5
Controlling Animations

In this chapter, we will cover:

- 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

Introduction

As animations are an essential part of games, playing/controlling those in the best possible way can improve the quality of your final product. In this chapter, we will learn a diversity of techniques to optimize the use of your game's animated models.

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 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: 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 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.

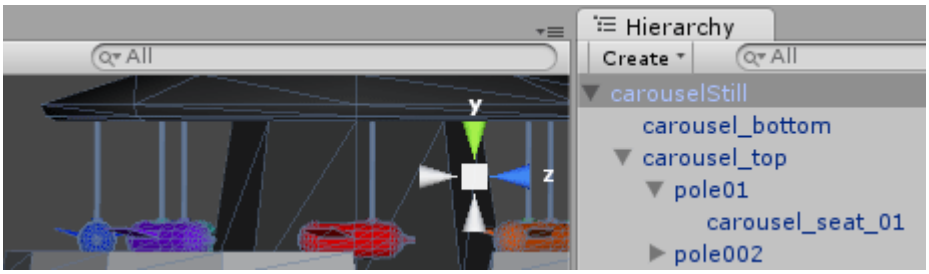
Getting ready

For this recipe, we have included a model named *carouselStill.FBX*, available in the folder 0423_05_01.

How to do it...

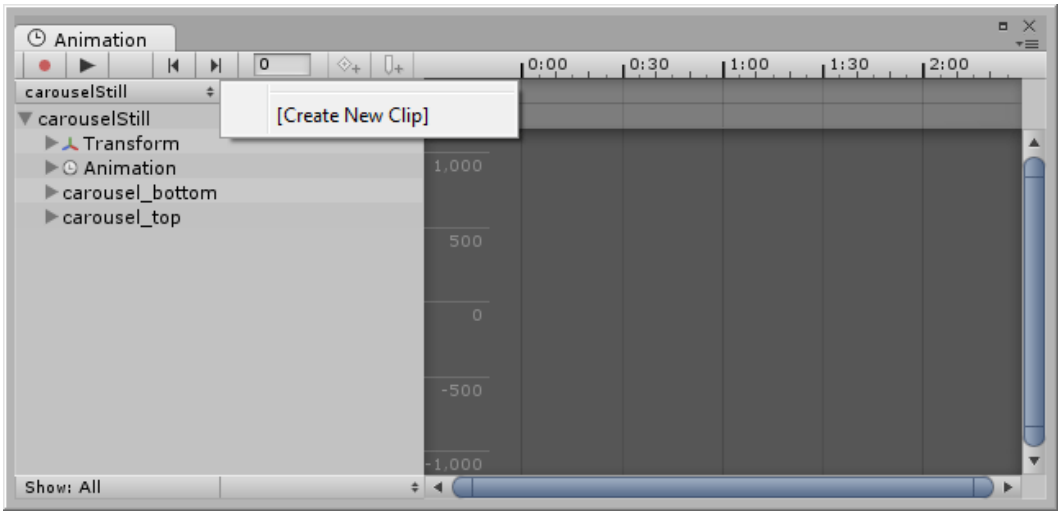
1. Import the FBX model *carouselStill* into your project and add it to your scene.

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 *Animation* component attached to the game object.



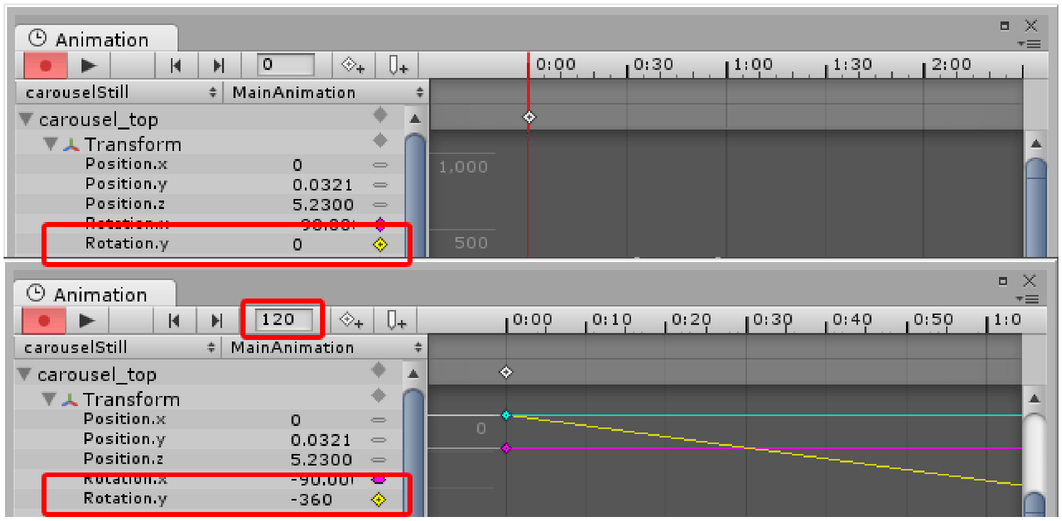
0423_10_01.png

3. With the *carouselStill* game object selected, open the *Animation* view (menu *Windows > Animation*).
4. Click the button on the right side to the one with the object's name to create a New Clip. Save it as *MainAnimation.anim* inside the *Assets* folder.



0423_10_02.png

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

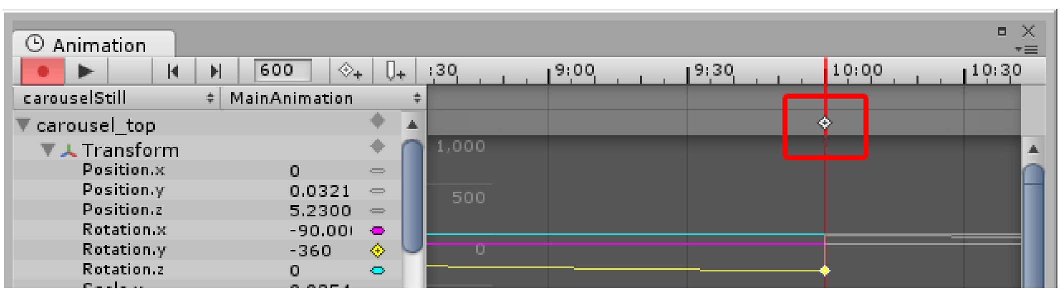


0423_10_03.png

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

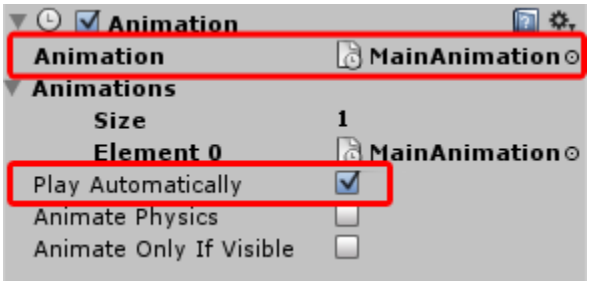
0423_10_04.png

10.Press 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.



