7
Controlling 3D Animations

In this chapter, we will cover:

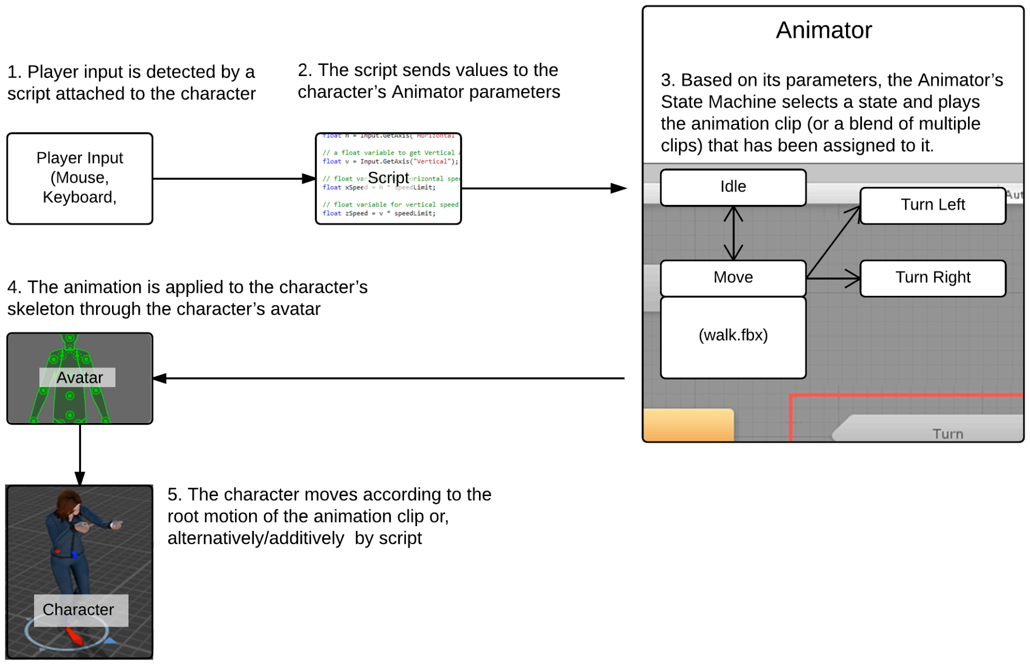
- Configuring a character's Avatar and Idle animation
- Moving your character with Root Motion and Blend Trees
- Mixing animations with Layers and Masks
- Organizing States into Sub-State Machines
- Transforming the Character Controller via script
- Adding rigid props to animated characters
- Using Animation Events to throw an object
- Applying Ragdoll physics to a character
- Rotating the character's torso to aim

Introduction

The **Mecanim** animation system has revolutionized how characters are animated and controlled within Unity. In this chapter, we will learn how to take advantage of its flexibility, power, and friendly, highly visual, interface.

The big picture

Controlling a playable character with the Mecanim System might look like a complex task, but it is actually very straightforward.


Animator
1. Player input is detected by a script attached to the character
2. The script sends values to the character's Animator parameters
3. Based on its parameters, the Animator's State Machine selects a state and plays the animation clip (or a blend of multiple clips) that has been assigned to it.
Player Input (Mouse, Keyboard,
Script
Idle
Turn Left
Move
Turn Right
(walk.fbx)
Turn
4. The animation is applied to the character's skeleton through the character's avatar
Avatar
Character
5. The character moves according to the root motion of the animation clip or, alternatively/additively by script

Insert image 1362OT_07_00.png

Hopefully, by the end of the chapter, you will have gained at least a basic understanding of the Mecanim system. For a more complete overview of the subject, consider taking a look at Jamie Dean's *Unity Character Animation with Mecanim*, also published by Packt Publishing.

An additional note: All recipes will make use of **Mixamo** motion packs. Mixamo is a complete solution for character production, rigging, and animation. In fact, the character in use was designed with Mixamo's character creation software **Fuse**, and rigged with the Mixamo **Auto-rigger**. You can find out more about Mixamo and their products at Unity's Asset Store (`u3d.as/publisher/mixamo/1GB`) or their website: `www.mixamo.com`.

Please note that, although Mixamo offers Mecanim-ready characters and animation clips, we will use, for the recipes in this chapter, unprepared animation clips. The reason is to make you more confident when dealing with assets obtained by other methods and sources.

Configuring a character's Avatar and Idle animation

A feature that makes Mecanim so flexible and powerful is the ability of quickly reassigning animation clips from one character to another. This is made possible through the use of **Avatars**, which are basically a layer between your character's original rig and Unity's **Animator** system.

In this recipe, we will learn how to configure an Avatar skeleton on a rigged character.

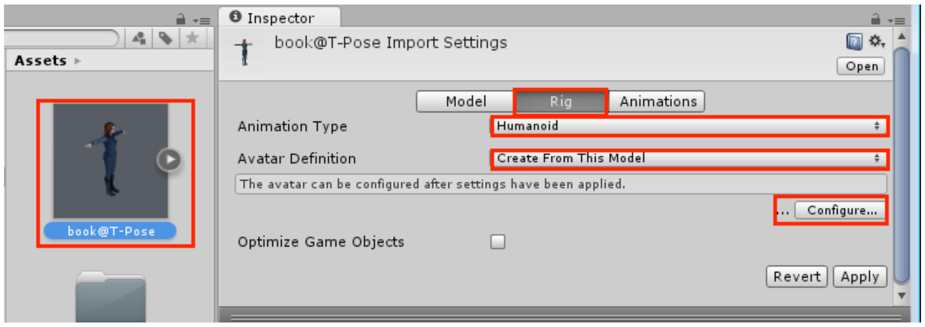
Getting ready

For this recipe, you will need the files `book@T-Pose.fbx` and `Swat@rifle_aiming_idle.fbx`, contained inside the `charater_and_clips` folder within `1362_code`.

How to do it...

To configure an Avatar skeleton, follow these steps:

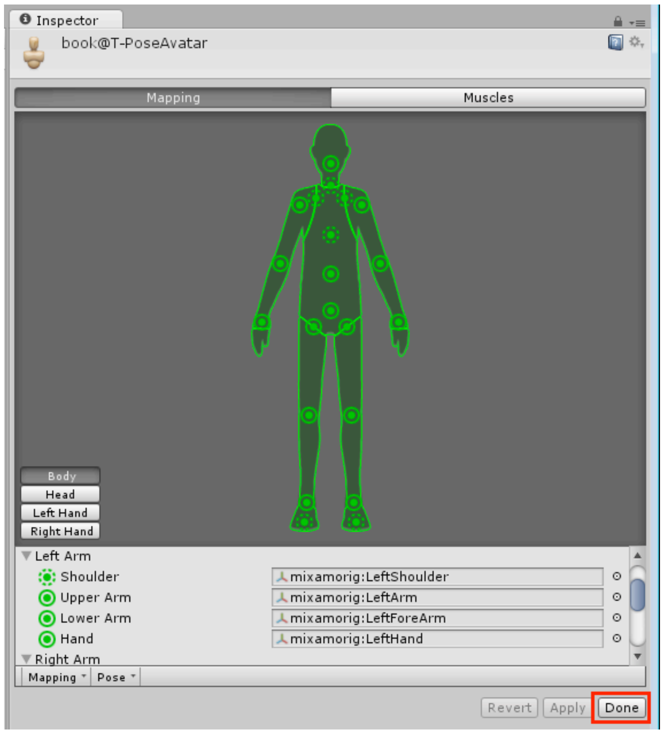
1. Import the files `book@T-Pose.fbx` and `Swat@rifle_aiming_idle.fbx` into your project.
2. Select, from the **Project** view, the `book@T-Pose` model.
3. In the **Inspector** view, under **book@T-Pose Import Settings**, activate the **Rig** section. Change **Animation Type** to **Humanoid**. Then, leave **Avatar Definition** as **Create From this Model**. Finally, click the **Configure…** button.



Animation Type: Humanoid
Avatar Definition: Create From This Model
The avatar can be configured after settings have been applied.
Optimize Game Objects

Insert image 1362OT_07_01.png

4. The **Inspector** view will show the newly created *Avatar*. Observe how Unity correctly mapped the bones of our character into its structure, assigning, for instance, the bone **mixamoRig:LeftForeArm** as the Avatar's **Lower Arm**. We could, of course, reassign bones if needed. For now, just click the **Done** button to close the view.



Left Arm

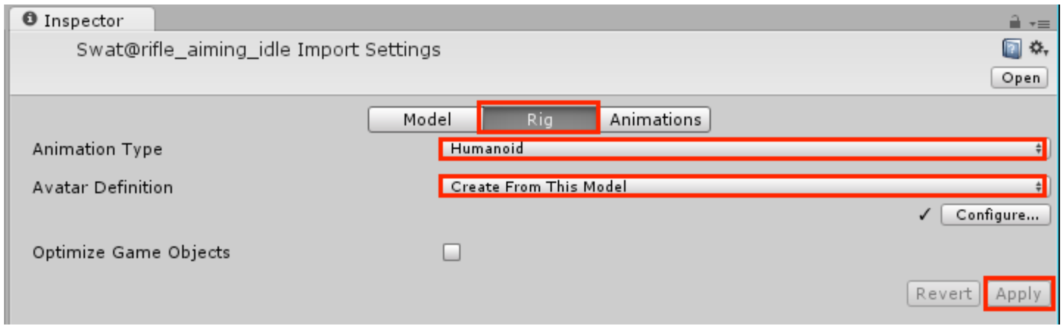
Shoulder	mixamorig:LeftShoulder
Upper Arm	mixamorig:LeftArm
Lower Arm	mixamorig:LeftForeArm
Hand	mixamorig:LeftHand

Right Arm

Mapping Pose

Insert image 1362OT_07_02.png

5. Now that we have our **Avatar** ready, let's configure our animation for the **Idle** state. From the **Project** view, select the file **Swat@rifle_aiming_idle**.
6. Activate the **Rig** section, Change **Animation Type** to **Humanoid** and **Avatar Definition** to **Create From This Model**. Confirm by clicking **Apply**.

