, the **PlayOneShot()** method was called from an animation event in the animation we created for game object **AnimatedCube**. Since this method requires an **AudioClip** parameter, the **Edit Animation Event** pop-up window created a parameter slot into which we could drag the audio clip we wished to be played at that frame.

There's more...

The following are some details you don't want to miss:

Offering a combo drop-down list of audio clips to choose from

The use of an enumerated type and an extra method can remove the need for the person editing the animation to have to locate and drag the sound clip into the parameter slot. This approach might be useful either to simplify animation editing for a less technical team member, or to ensure only one from a limited set of audio clips may be played for a given game object. See the following screenshot for how this would look.

The approach works by making the animation event method's `message` parameter an integer from a restricted range (via nicely named enumerations), and then using a `switch` statement to select one of the public audio clips to actually play. These animation clips' variables will then need to be assigned by being dragged into the scripted component via the **Inspector** as is usual for public variables.

The following class listing provides the necessary extra code to implement this approach:

```
// file: CubeAnimMethodsDropDownList.cs
using UnityEngine;
using System.Collections;

public class CubeAnimMethodsDropDownList : MonoBehaviour {
    public AudioClip waterDropSound1;
    public AudioClip waterDropSound2;

    public enum ClipName {
        WATER_DROP1,
        WATER_DROP2
    }

    public void PlayClipFromList(ClipName clipInt) {
        AudioClip clip = GetClip( clipInt );
        audio.PlayOneShot(clip);
    }

    private AudioClip GetClip(ClipName clipInt) {
        switch( clipInt )
        {
            case ClipName.WATER_DROP1:
                return waterDropSound1;
            case ClipName.WATER_DROP2:
            default:
                return waterDropSound1;
        }
    }
}
```