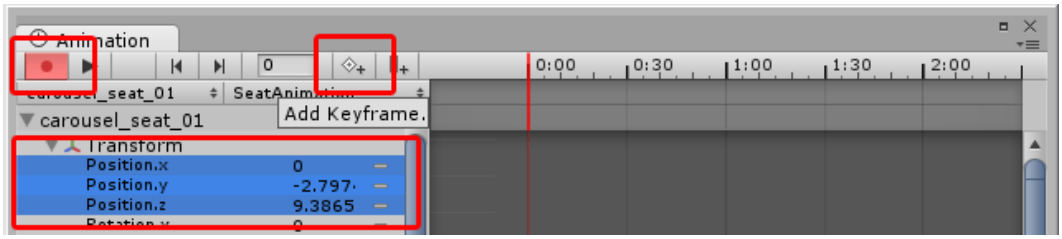
0423_10_05.png

11. Select the *CarouselStill* game object and, in the *Inspector* window, check out its *Animation* component. Make sure the option *Play Automatically* is selected and the chose *MainAnimation* as its default clip.



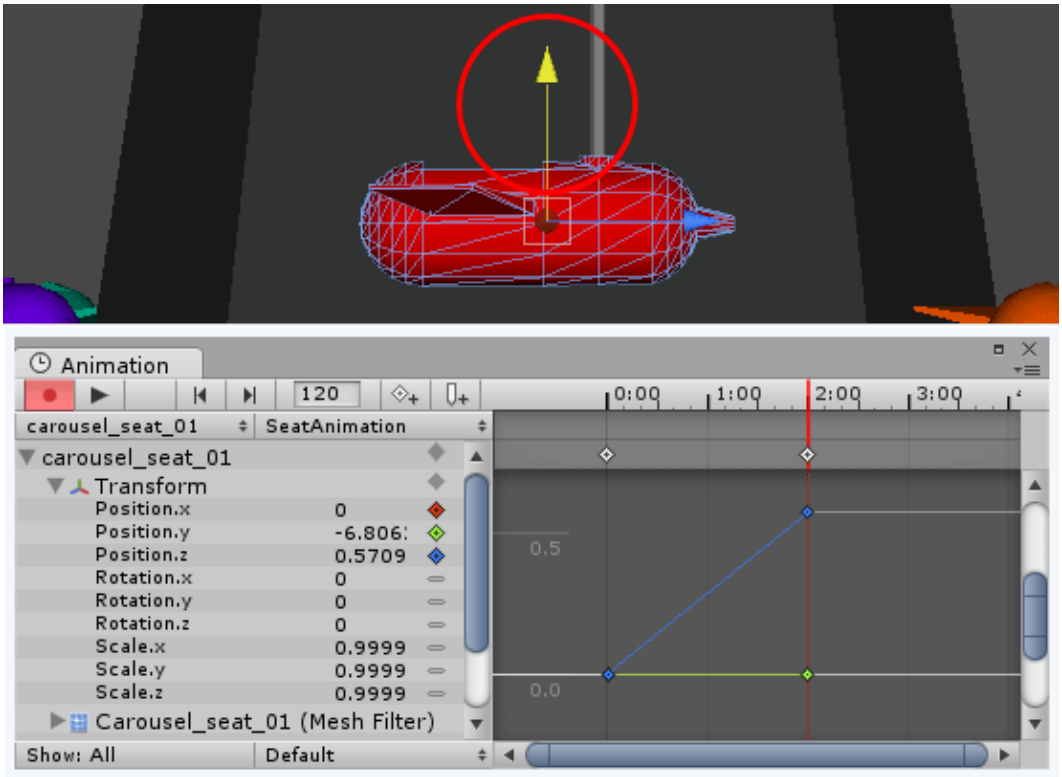
0423_10_06.png

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 (menu *Component > Miscellaneous > Animation*).
13. In the *Animation* view, create a New Clip and save it as *SeatAnimation.anim* inside the *Assets* folder.
14. Click the Record button and, at 0:00, highlight all *Transform/ Position* fields (x,y and z) and click the Add Keyframe button to add a new Keyframe.



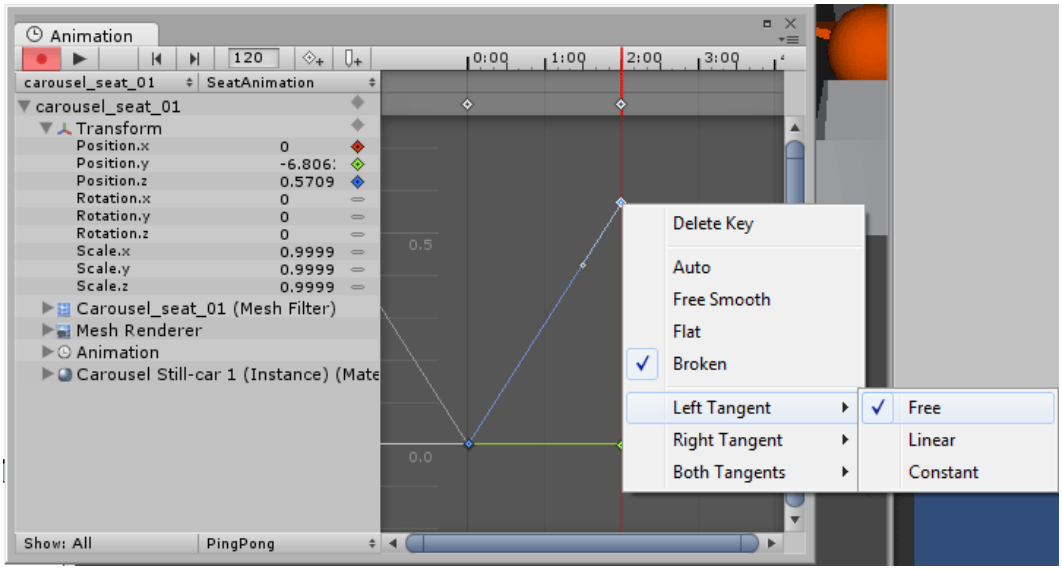
0423_10_07.png

15. Advance the time indicator to 2:00 (or type *120* in the counter field and press *Enter*) 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.