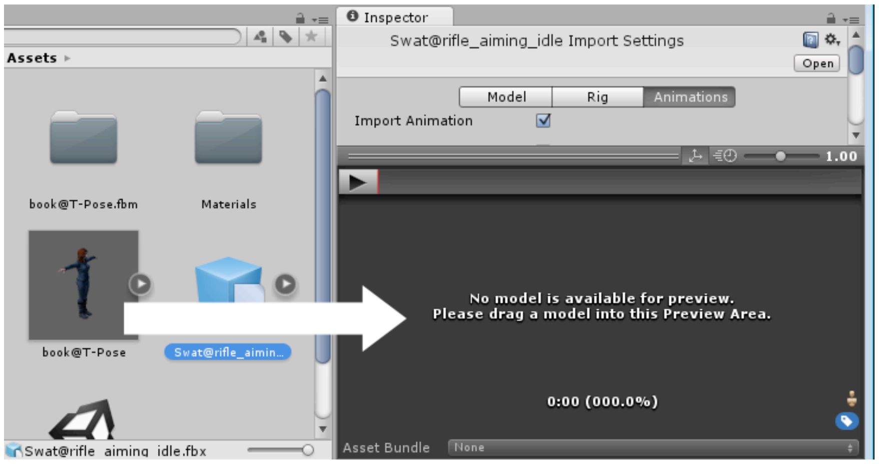


Swat@rifle_aiming_idle Import Settings

Animation Type	Humanoid
Avatar Definition	Create From This Model
Optimize Game Objects	☐

Insert image 1362OT_07_03.png

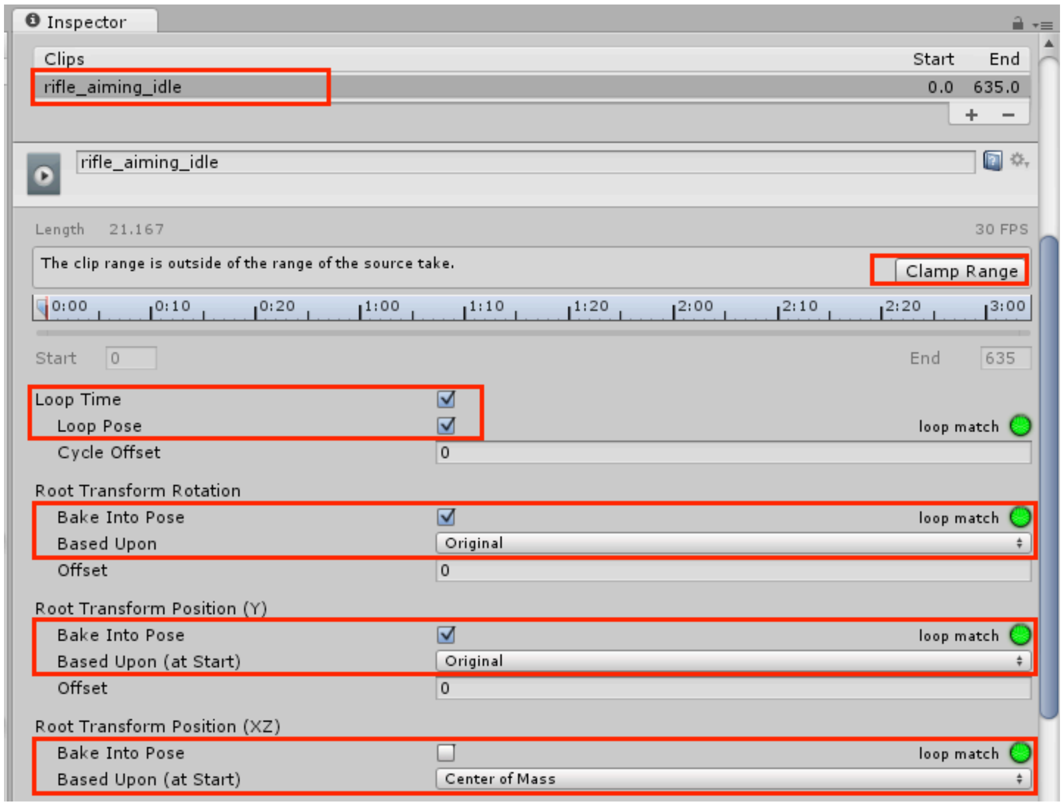
7. Activate the **Animations** section (to the right of the **Rig**). Select the clip **rifle_aiming_idle** (from the **Clips** list). The **Preview** area (on the bottom of the Inspector) should display the message **No model is available for preview. Please, drag a model into this Preview area**. Drag the **book@T-Pose** into the Preview area to correct this.



No model is available for preview.
Please drag a model into this Preview Area.

Insert image 1362OT_07_04.png

8. With **rifle_aiming_idle** selected from the **Clips** list, check the options **Loop Time** and **Loop Pose**. Also, click the button **Clamp Range** to adjust the timeline to the actual time of the animation clip. Then, under **Root Transform Rotation**, check **Bake into Pose,** select **Baked upon | Original**; Under **Root Transform Position (Y)**, check **Bake into Pose,** select **Baked upon (at Start) | Original**; Under **Root Transform Position (XZ)**, leave **Bake into Pose** unchecked, select **Baked upon (at Start) | Center of Mass**. Finally, click **Apply** to confirm changes.


Inspector

Clips	Start	End
rifle_aiming_idle	0.0	635.0

rifle_aiming_idle

Length 21.167 30 FPS

The clip range is outside of the range of the source take. Clamp Range

Start 0 End 635

Loop Time ✓
Loop Pose ✓ loop match
Cycle Offset 0

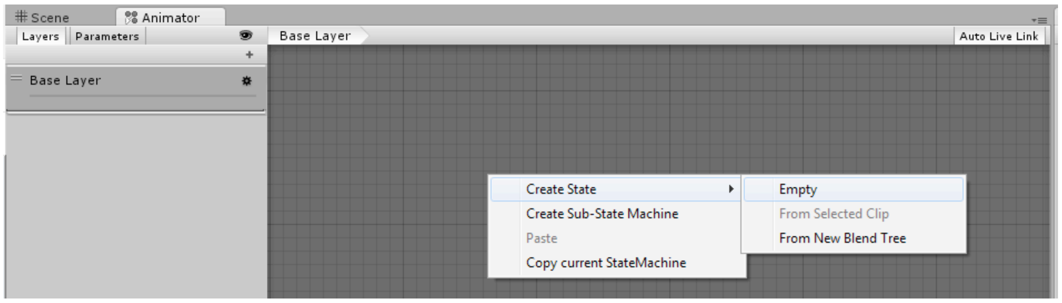
Root Transform Rotation
Bake Into Pose ✓ loop match
Based Upon Original
Offset 0

Root Transform Position (Y)
Bake Into Pose ✓ loop match
Based Upon (at Start) Original
Offset 0

Root Transform Position (XZ)
Bake Into Pose ☐ loop match
Based Upon (at Start) Center of Mass

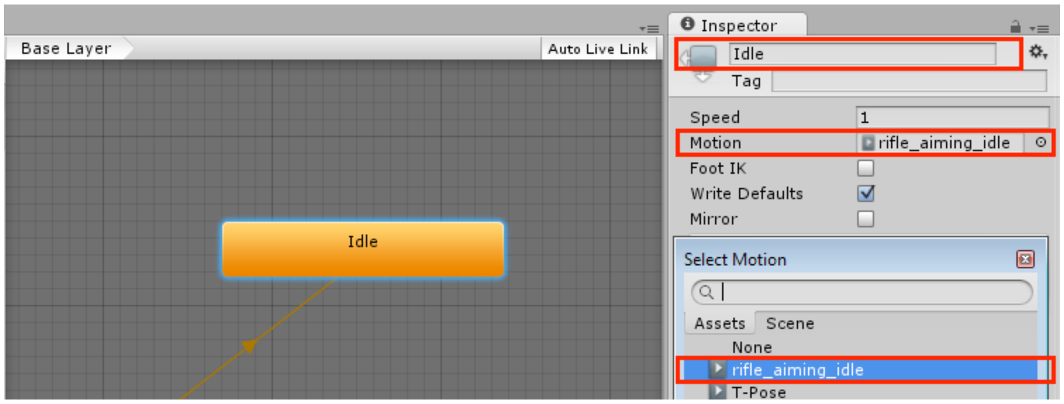
Insert image 1362OT_07_05.png

9. In order to access animation clips and play them, we need to create a controller. Do that by clicking the **Create** button from the **Project** view and selecting the option **Animator Controller**. Name it `MainCharacter`.
10. Double-click the **Animator Controller** to open the **Animator** view.
11. From the **Animator** view, right-click the grid to open a context menu. Then, select the option **Create State | Empty.** A new box named **New State** will appear. It should be orange, indicating it is the default state.



Insert image 1362OT_07_06.png

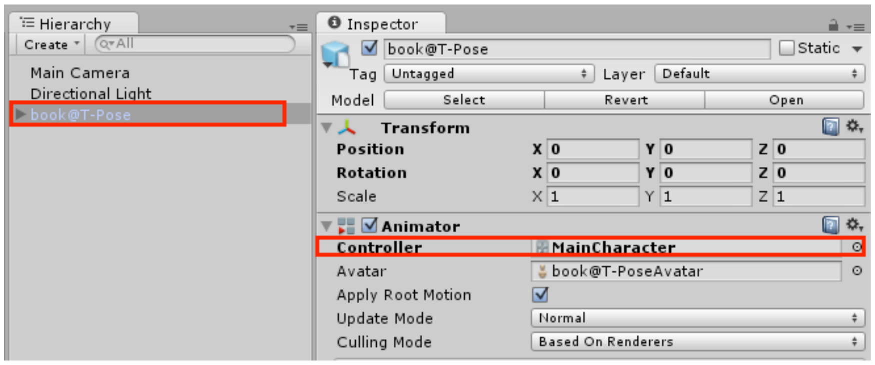
12. Select the **New State** and, in the **Inspector** view, change its name to `Idle`. Also, in the **Motion** field, choose **rifle_aiming_idle** by either selecting it from the list or dragging it from the **Project** view.