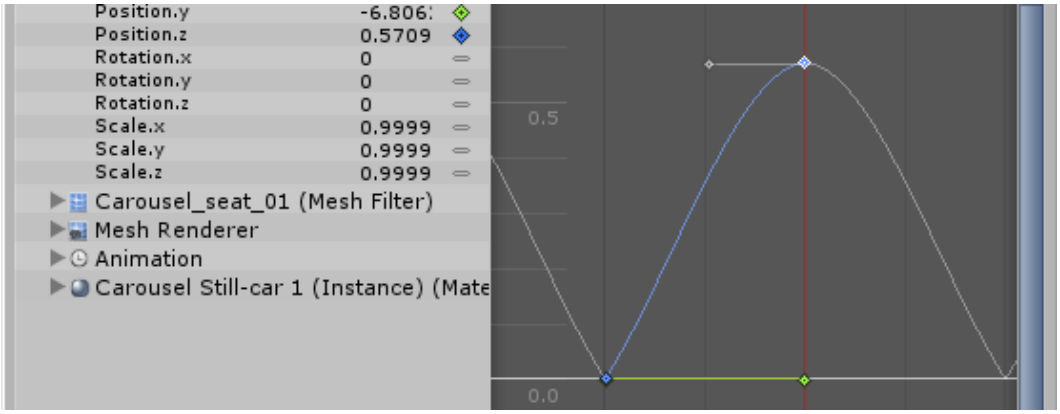
0423_10_08.png

17. In the bottom part of the Animation view, click the appropriate button to change the *Wrap Mode* to *PingPong*.
18. Right-click the keyframe at 2:00 and change its *Left Tangent* to *Free*.



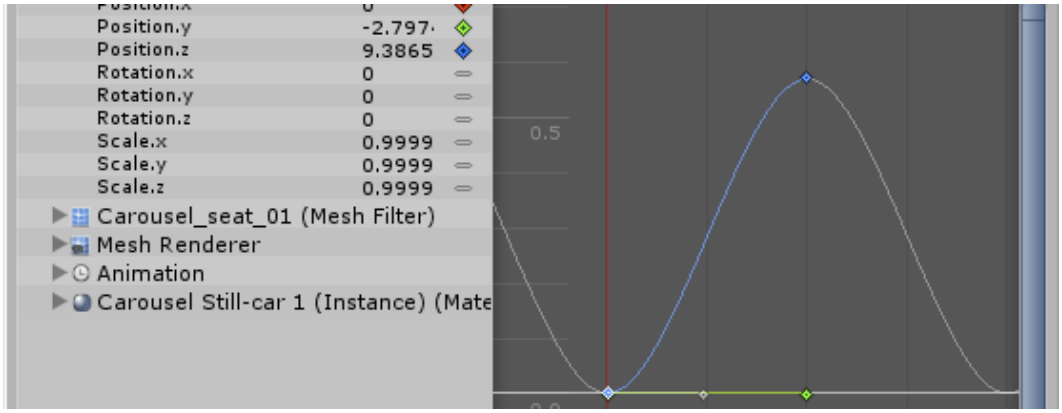
0423_10_09.png

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



0423_10_10.png

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



0423_10_11.png

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*, 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 the *carousel_seat_01* enabled 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

See also

Dividing animation into clips

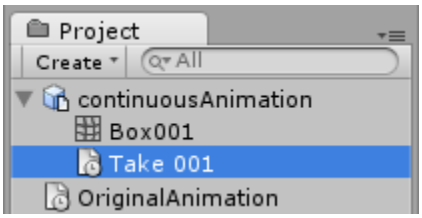
Exporting all animations for a 3d model in a single file is very practical in terms of assets management and art production workflow. However, once it has been imported, those individual animations, now merged into a single file, have to be divided into *Clips*. In this recipe, we will learn how to split all animations imported into separate *Clips*. Better yet, we will pretend we have no access to the artist who created the file and who could tell us how those clips should be divided.

Getting ready

For this recipe, we have prepared an animated 3d model named *continuousAnimation*, available in the folder 0423_05_02.

How to do it...

1. Import *continuousAnimation.FBX* into your project.
2. In the *Project* view, expand the *continuousAnimation* and duplicate its animation clip *Take001*. Rename the copy as *OriginalAnimation*.

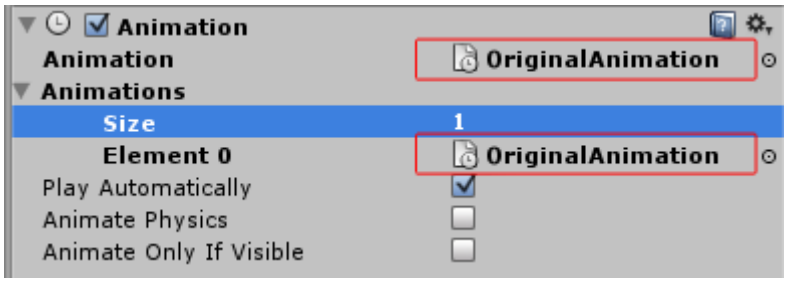


0423_10_12.png

3.

Drag it from the *Project* view into the *Hierarchy* view to place it on the scene. Make sure you can see it from the *Game* view (adjust the camera position, if necessary).

In the *Hierarchy* view, select *continuousAnimation.* Then, in the Inspector view, change its Animation clip to OriginalAnimation (in both *Animation* and *Animations* fields).



0423_10_13.png

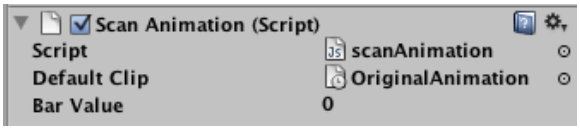
4. From the *Project* view, create a new *C# Script* and name it *ScanAnimation*.

6. Open your script and replace everything with the following code:

```
using UnityEngine;
using System.Collections;

public class ScanAnimation : MonoBehaviour
{
    public AnimationClip defaultClip;
    private float barValue;
    private float speed = 1.0f;
    private string buttonLabel = "PLAY";
    private bool isPlaying = false;
   private AnimationState ans;

    void Start(){
        ans = animation[defaultClip.name];
      ans.wrapMode = WrapMode.Loop;
        ans.speed = 0;
    }

    void OnGUI(){
        GUI.Label(new Rect(10, 10, 120, 20), "Animation
Frame:");
```

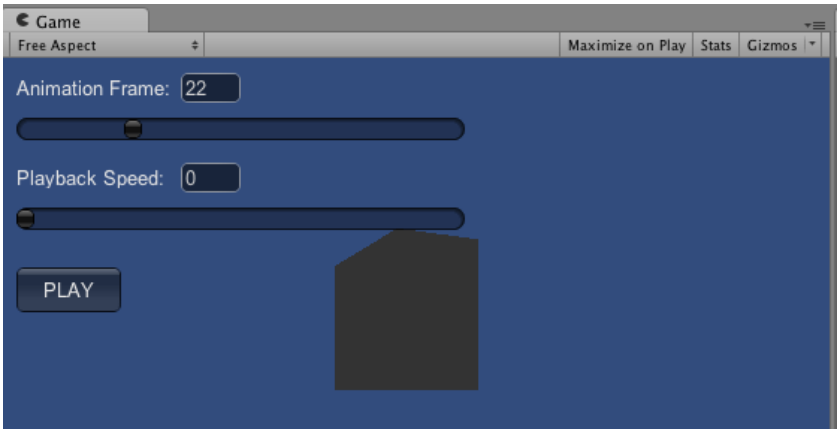
```
        GUI.TextField(new Rect(120, 10, 40, 20),
Mathf.FloorToInt(ans.clip.frameRate * ans.time).ToString(),
25);
        Rect sb = new Rect(10, 40, 300, 30);
      Rect lb = new Rect(10, 70, 120, 20);
      Rect tf = new Rect(120, 70, 40, 20);
      Rect hs = new Rect(10, 100, 300, 30);
      Rect bt = new Rect(10, 140, 70, 30);
      barValue = GUI.HorizontalScrollbar(sb, barValue, 0.0f,
0.0f, ans.clip.frameRate * ans.length);
        GUI.Label(lb, "Playback Speed:");
        GUI.TextField(tf, speed.ToString(), 25);
        speed = GUI.HorizontalScrollbar(hs, speed, 0.0f, 0.0f,
1.0f);
        if (GUI.Button(bt, buttonLabel)){
            if (isPlaying){
                isPlaying = false;
                buttonLabel = "PLAY";
                ans.speed = 0;
            }else{
                isPlaying = true;
                buttonLabel = "PAUSE";
                ans.speed = speed;
            }
      }
    }

    void Update(){
        if (!isPlaying){
            ans.time = barValue / ans.clip.frameRate;
            ans.speed = 0;
        }else{
            barValue = ans.clip.frameRate * ans.time;
            ans.speed = speed;
        }
    }
}
```

7. Save your script and attach it to the *continuousAnimation* game object in the *Hierarchy* view. Then, in the Inspector view, select *ContinuousAnimation* as the *DefaultClip* in the *Scan Animation* component.



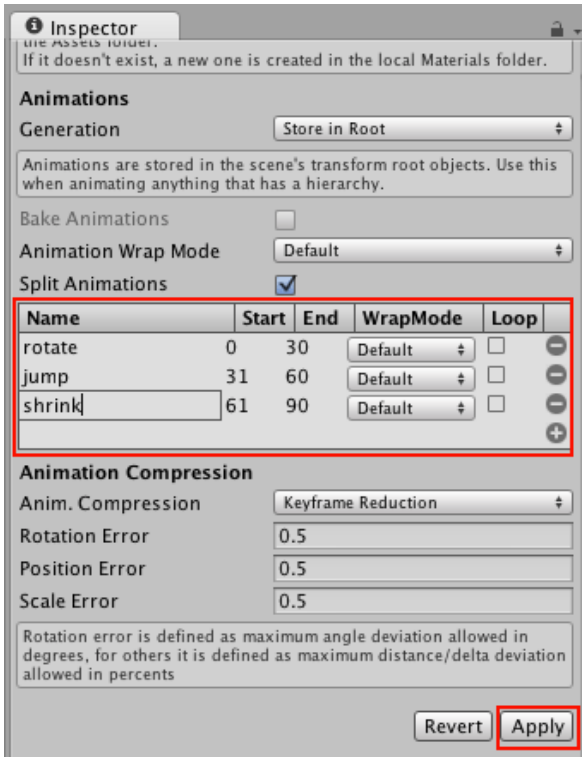
0423_10_14.png

8. Play your scene and test the GUI. You should be able to control the playback speed, read the current frame number and select it from the appropriate slidebars.



0423_10_15.png

9. Observe how the object's animation can be divided into three separate actions. First, from frame 0 to 30 (rotate); second, from 31 to 60 (jump); third, from 61 to 90 (shrink). Stop your scene.
10. In the *Project* view, select the *continuousAnimation* asset. Then, in the *Inspector* view, look for the *Animation* import settings. You will find a table below the option *Split Animations.*
11. Click the round *plus* button three times to add three animation clips. Then, name them *rotate*, *jump* and *shrink*. Finally, assign their Start and End values respectively as *0* and *30*; *31* and *60*; and *61* and *90*.
12. Click the Apply button to confirm the changes you have made.



0423_05_16.png

How it works...

The actual division of the animation into separate clips is made in the *Inspector* view, through the object's *Import Settings,* where we have established the beginning and end of each clip. The *ScanAnimation* script simply controls the animation's playback and speed, making it possible for us to analyze it.

There's more...

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

Importing multiple files as Animation clips

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

Wrap Mode and Loop Frame

From the *Import Settings*, you can also select the animation clip's *Wrap Mode*, which is basically a 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 check the *Loop* tick box, Unity will create an extra frame, identical to the first one, at the end of the animation clip. This should be used in case you have selected the *Loop* wrap mode on an animation where the first and last frame are not identical and its obvious when it has re-started.

See also

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 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 folder 0423_05_03.

How to do it...

1.Import the package *shipScene* into your project.

2.Open the scene named 05_03 and play it. Observe 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*. *shipController.cs*.

4.Open the script. Immediately below the line `private float speed = 0.0f;` , add the following code:

```
public AnimationClip animationClip;

void Start() {

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

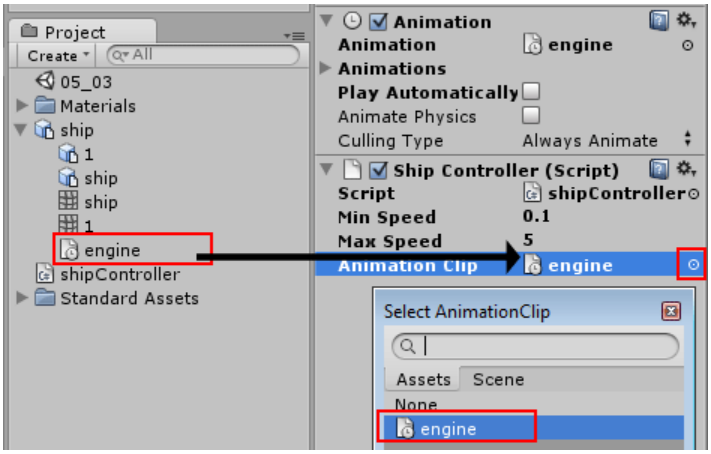
5.Then Immediately below the line `transform.Rotate (0, rotationY, 0);` add:

```
animation[animationClip.name].speed = speed;
```

6.Save and close the script.

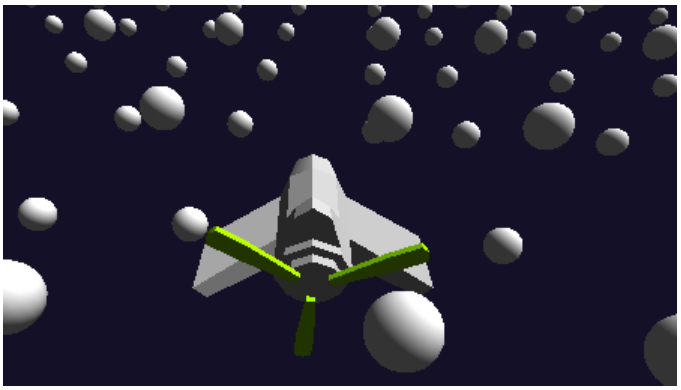
7.In the *Hierarchy* view, select the *ship* game object. Then, in the *Inspector* view, the *Inspector* view, locate the *Ship Controller* component.

8.In the *Ship Controller* component, Change the newly created *Animation Clip* *Animation Clip* parameter to *engine* by either dragging it from the Project view the Project view or selecting it from the list.



0423_05_17.png

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



0423_05_18.png

How it works...

As you have probably noticed, the whole process is very straightforward: all we needed to do was: (a) creating the variable *animationClip* to hold our clip; (b) setting the animation *warp mode* to *Loop* and playing it at *Start()*; and (d) adjusting its speed in the *Update()* function to reflect the ship's Translation speed.

There's more...

Reverse animation

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

See also

Setting up an animated Mixamo character

Since we will be using a character provided by Mixamo in our character-based recipes, it is a good start learning how to set them up properly. 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* containing several assets such as levels, an animated character named *Swat* (modeled by xxxxx and animated by xxxx) and some props. You can find it in the folder 0423_05_codes.

How to do it...

1. Open the project *MixamoProject* and open the level named *05_04* (under the folder *Levels*). This is a level containing a plane, a camera and a directional light.

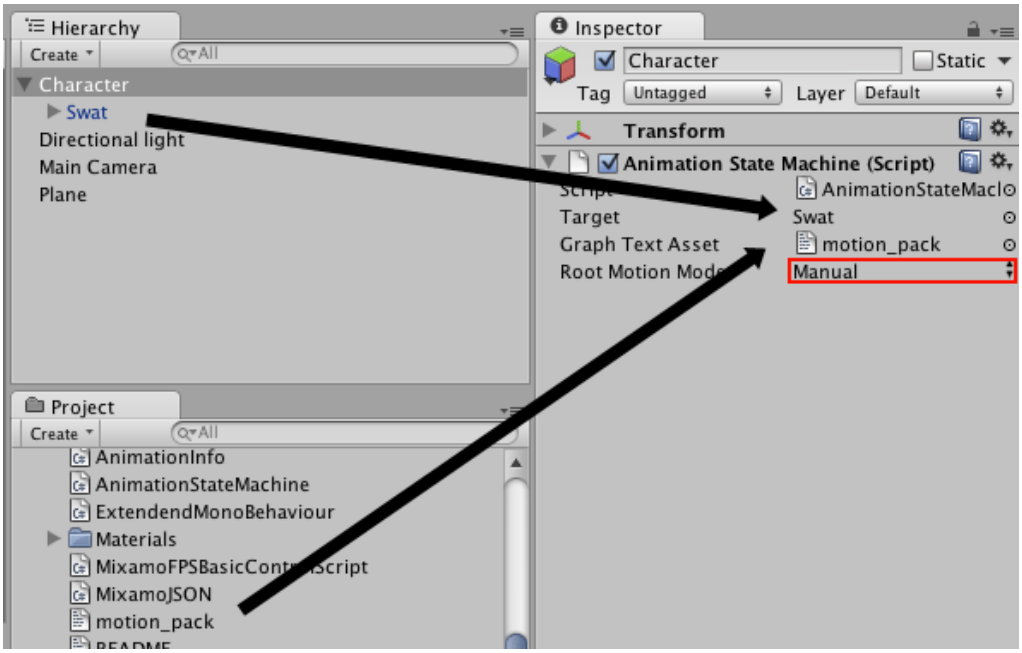
Create an empty *Game Object* (from the menu *GameObject > Create Empty*). Rename it to *Character* and, in the *Inspector* view, set its Position to X: 0, Y: 0, Z: 0.

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.*

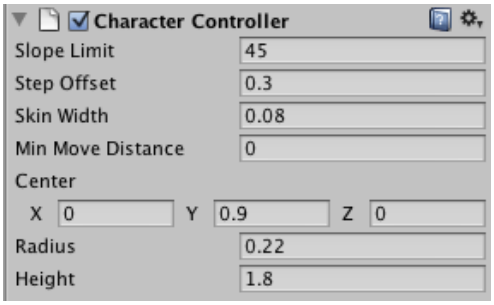
In the *Project* folder, locate the script *AnimatedStateMachine.cs*, located inside the *swatdemo* folder, and attach it to the *Character* game object.

5. Select the *Character* game object and, in the *Inspector* view, access its *Animated State Machine* component. Then, drag the *Swat* from the *Hierarchy* view into the *Target* field. Finally, change its *Root Motion Mode* to *Manual*.



0423_05_19.png

6. Now, add a Character Controller to the Character game object (menu Components > Physics > Character Controller). Then, in the Inspector view, change the fields *Center Y* to 0.9, *Radius* to *0.22* and *Height* to 1.8.



0423_05_20.png

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 control your animated character. You can also jump using the space bar, rotate by pressing *Q* and *E,* reload the weapon by pressing *R* and toss a grenade by grenade by pressing *F*.

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 *Mixamo FPSBasic Control* script designates a control scheme to our character, triggering the animation clips. as previously organized, according to the player's input.

There's more...

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

Using Unity's Third Person Controller

Should you try using Unity`s Third Person Controller instead of Mixamo's solution, we have included a level named 05_04_alternative. The controllers, now attached directly to the *Swat* model, should work! However, the character animations will look strange. That's because Unity's approach uses characters that are animated *in place* (just imagine a hamster in a wheel), while *Mixamo* animated characters actually move in space.

See also

Adding rigid props to animated characters

In case you haven't included a sufficient number of props to your character when modelling 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 two Project folders: The one named *MixamoProject* is for those using Unity's old Animation system. The other, named *MixamoMecanim*, is for those who want to use the Unity's new *Mecanim* system. They both contain several assets such as levels, animated character and props. You can find it in the folder 0423_05_codes.

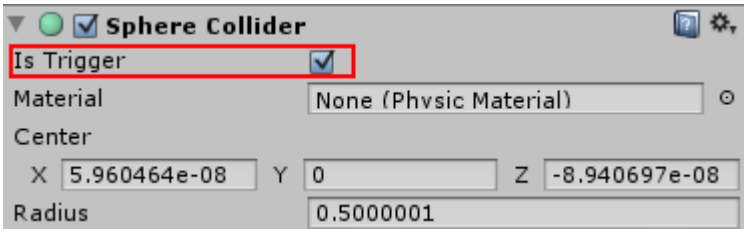
How to do it...

1. Open the project *MixamoProject* or *MixamoMecanim* and open the level named *05_05* (under the folder *Levels*).
2. You should see an Mixamo's animated Swat 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;
```