Idle
Tag
Speed 1
Motion rifle_aiming_idle
Foot IK
Write Defaults ✓
Mirror

Insert image 1362OT_07_07.png

13. Drag the **book@T-Pose** model from the **Project** view into the **Hierarchy**, placing it on the scene.
14. Select **book@T-Pose** from the **Hierarchy** view and observe its **Animator** component, in the **Inspector** view. Then, assign the newly created **MainCharacter controller** to its **Controller** field.


Hierarchy
Create
Main Camera
Directional Light
book@T-Pose

Inspector
book@T-Pose
Static
Tag Untagged
Layer Default
Model Select Revert Open
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Animator
Controller MainCharacter
Avatar book@T-PoseAvatar
Apply Root Motion ✓
Update Mode Normal
Culling Mode Based On Renderers

Insert image 1362OT_07_08.png

15. Play your scene to see the character correctly animated.

How it works...

Preparing our character for animation took many steps. First, we have created its **Avatar** based on the character model's original bone structure. Then we set up the **animation clip** (which, as the character mesh, is stored in an `.fbx` file), using its own Avatar. After that, we've adjusted the animation clip, clamping its size and making it a loop. We have also baked its Root Transform Rotation to obey the original file's orientation. Finally, an **Animator Controller** was created, and the edited animation clip was made into its default **Animation state**.

The concept of the **Avatar** is what makes **Mecanim** so flexible. Once you have a **Controller**, you can apply it to other humanoid characters, as long as they have an Avatar body mask. If you want to try it yourself, import the `mascot.fbx`, also available inside the `charater_and_clips` folder, apply steps 3 and 4 into that character, place it on the scene and apply the **MainCharacter** as its **Controller** in the **Animator** component. Then, play the scene to see the mascot playing the **rifle_aiming_idle** animation clip.

There's more...

To read more information about the Animator Controller, check Unity's documentation at: `docs.unity3d.com/Documentation/Components/class-AnimatorController.html`.

Moving your character with Root Motion and Blend Trees

Mecanim animation system is capable of applying root motion to characters. In other words, it *actually* moves the character according to the animation clip, as opposed to arbitrarily translating the character model while playing an in-place animation cycle. That makes most **Mixamo** animation clips perfect for use with **Mecanim**.

Another feature of the animation system is **Blend Trees**, which can blend animation clips smoothly and easily. In this recipe, we will take advantage of these features to make our character walk / run forward and backwards and also strafe right and left at different speeds.

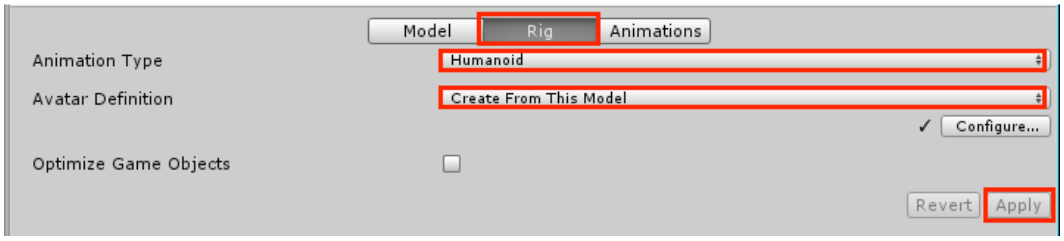
Getting ready

For this recipe, we have prepared a Unity package named `Character_02`, containing a character featuring a basic Animator Controller. The package can be found inside the `1362_07_02` folder, along with `.fbx` files for the necessary animation clips.

How to do it...

To apply root motion to your character using blend trees, follow these steps:

1. Import `Character_02.unityPackage` into a new project. Also, import `Swat@rifle_run, Swat@run_backwards, Swat@strafe, Swat@strafe_2, Swat@strafe_left, Swat@strafe_right, Swat@walking, and Swat@walking_backwards` fbx files.
2. We need to configure our animation clips. From the **Project** view, select **Swat@rifle_run**.
3. Activate the **Rig** section. Change **Animation Type** to **Humanoid** and **Avatar Definition** to **Create From this Model**. Confirm by clicking **Apply**.



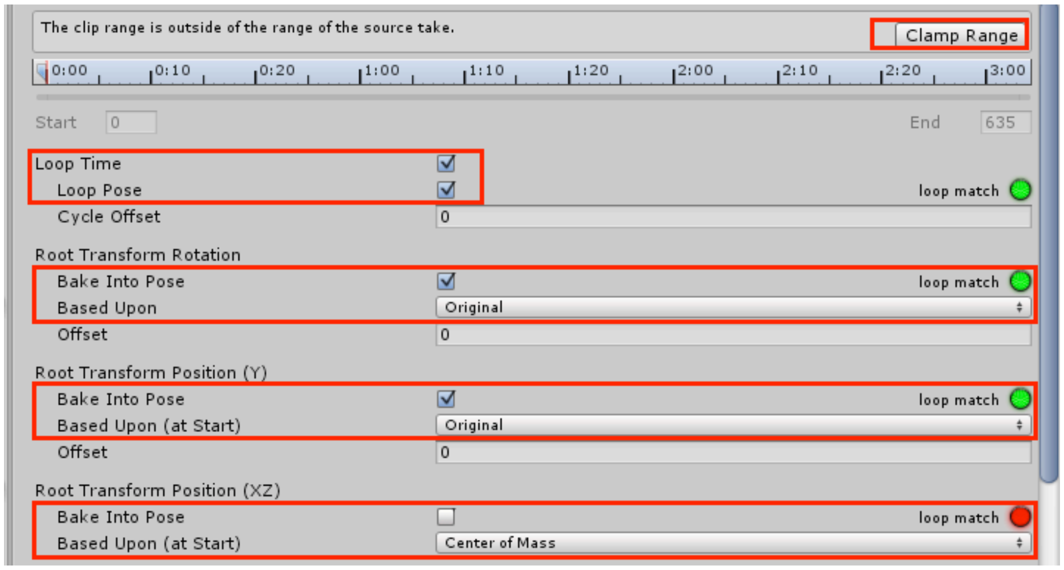
Animation Type: Humanoid
Avatar Definition: Create From This Model
Optimize Game Objects: ☐

Insert image 1362OT_07_09.png

4. Now activate the **Animations** section (to the right of the **Rig**). Select the clip **rifle_run** (from the **Clips** list). The **Preview** area (on the bottom of the

Inspector) should display the message **No model is available for preview. Please, drag a model into this Preview area**. Drag the **book@T-Pose** into the Preview area to correct this.

5. With **rifle_ run** selected from the **Clips** list, select the clip **rifle_run** (from the **Clips** list) and check the options **Loop Time** and **Loop Pose**. Also, click the button **Clamp Range** to adjust the timeline to the actual time of the animation clip. Then, under **Root Transform Rotation**, check **Bake into Pose,** select **Baked upon (at Start) | Original**; Under **Root Transform Position (Y)**, check **Bake into Pose,** select **Baked upon | Original**; Under **Root Transform Position (XZ)**, leave **Bake into Pose** unchecked, select **Baked upon (at Start) | Center of Mass**. Finally, click **Apply** to confirm changes.


The clip range is outside of the range of the source take.

Start 0 End 635

Loop Time ✓
Loop Pose ✓ loop match ●
Cycle Offset 0

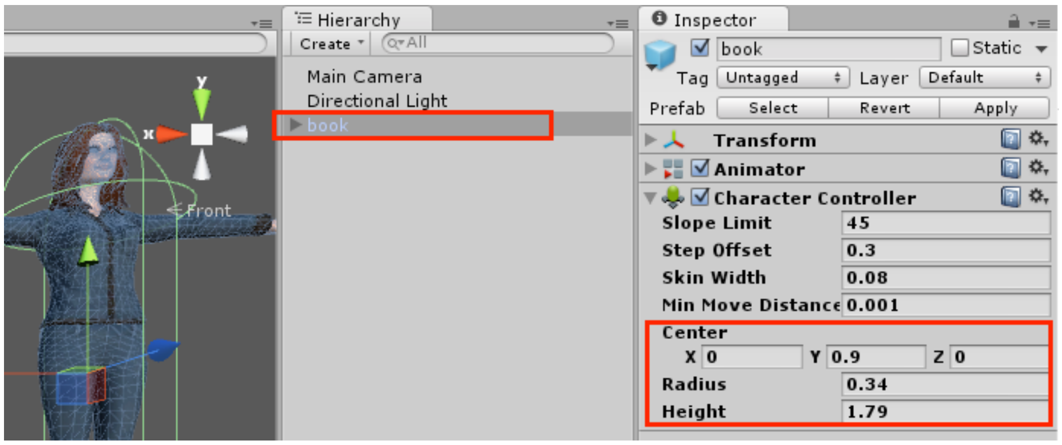
Root Transform Rotation
Bake Into Pose ✓ loop match ●
Based Upon Original
Offset 0

Root Transform Position (Y)
Bake Into Pose ✓ loop match ●
Based Upon (at Start) Original
Offset 0

Root Transform Position (XZ)
Bake Into Pose ☐ loop match ●
Based Upon (at Start) Center of Mass

Insert image 1362OT_07_10.png

6. Repeat steps 3 to 5 for each one of the following animation clips: **Swat@run_backwards**, **Swat@strafe**, **Swat@strafe_2**, **Swat@strafe_left**, **Swat@strafe_right**, **Swat@walking**, and **Swat@walking_backwards**.
7. From the **Project** view, select the **book** prefab and drag it into the **Hierarchy** view, placing it on the scene.
8. From the **Hierarchy** view, select the **book** game object and attach a **Character Controller** component to it (**menu Component | Physics | Character Controller**). Then set its **Skin Width** as `0.0001`, and its **Center** as **X: 0**, **Y: 0.9**, **Z:0**; Also, change its **Radius** to **0.34** and its **Height** to **1.79**.