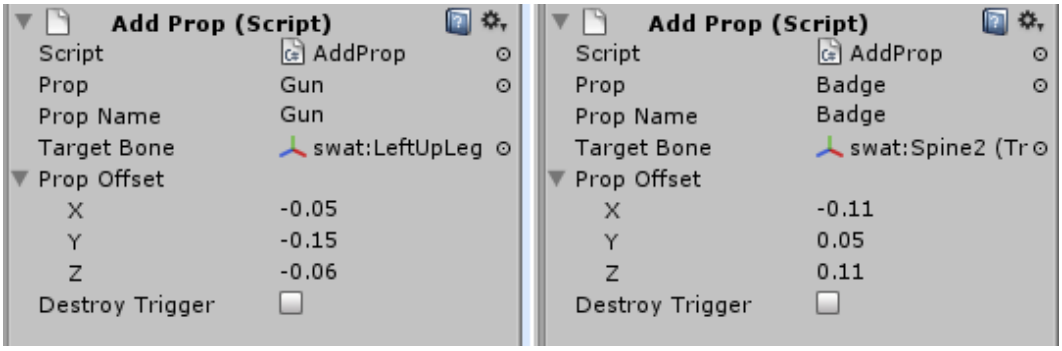
```
            newprop.transform.localPosition += propOffset;
            if(destroyTrigger)
               Destroy(gameObject);
            }
        }
}
}
```

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), check the option *Is Trigger*.



0423_05_21.png

8. Select the *Sphere Blue* 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 field *Prop Name*, type in *Gun*.
11. In the field *Target Bone*, select the Transform *swat:LeftUpLeg* from the list (or drag it from the *Hierarchy* view: it is a child of the *Character* game object).
12. Expand the field *Prop Offset* and type the values of X, Y and Z as *-0.05, -0.15 and -0.06* respectively. Finally, unselect the option *Destroy Trigger*.
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, Z = 0.11. Also, uncheck *Destroy Trigger*.



0423_05_22.png

14. Play the scene. Using the *WASD* keyboard control scheme, direct the character to the 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 side of the character's chest).

0423_05_23.png

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* 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...

Removing Props

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

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

See also

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.

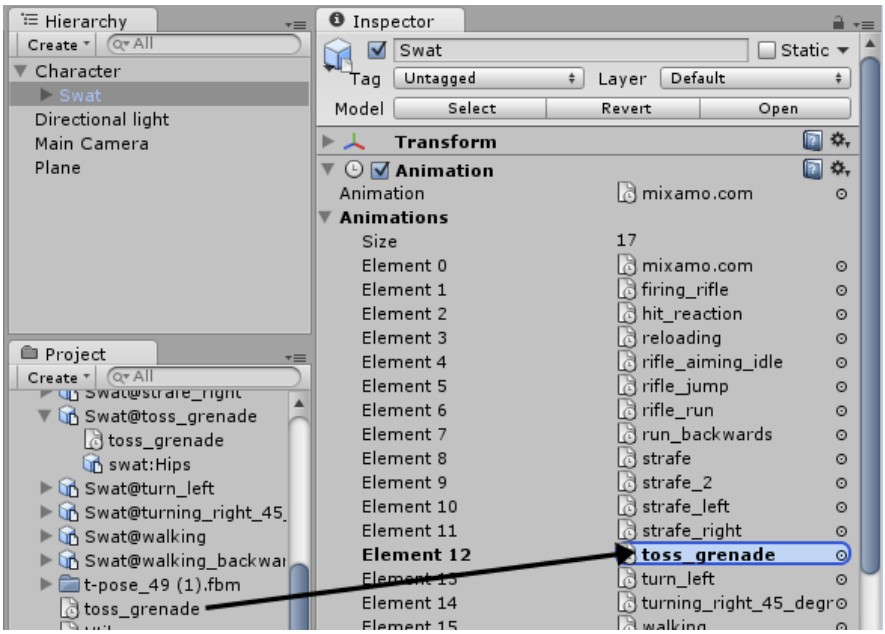
Getting ready

For this recipe, we have prepared two Project folders: The one named *MixamoProject* is for those using Unity's old Animation system. The other, named *MixamoMecanim*, is for those who want to use the Unity's new *Mecanim* system. They both contain several assets such as levels, animated character and props. You can find it in the folder 0423_05_codes.

How to do it...

1. Open the project *MixamoProject* and open the level named *05_06* (under the folder *Levels*).
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.



0423_05_24.png

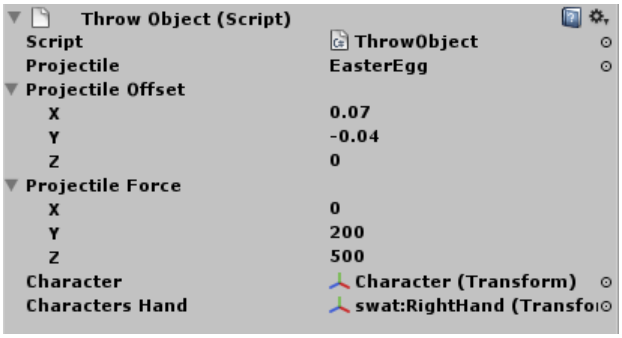
5. In the *Project* view, create a new *C# Script* named *ThrowObject.cs*.

6. Open the script and add the following code:

```
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>();
      Physics.IgnoreCollision(projectile.collider, collider);
      projectile.rigidbody.AddRelativeForce(projectileForce);
   }
}
```

7. Save and close the script.
8. Attach the script *ThrowObject.cs* to the game object named *Swat* (child of the game object named *Character*).
9. 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.
10. From the *Hierarchy* view, drag the *Character* game object into the *Character* field of the component. Also, in the field *Characters Hand*, select the transform *swat:RightHand*.

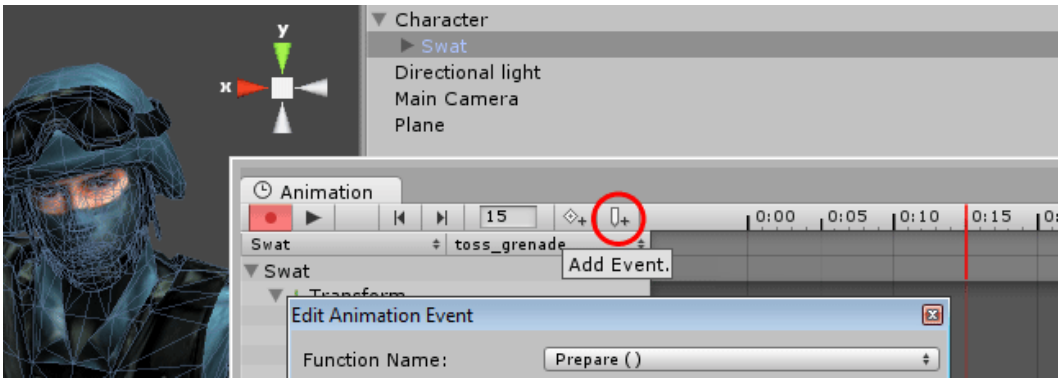
11. As we are still on the *Throw Object* component, fill in the variables as indicated: *Projectile Offset:* X = -0.07, Y = -0.04, Z = 0; *Projectile Force:* X = -0, Y = 200, Z = 500.



0423_05_25.png

12. Select the *Swat* game object and access the *Animation* view (menu option *Window > Animation*). Now, select toss_grenade from the animation list.

13. Drag the time indicator to 0:15 and click the *Add Event* button. From the dialog box that pops up, select the function *Prepare()*.



0423_05_26.png

14. Now, drag the time indicator to 1:25 and click the *Add Event* button. This time, select the function *Throw()*.

15. 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 script *ThrowObject*. This function instantiates a prefab, now named *projectile*, into the character's hand, also making it respect the character's hierarchy. Also, 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 thrown by the character.

There's more...

Scripting Animation Events

Alternatively to 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

See also

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 in a landmine object. We will also use the opportunity to reset the character's position and animations a number of seconds after that event.

Getting ready

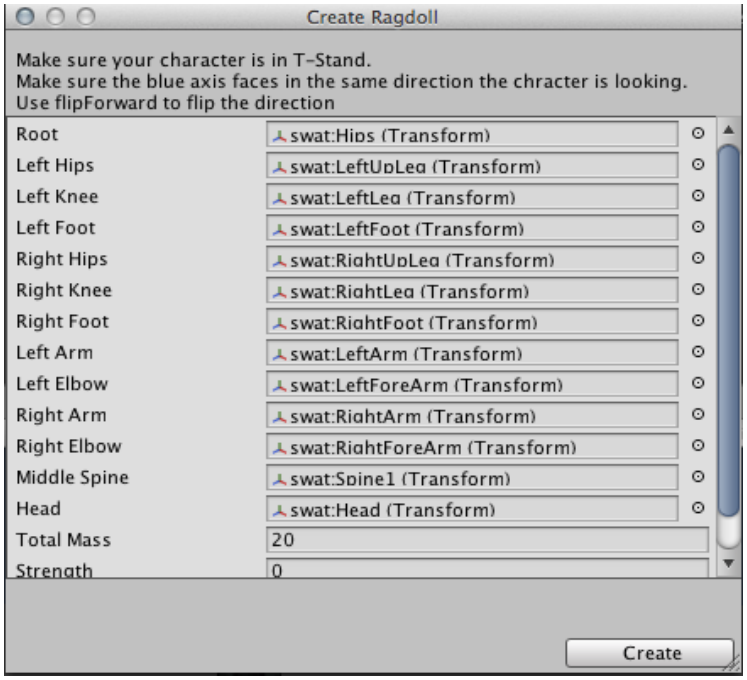
For this recipe, we have prepared two Project folders: The one named *MixamoProject* is for those using Unity's old Animation system. The other, named *MixamoMecanim*, is for those who want to use the Unity's new *Mecanim* system. They both contains several

assets such as levels, animated character and props. You can find it in the folder 0423_05_codes.

How to do it...

1. Open the project *MixamoProject* or *MixamoMecanim* and open the level named *05_07* (under the folder *Levels*).
2. You should see an Mixamo's animated Swat soldier and two cylinders: the *Landmine* and the *Spawnpoint*.
3. First, let`s set up our Ragdoll. Access the menu *GameObject>Create Other>Ragdoll...* The Ragdoll wizard should pop-up.
4. Assign the transforms as follow:

- 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.



0423_05_27.png

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 () {
      if(wasHit){
         if(Time.time >= hitTime + 5.0f)
            DeactivateRagdoll();
         }
```

```
      }
   public void ActivateRagdoll(){

      // A - If using Mecanim, delete or comment:
      this.GetComponent<AnimationStateMachine>().enabled =
false;
      this.GetComponent<MixamoFPSBasicControlScript>().enabled
= false;
      foreach(Animation anim in
GetComponentsInChildren<Animation>()){
         anim.GetComponent<Animation>().enabled = false;
      }
    // A - stop deleting
   // B – If using Mixamo's controller, delete or comment:
      this.GetComponent<BasicController>().enabled = false;
      this.GetComponent<Animator>().enabled = false;
   // B - stop deleting

      foreach(Rigidbody bone in
GetComponentsInChildren<Rigidbody>()){
         bone.isKinematic = false;
            bone.detectCollisions = true;
      }

      wasHit = true;
      hitTime = Time.time;
   }

   public void DeactivateRagdoll(){
    // C - If using Mecanim, delete or comment:
      this.GetComponent<AnimationStateMachine>().enabled =
true;
      this.GetComponent<MixamoFPSBasicControlScript>().enabled
= true;

      foreach(Animation anim in
GetComponentsInChildren<Animation>()){
         anim.GetComponent<Animation>().enabled = true;
      }
    // C - stop deleting
    // D - If using Mixamo's controller, delete or comment:
      this.GetComponent<BasicController>().enabled = true;
      this.GetComponent<Animator>().enabled = true;

    // D - stop deleting
      foreach(Rigidbody bone in
GetComponentsInChildren<Rigidbody>()){
         bone.isKinematic = true;
         bone.detectCollisions = false;
      }
```