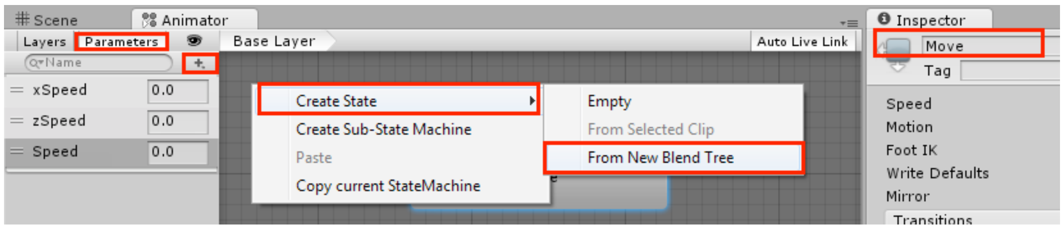


book | Static
Tag Untagged | Layer Default
Prefab Select Revert Apply
Transform
Animator
Character Controller

Slope Limit	45
Step Offset	0.3
Skin Width	0.08
Min Move Distance	0.001
Center	X 0 Y 0.9 Z 0
Radius	0.34
Height	1.79

Insert image 1362OT_07_11.png

9. In the **Project** view, open the **MainCharacter** controller.
10. In the top-left corner of the **Animator** view, activate the **Parameters** section and use the + sign to create three new **Parameters (Float)** named `xSpeed`, `zSpeed`, and `Speed`.
11. We do have an **Idle** state for our character, but we need new ones. Right-click the gridded area and, from the context menu, select **Create State | From New Blend Tree**. Change its name, from the **Inspector** view, to `Move`.

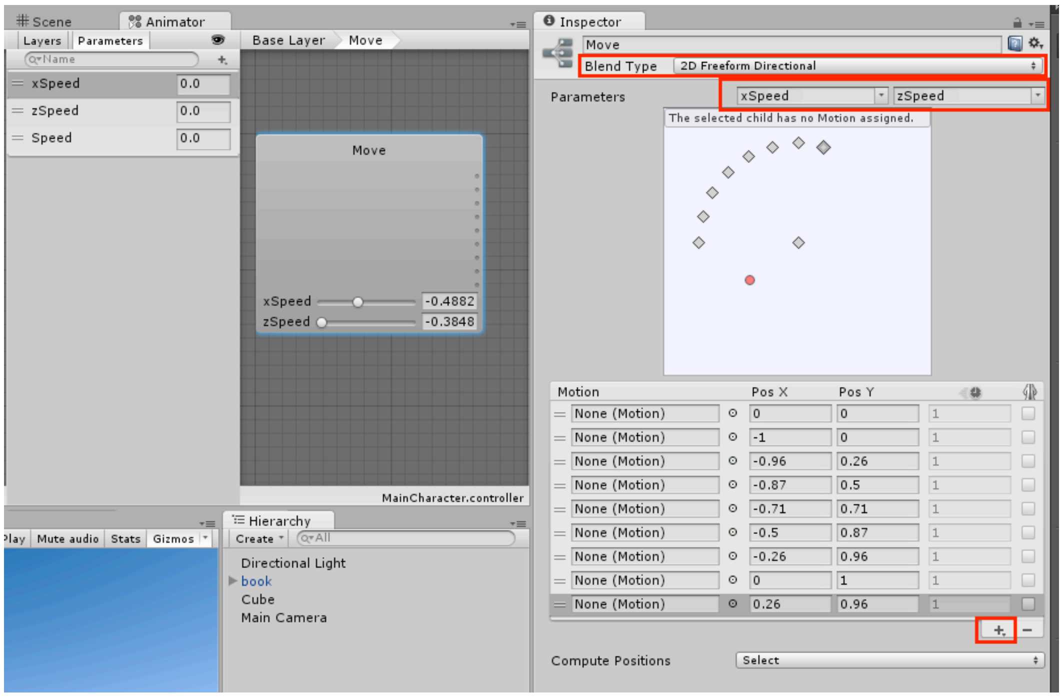


xSpeed	0.0
zSpeed	0.0
Speed	0.0

Move

Insert image 1362OT_07_12.png

12. Double-click the **Move** state. You will see the empty **Blend Tree** you have created. Select it and, in the **Inspector** view, rename it to `Move`. Then, change its **Blend Type** to **2D Freeform Directional**, also setting as **Parameters xSpeed** and **zSpeed**. Finally, using the + sign from the bottom of the **Motion** list, add nine new **Motion Fields**.



xSpeed	0.0
zSpeed	0.0
Speed	0.0

Move

xSpeed -0.4882
zSpeed -0.3848

Directional Light
book
Cube
Main Camera

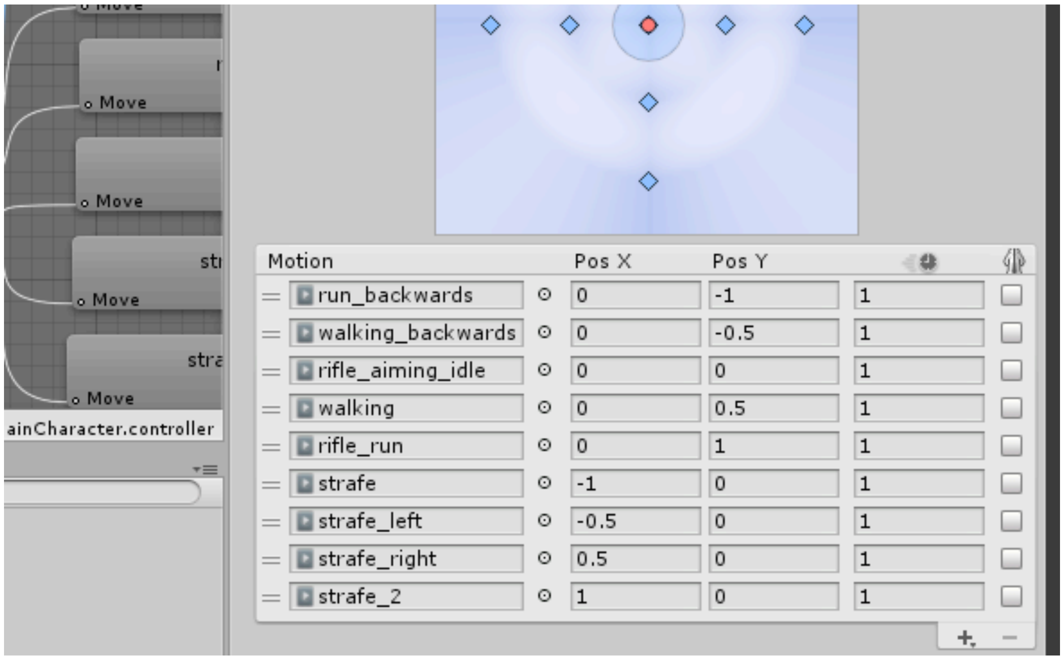
Move
Blend Type 2D Freeform Directional
Parameters xSpeed zSpeed
The selected child has no Motion assigned.

Motion	Pos X	Pos Y	
None (Motion)	0	0	1
None (Motion)	-1	0	1
None (Motion)	-0.96	0.26	1
None (Motion)	-0.87	0.5	1
None (Motion)	-0.71	0.71	1
None (Motion)	-0.5	0.87	1
None (Motion)	-0.26	0.96	1
None (Motion)	0	1	1
None (Motion)	0.26	0.96	1

Compute Positions Select

Insert image 1362OT_07_13.png

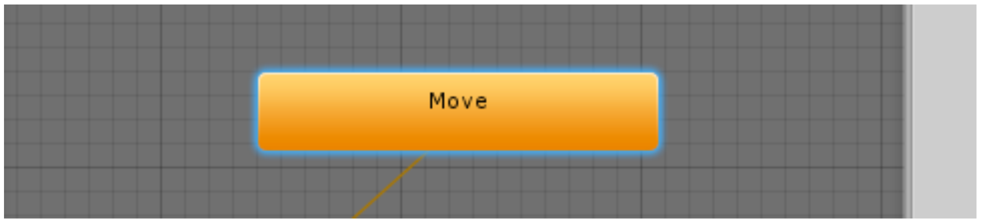
13. Now, populate the **Motion** list with the following Motion clips and respective **Pos X** and **Pos Y** values: **run_backwards**, `0`, `-1`; **walking_backwards**, `0`,`-0.5`; **rifle_aiming_idle**, `0`, `0`; **walking**, `0`, `0.5`; **rifle_run**, `0`, `1`; **strafe**, `-1`, `0`; **strafe_left**, -`0.5`, `0`; **strafe_right**, `0.5`, `0`; **strafe_2**, `1`, `0`. You can populate the **Motion** list by selecting it from the list or, if there is more than one clip with the same name, dragging it from the **Project** view into the slot (by expanding the appropriate model icon).



Motion	Pos X	Pos Y		
run_backwards	0	-1	1	☐
walking_backwards	0	-0.5	1	☐
rifle_aiming_idle	0	0	1	☐
walking	0	0.5	1	☐
rifle_run	0	1	1	☐
strafe	-1	0	1	☐
strafe_left	-0.5	0	1	☐
strafe_right	0.5	0	1	☐
strafe_2	1	0	1	☐

Insert image 1362OT_07_14.png

14. Double-click the gridded area to go from the **Move** blend tree back to the **Base Layer**.
15. Since we have the `rifle_aiming_idle` Motion clip within our **Move** blend tree, we can get rid of the original **Idle** state. Right-click the **Idle** state box and, from the menu, select **Delete**. The **Move** blend state will become the new default state, turning orange.



Insert image 1362OT_07_15.png

16. Now we must create the script that will actually transform the player's input into those variables created to control the animation.
17. From the **Project** view, create a new **C# Script** and name it `BasicController`.

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

```
using UnityEngine;
using System.Collections;

      public class BasicController: MonoBehaviour {
         private Animator anim;
         private CharacterController controller;
         public float transitionTime = .25f;
         private float speedLimit = 1.0f;
         public bool moveDiagonally = true;
         public bool mouseRotate = true;
         public bool keyboardRotate = false;

         void Start () {
            controller = GetComponent<CharacterController>();
            anim = GetComponent<Animator>();
         }

         void Update () {
            if(controller.isGrounded){
               if (Input.GetKey (KeyCode.RightShift)
      ||Input.GetKey (KeyCode.LeftShift))
                  speedLimit = 0.5f;
                else
                  speedLimit = 1.0f;

               float h = Input.GetAxis("Horizontal");
               float v = Input.GetAxis("Vertical");
               float xSpeed = h * speedLimit;
               float zSpeed = v * speedLimit;
               float speed = Mathf.Sqrt(h*h+v*v);

               if(v!=0 && !moveDiagonally)
                  xSpeed = 0;

               if(v!=0 && keyboardRotate)
                  this.transform.Rotate(Vector3.up * h,
      Space.World);

               if(mouseRotate)
                  this.transform.Rotate(Vector3.up *
      (Input.GetAxis("Mouse X")) * Mathf.Sign(v), Space.World);

               anim.SetFloat("zSpeed", zSpeed, transitionTime,
      Time.deltaTime);
```

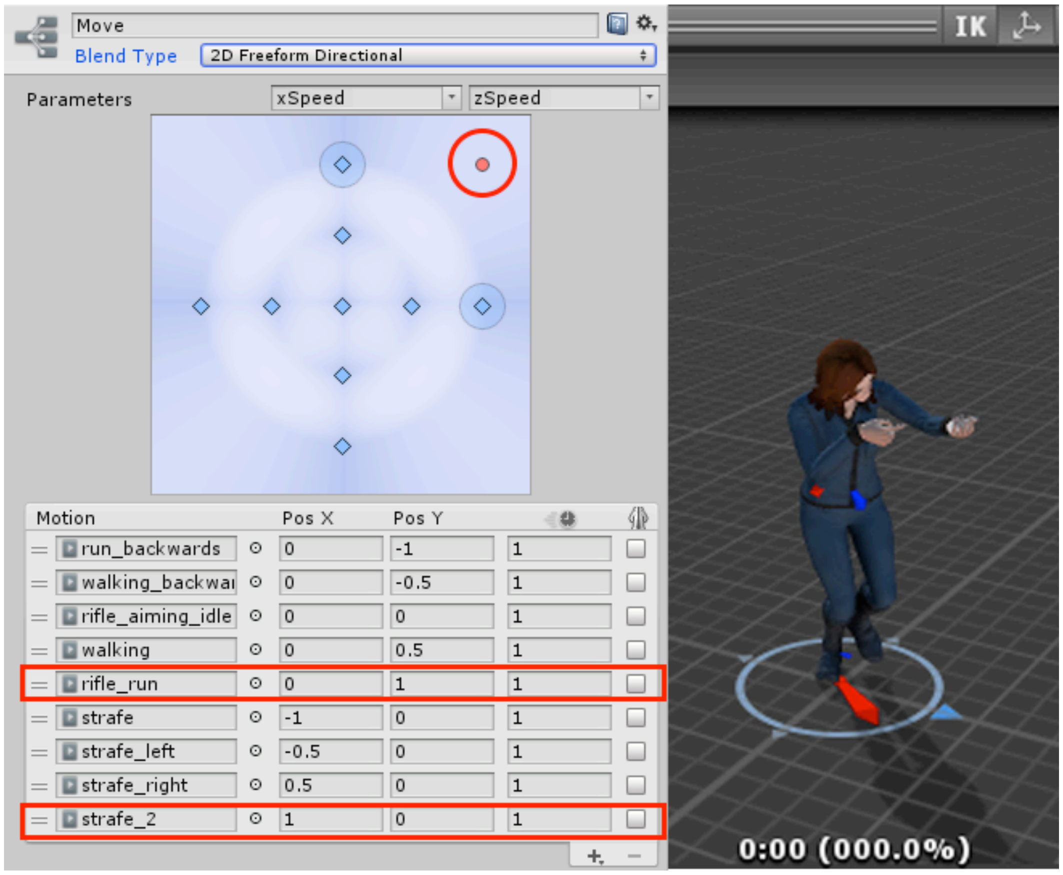
```
            anim.SetFloat("xSpeed", xSpeed, transitionTime,
Time.deltaTime);
            anim.SetFloat("Speed", speed, transitionTime,
Time.deltaTime);
        }
    }
}
```

19. Save your script and attach it to the **book** game object in the **Hierarchy** view. Then, add a **Plane** (menu option **GameObject | 3D Object | Plane**) and place it beneath the character.
20. Play your scene and test the game. You should be able to control your character with the arrow keys (or *WASD* keys). Keeping the *Shift* key pressed will slow it down.

How it works...

Whenever the `BasicController` script detects any directional keys in use, it sets the `Speed` variable of the **Animator** to a value higher than 0, changing the **Animator** state from **Idle** to **Move**. The **Move** state, in its turn, blends the motion clips it was populated with according to the input values for `xSpeed` (obtained from **Horizontal Axis i**nput, typically *A* and *D* keys) and `zSpeed` (obtained from **Vertical Axis** input, typically *W* and *S* keys). Since **Mecanim** is capable of applying Root Motion to characters, our character will actually move in the resulting direction.

For instance, if *W* and *D* keys are pressed, `xSpeed` and `zSpeed` values will rise to 1.0. From the **Inspector** view, it is possible to see that such combination will result in a blend between motion clips **rifle_run** and **strafe_2**, making the character run diagonally (front + right).



Move

Blend Type: 2D Freeform Directional

Parameters: xSpeed, zSpeed

Motion	Pos X	Pos Y		
run_backwards	0	-1	1	☐
walking_backwar	0	-0.5	1	☐
rifle_aiming_idle	0	0	1	☐
walking	0	0.5	1	☐
rifle_run	0	1	1	☐
strafe	-1	0	1	☐
strafe_left	-0.5	0	1	☐
strafe_right	0.5	0	1	☐
strafe_2	1	0	1	☐

Insert image 1362OT_07_18.png

Our **BasicController** includes three checkboxes for more options: **Move Diagonally**, set as **true** by default, allows for blends between forward/backward and left/right clips; **Mouse Rotate**, set as **true** by default, allows for rotating the character with the mouse, changing her direction while moving; **Keyboard Rotate**, set as **false** by default, allows for rotating the character through simultaneous use of left/right and forward/backwards directional keys.

There's more...

Our **Blend Tree** used the **2D Freeform Directional Blend Type**. However, if we had only four animation clips (forward, backwards, left and right), **2D Simple Directional** would have been a better option. Learn more on the following links:

- Learn more about Blend Trees and 2D Blending from Unity's Documentation at:

`docs.unity3d.com/Manual/BlendTree-2DBlending.html`

- Also, if you want to learn more about Mecanim Animation System, there are some links you might want to check out, such as Unity's documentation at : `docs.unity3d.com/Documentation/Manual/MecanimAnimationSystem.html`.
- Mecanim Example Scenes, available at Unity Asset Store: `u3d.as/content/unity-technologies/mecanim-example-scenes/3Bs`
- Mecanim Video Tutorial, at: `video.unity3d.com/video/7362044/unity-40-mecanim-animation`

Mixing animations with Layers and Masks

Mixing animations is a great way of adding complexity to your animated characters without requiring a vast number of animated clips. Using **Layers** and **Masks**, we can combine different animations by playing specific clips for specific body parts of the character. In this recipe, we will apply this technique to our animated character, triggering animation clips for firing a rifle and throwing a grenade with the character's upper body, while keeping the lower body moving or idle, according to the player's input.

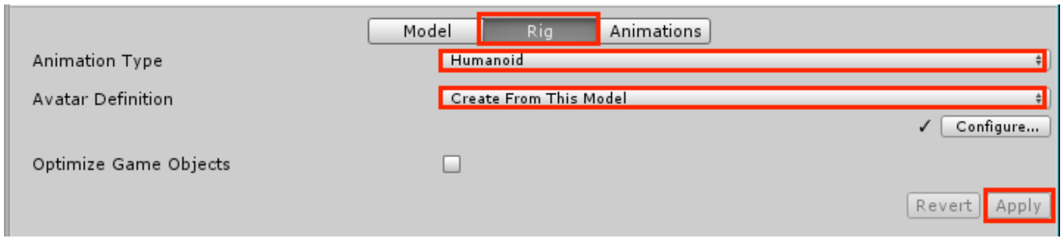
Getting ready

For this recipe, we have prepared a Unity Package named `Mixing`, containing a basic scene that features an animated character. The package can be found inside the `1362_07_03` folder, along with animation clips `Swat@firing_rifle.fbx` and `Swat@toss_grenade.fbx`.

How to do it...

To mix animations using **Layers** and **Masks**, follow these steps:

1. Create a new project and import the `Mixing` Unity Package. Then, from the **Project** view, open the **mecanimPlayground** level.
2. Import the files `Swat@firing_rifle.fbx` and `Swat@toss_grenade.fbx` into the project.
3. We need to configure the animation clips. From the **Project** view, select the **Swat@firing_rifle** animation clip.
4. Activate the **Rig** section. Change **Animation Type** to **Humanoid** and **Avatar Definition** to **Create From this Model**. Confirm by clicking **Apply**.