```
        transform.position =
GameObject.Find("Spawnpoint").transform.position;
        transform.rotation =
GameObject.Find("Spawnpoint").transform.rotation;
        wasHit = false;
    }
}
```

7. Comment or delete the appropriate lines as indicated along the code.
8. Save and close the script.
9. Attach the script *RagdollCharacter.cs* to the *Character* Game Object.
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 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 re-spawn our character in the appropriate place.

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

There's more...

Re-spawing

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

See also

Rotating the character's torso to aim

When playing a third person character, you might want him to aim his weapon at some 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 two Project folders: The one named *MixamoProject* is for those using Unity's old Animation system. The other, named *MixamoMecanim*, is for

those who want to use the Unity's new *Mecanim* system. They both contains several assets such as levels, animated character and props. You can find it in the folder 0423_05_codes.

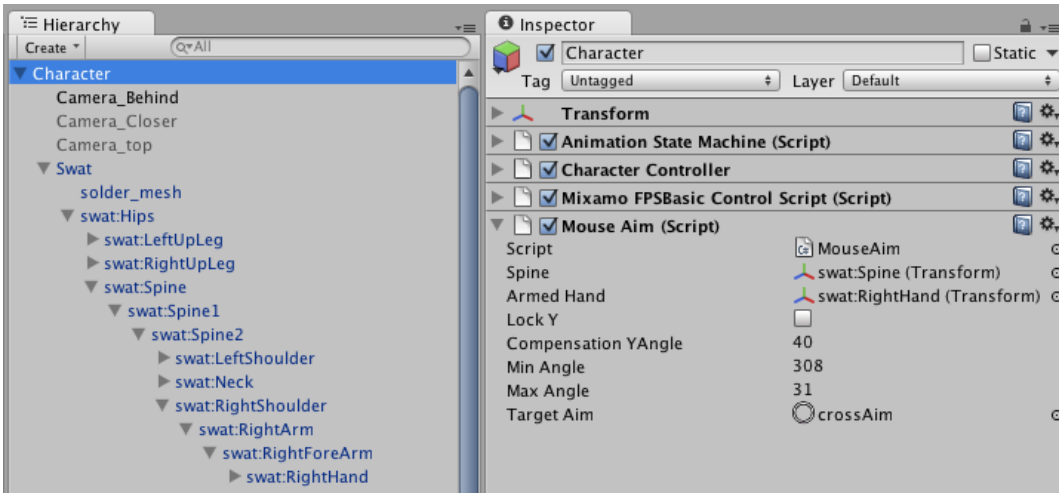
How to do it...

1. Open the project *MixamoProject* or *MixamoMecanim* and open the level named *05_08* (under the folder *Levels*).
2. You should see Mixamo's animated Swat 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;

      // If using Mecanim, delete or comment the line below:
   public void Aim(){
      // If using Mixamo's controller, delete or comment the
line below:
   public void LateUpdate(){


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



0423_05_28.png

9. If using Mixamo's controller (instead of Mecanim), locate and open the script named *AnimationStateMachine.cs.*
10. Locate the *LateUpdate()* function and add the following lines just before it before it closes:

```
if(GetComponent<MouseAim>())
      GetComponent<MouseAim>().Aim();
```

11. 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.

0423_05_29.png

How it works...

Our *Aim()* function starts by converting our bi-dimensional mouse cursor screen's coordinates to three-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,* otherwise causing distortions to the character model.

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

Also, we have included a *Compensation YAngle* variable that makes 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...

Shooting

In case you want to implement shooting, we have provided you 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 command 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.

See also

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:

- Cross-fading: Creating a smooth transition from an animated state to the next, instead of abruptly stopping an animation to starting 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.

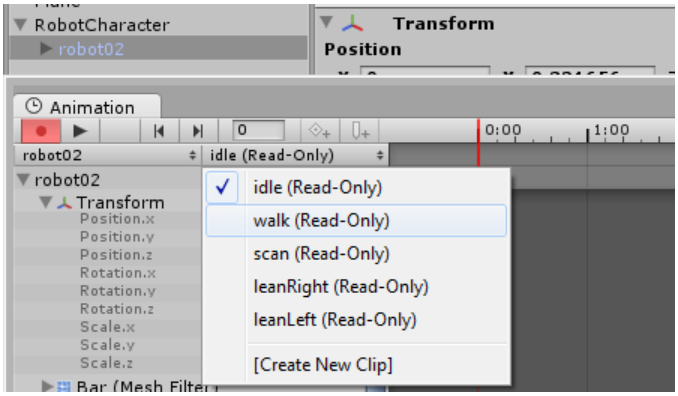
Additive blending: Overlaying two different animations. In our recipe, we will combine animations of the character leaning towards his sides with either *idle* or *walk* animation, depending on the character's current state.

Getting ready

For this recipe, we have prepared a prefab named *combiningStates* including level that contains a robot character with basic input controls. You can find it in the folder 0423_05_09.

How to do it...

1. Import the package *combiningStates* into your project.
2. Open the level named *05_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 (menu *Window > Animation*). You can now preview each animation clip we are about to use.



0423_05_30.png

4. In the *Project* view, open the c# script named *RobotControl.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();
    }
```

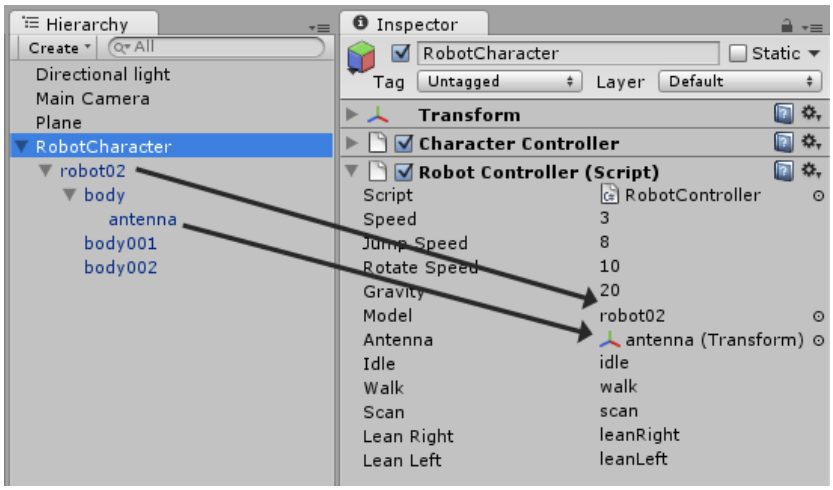
6. Then, include the following code inside the *Update()*function , just before it closes:

```
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.

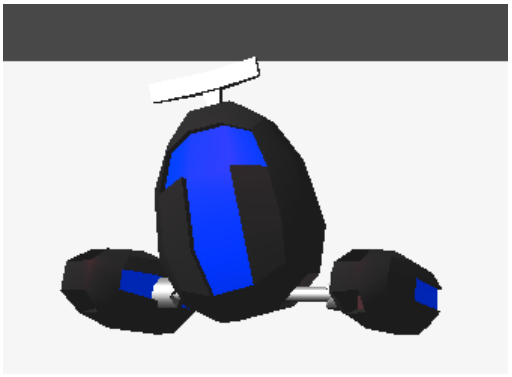
8. Select the Robot*Character* game object and, in the *Inspector* view, access the *Robot Controller* component. As most of our variables are already filled in, we just need to assign the *Model* and *Antenna*.

9. In the Model field, select the *robot02* game object (child of *RobotCharacter*). In the Antenna field, select the *antenna* transform (child of *RobotCharacter > robot02 > body > antenna*).



0423_05_31.png

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.



0423_05_32.png

How it works...

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

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 *left* or *right*) 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/AnimationScripting.html`

See also