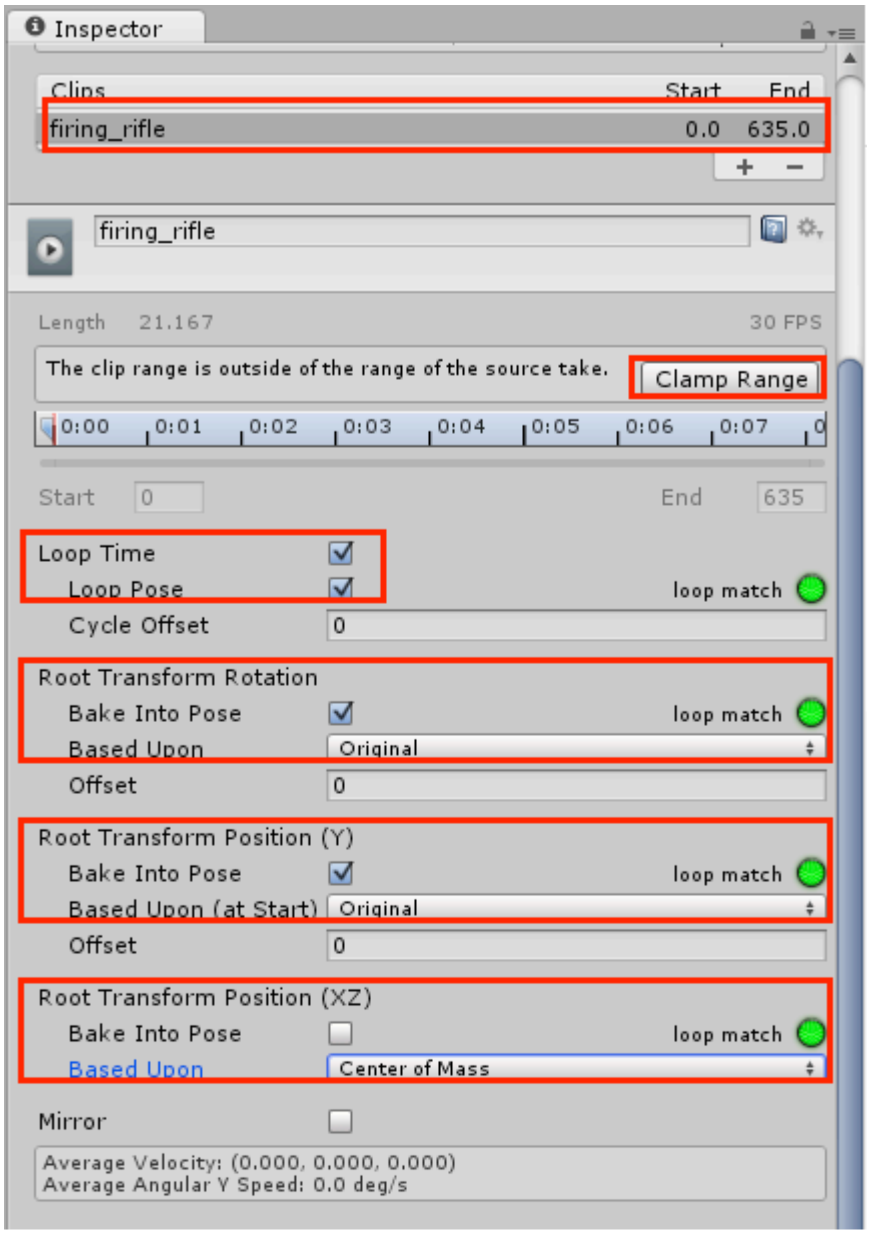
Animation Type: Humanoid

Avatar Definition: Create From This Model

Optimize Game Objects

Insert image 1362OT_07_09.png

5. Now activate the **Animations** section. Select the clip **firing_rifle** (from the **Clips** list), click the button **Clamp Range** to adjust the timeline, and check the options **Loop Time** and **Loop Pose**. Under **Root Transform Rotation**, check **Bake into Pose**, select **Baked upon | Original**; Under **Root Transform Position (Y)**, check **Bake into Pose**, select **Baked upon (at Start) | Original**; Under **Root Transform Position (XZ)**, leave **Bake into Pose** unchecked. Click **Apply** to confirm changes.



firing_rifle 0.0 635.0

firing_rifle

Length 21.167 30 FPS

The clip range is outside of the range of the source take. Clamp Range

Start 0 End 635

Loop Time ✓
Loop Pose ✓ loop match
Cycle Offset 0

Root Transform Rotation
Bake Into Pose ✓ loop match
Based Upon Original
Offset 0

Root Transform Position (Y)
Bake Into Pose ✓ loop match
Based Upon (at Start) Original
Offset 0

Root Transform Position (XZ)
Bake Into Pose ☐ loop match
Based Upon Center of Mass

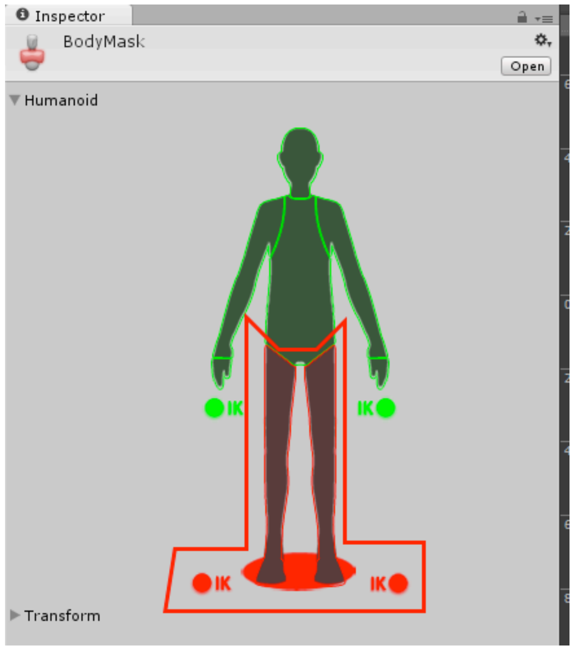
Mirror ☐

Average Velocity: (0.000, 0.000, 0.000)
Average Angular Y Speed: 0.0 deg/s

Insert image 1362OT_07_19.png

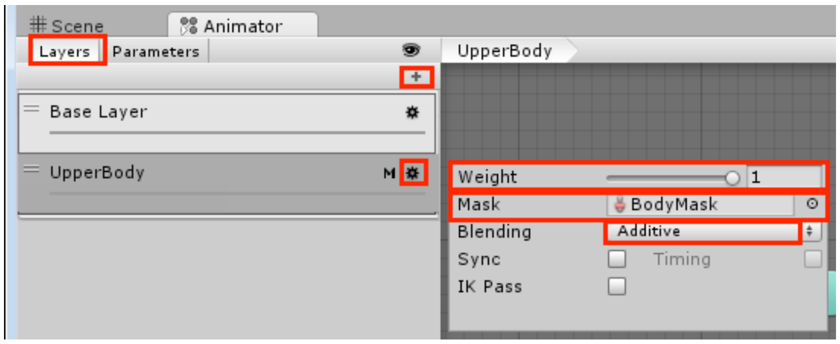
6. Select the **Swat@toss_grenade** animation clip. Activate the **Rig** section. Then, change **Animation Type** to **Humanoid** and **Avatar Definition** to **Create From this Model**. Confirm by clicking **Apply**.

7. Now activate the **Animations** section. Select the clip **toss_grenade** (from the **Clips** list), click the button **Clamp Range** to adjust the timeline, and leave the options **Loop Time** and **Loop Pose** unchecked. Under **Root Transform Rotation**, check **Bake into Pose**, select **Baked upon (at Start) | Original**; Under **Root Transform Position (Y)**, check **Bake into Pose**, select **Baked upon (at Start) | Original)**. Under **Root Transform Position (XZ)**, leave **Bake into Pose** unchecked. Click **Apply** to confirm changes.
8. Let's create a **Mask**. From the **Project** view, click the **Create** button and add an **Avatar Mask** to the project. Name it **BodyMask**.
9. Select the **BodyMask** and, in the **Inspector** view, expand the **Humanoid** section to unselect the character's legs, base, and IK spots, turning their outline red.



Insert image 1362OT_07_20.png

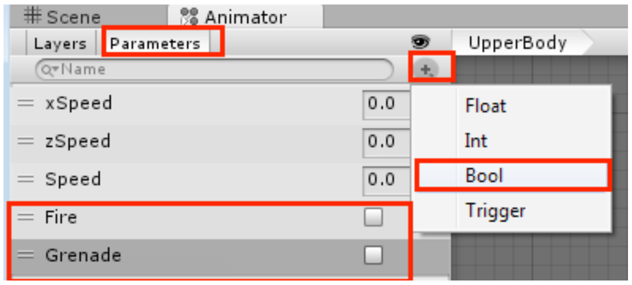
10. From the **Hierarchy** view, select the **book** character. Then, from the **Animator** component in the **Inspector** view, double click the **MainCharacter** controller to open it.
11. In the **Animator** view, create a new **Layer** by clicking the **+** sign on the top-left **Layers** tab, above the **Base Layer**.
12. Name the new Layer **UpperBody** and click the gear icon for the settings. Then, change its **Weight** to `1` and select the **BodyMask** in the **Mask** slot. Also, change Blending to **Additive**.



Weight 1
Mask BodyMask
Blending Additive
Sync
Timing
IK Pass

Insert image 1362OT_07_21.png

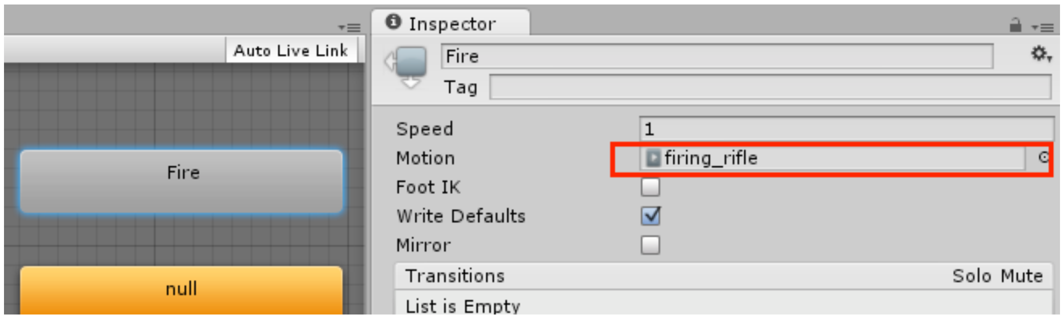
13. Now, in the **Animator** view, with the **UpperBody** layer selected, create three new empty states (by right-clicking the gridded area and selecting, from the menu, **Create State | Empty**). Name the default (orange) state **null**, and the other two as **Fire** and **Grenade**.
14. Now, access the **Parameters** tab and add two new parameters of the Boolean type: `Fire` and `Grenade`.



xSpeed 0.0
zSpeed 0.0
Speed 0.0
Fire
Grenade

Insert image 1362OT_07_22.png

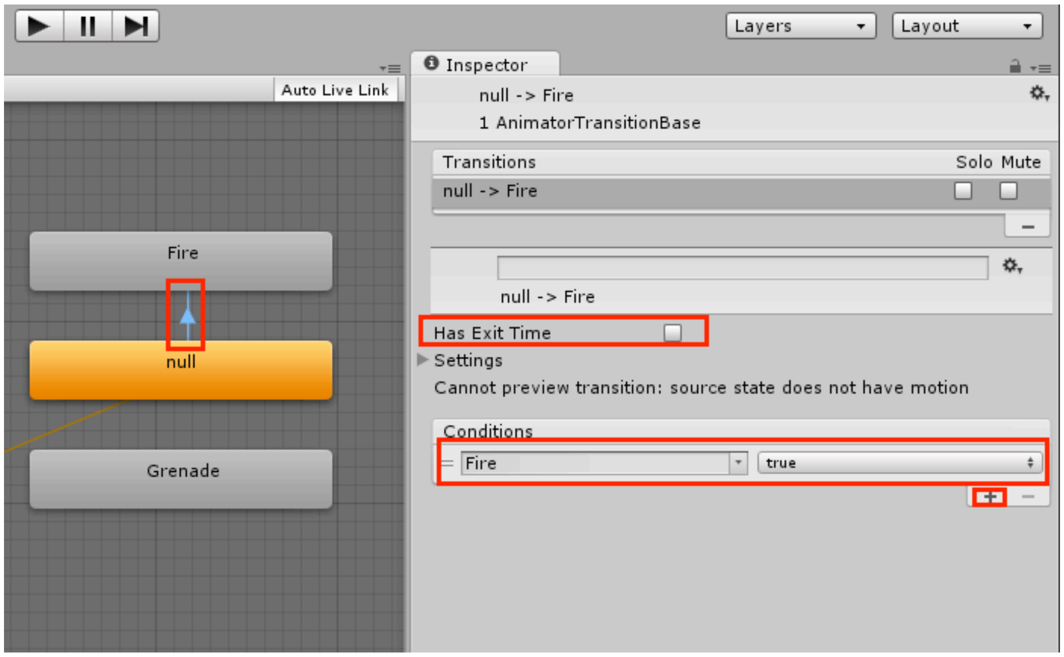
15. Select the **Fire** state and, in the **Inspector** view, add the **firing_rifle** animation clip to the **Motion** field.



Fire
Tag
Speed 1
Motion firing_rifle
Foot IK
Write Defaults ✓
Mirror
Transitions Solo Mute
List is Empty

Insert image 1362OT_07_23.png

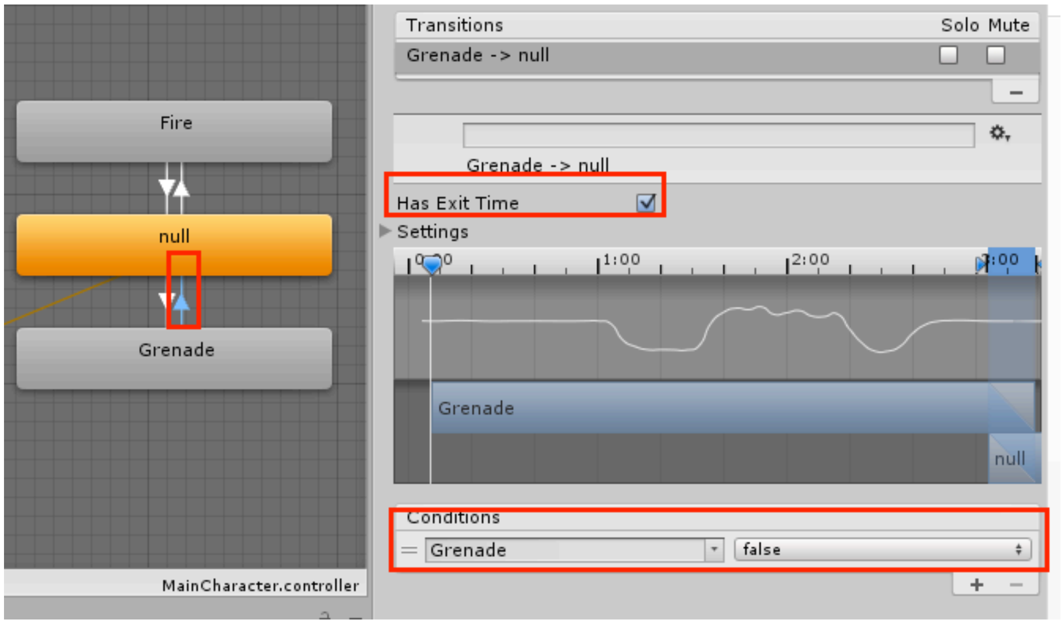
16. Now select the **Grenade** state and, in the **Inspector** view, add the **toss_grenade** animation clip to the **Motion** field.
17. Right-click the **null** state box and, from the menu, select **Make Transition**. Then, drag the white arrow into the **Fire** box.
18. Select the arrow (it should turn blue). From the **Inspector** view, uncheck the option **Has Exit Time**. Then, access the **Conditions** list, click the **+** sign to add a new condition, and set it as **Fire**, **true**.



null -> Fire
1 AnimatorTransitionBase
Transitions Solo Mute
null -> Fire
null -> Fire
Has Exit Time
Settings
Cannot preview transition: source state does not have motion
Conditions
Fire true

Insert image 1362OT_07_24.png

19. Now, make a transition from **null** to **Grenade**. Select the arrow (it should turn blue). From the **Inspector** view, uncheck the option **Has Exit Time**. Then, access the **Conditions** list, click the **+** sign to add a new condition, and set it as **Grenade**, **true**.
20. Now, create transitions from **Fire** to **null** and from **Grenade** to **null**. Then, select the arrow that goes from **Fire** to **null** and, in the **Conditions** box, select the options **Fire**, **false**. Leave the option **Has Exit Time** checked.
21. Finally, select the arrow that goes from **Grenade** to **null**. Leave the option **Has Exit Time** checked.



Insert image 1362OT_07_25.png

22. From the **Hierarchy** view, select the **book** character. Locate, in the **Inspector** view, the **Basic Controller** component and open its script.
23. Immediately before the end of the `Update()` function, add the following code:

```
if(Input.GetKeyDown(KeyCode.F)){
    animator.SetBool("Grenade", true);
} else {
    anim.SetBool("Grenade", false);
}
if(Input.GetButtonDown("Fire1")){
    anim.SetBool("Fire", true);
}
```

```
        if(Input.GetButtonUp("Fire1")){
            anim.SetBool("Fire", false);
        }
```

24. Save the script and play your scene. You should be able to trigger **firing_rifle** and **toss_grenade** animations by clicking the fire button and pressing the *F* key. Observe how the character legs still respond to the **Move** animation state.

How it works...

Once the **Avatar Mask** is created, it can be used as a way of filtering which body parts will actually play the animation states of a particular layer. In our case, we have constrained our **fire_rifle** and **toss_grenade** animation clips to the upper body of our character, leaving the lower body free to play the movement-related animation clips such as walking, running, and strafing.

There's more...

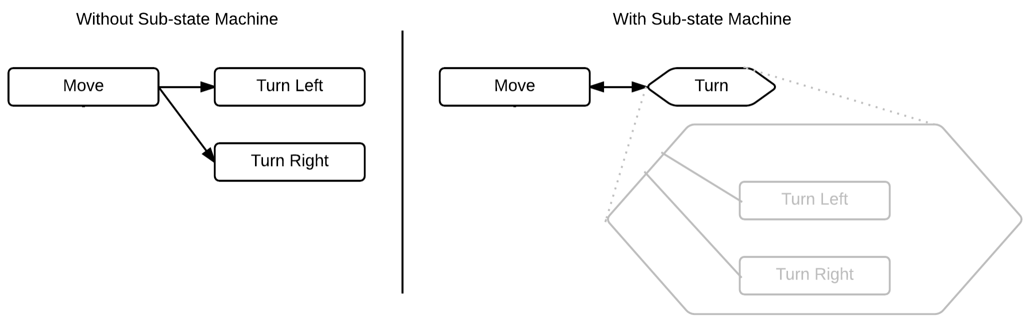
You might have noticed that the **UpperBody** layer has a parameter named **Blending**, which we have set to **Additive**. That means that animation states in that layer will animation be added to the ones from lower layers. If changed to **Override**, the animation from that would override animation states from lower layers when played. In our case, **Additive** helped keeping the aim when firing while running.

For more information on **Animation Layers** and **Avatar Body Masks**, check out Unity's documentation at:

- `docs.unity3d.com/Manual/AnimationLayers.html`
- `docs.unity3d.com/Manual/class-AvatarBodyMask.html`

Organizing States into Sub-State Machines

Whenever the Animator area gets too cluttered, you can think of organizing your Animation States into Sub-State Machines. In this recipe, we will use this technique to organize animation states for turning the character. Also, since the provided animation clips do not include root motion, we will use the opportunity to illustrate how to overcome the lack of root motion via script, using it to turn the character 45 degrees to the left and right sides.



Insert image 1362OT_07_37.png

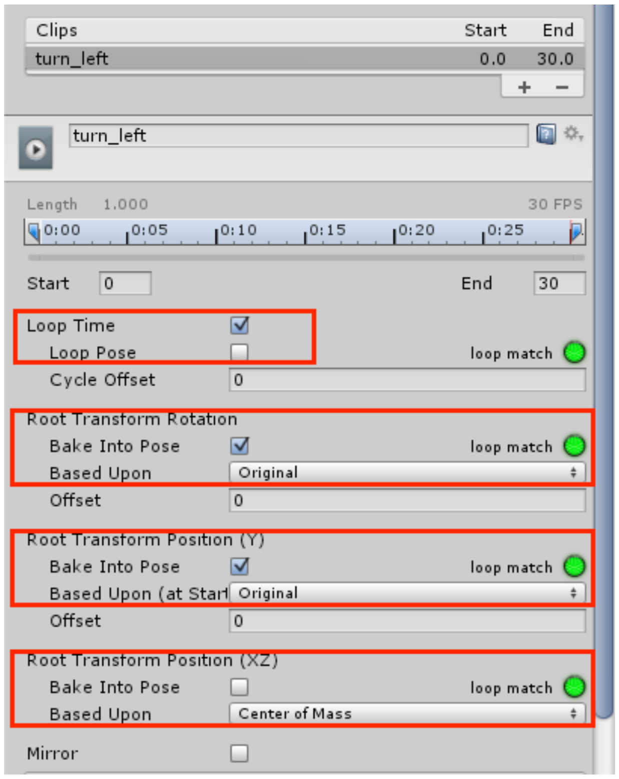
Getting ready

For this recipe, we have prepared a Unity Package named `Turning`, containing a basic scene that features an animated character. The package can be found inside the `1362_07_04` folder, along with animation clips `Swat@turn_right_45_degrees.fbx` and `Swat@turn_left.fbx`.

How to do it...

To apply root motion via script, please follow these steps:

1. Create a new project and import the `Turning` Unity Package. Then, from the **Project** view, open the **mecanimPlayground** level.
2. Import the files `Swat@turn_right_45_degrees.fbx` and `Swat@turn_left.fbx` into the project.
3. We need to configure our animation clips. Select the **Swat@turn_left** file from the **Project** view.
4. Activate the **Rig** section. Change **Animation Type** to **Humanoid** and **Avatar Definition** to **Create From this Model**. Confirm by clicking **Apply**.
5. Now activate the **Animations** section. Select the clip **turn_left** (from the **Clips** list), click the button **Clamp Range** to adjust the timeline, and check the option **Loop Time**. Under **Root Transform Rotation**, check **Bake into Pose**, select **Baked upon (at Start) | Original**; Under **Root Transform Position (Y)**, check **Bake into Pose**, select **Baked upon (at Start) | Original**; Under **Root Transform Position (XZ)**, leave **Bake into Pose** unchecked. Click **Apply** to confirm changes.



Clips	Start	End
turn_left	0.0	30.0

turn_left

Length 1.000 30 FPS

Start 0 End 30

Loop Time ✓
Loop Pose ☐ loop match
Cycle Offset 0

Root Transform Rotation
Bake Into Pose ✓ loop match
Based Upon Original
Offset 0

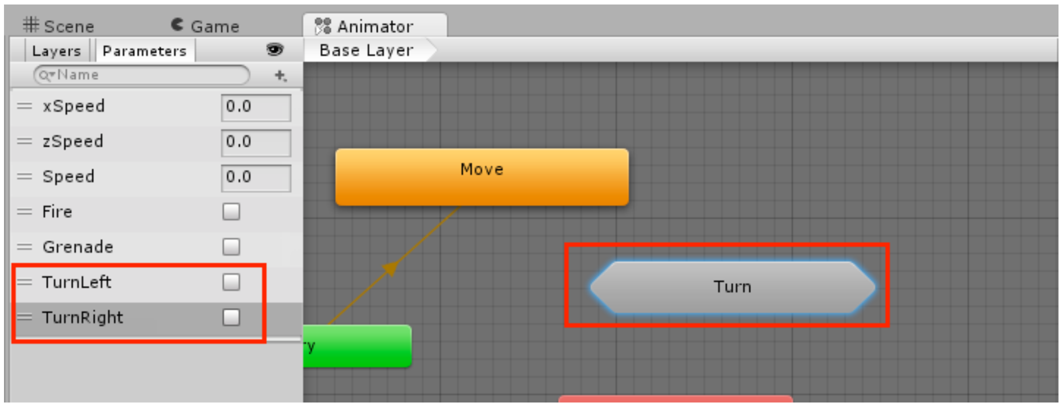
Root Transform Position (Y)
Bake Into Pose ✓ loop match
Based Upon (at Start) Original
Offset 0

Root Transform Position (XZ)
Bake Into Pose ☐ loop match
Based Upon Center of Mass

Mirror ☐

Insert image 1362OT_07_26.png

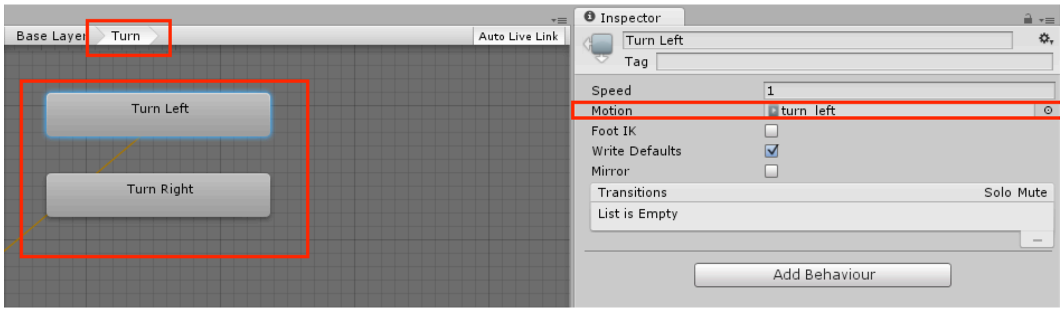
6. Repeat steps 4 and 5 for **Swat@turning_right_45_degrees**.
7. From the **Hierarchy** view, select the **book** character. Then, from the **Animator** component in the **Inspector** view, open the **MainCharacter** controller.
8. From the top-left corner of the **Animator** view, activate the **Parameters** section and use the **+** sign to create two new **Parameters (Boolean)** named `TurnLeft` and `TurnRight`.
9. Right-click the gridded area. From the context menu, select **Create Sub-State Machine**. From the **Inspector** view, rename it `Turn`.



Insert image 1362OT_07_27.png

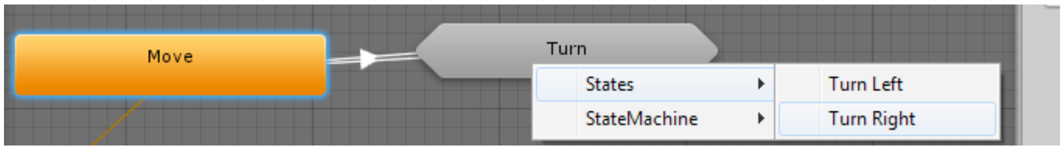
10. Double-click the **Turn** Sub-State Machine. Right-click the gridded area, select **Create State | Empty** and add a new State. Rename it `Turn Left`. Then add another state, named `Turn Right`.

11. From the **Inspector** view, populate `Turn Left` with the **turn_left** Motion clip. Then, populate `Turn Right` with **turning_right_45_degrees**.



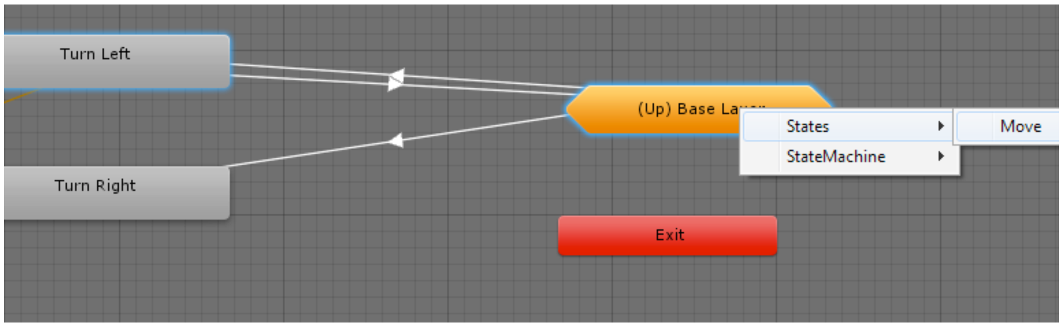
Insert image 1362OT_07_28.png

12. Get out of the **Turn** Sub-state machine, back into the **Base Layer**. By right-clicking each state and selecting the option **Make Transition**, create transitions between **Move** and **Turn Left**; **Move** and **Turn Right**.



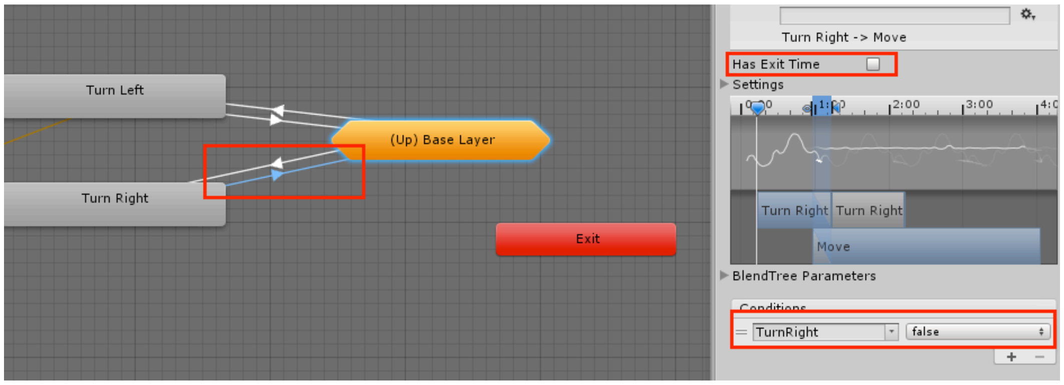
Insert image 1362OT_07_29.png

13. Enter the **Turn** Sub-State machine. Then, create transitions from **Turn Left** and **Turn Right** into the **Move** state.



Insert image 1362OT_07_30.png

14. Select the arrow that goes form **Turn Right** to **(Up) Base Layer**. It should turn blue. From the **Inspector** view, uncheck the option **Has Exit Time**. Then, access the **Conditions** list, click the **+** sign to add a new condition and set it as **TurnRight**, **false**.



Has Exit Time ☐

TurnRight false

Insert image 1362OT_07_31.png

15. Select the arrow that goes from **(Up) Base Layer** to **Turn Right**. From the **Inspector** view, uncheck the option **Has Exit Time**. Then, access the **Conditions** list, click the **+** sign to add a new condition and set it as **TurnRight**, **true**.

16. Repeat steps 14 and 15 with the arrows that go between **(Up) Base Layer** and **Turn Left**, using **TurningLeft** as condition, this time.

17. From the **Hierarchy** view, select the **book** character. Then, from the **Inspector** view, open the Script from the **BasicController** component.

18. Immediately after the line `if(controller.isGrounded){` , add:

```
if(Input.GetKey(KeyCode.Q)){
   anim.SetBool("TurnLeft", true);
   transform.Rotate(Vector3.up * (Time.deltaTime * -45.0f),
Space.World);
}  else {
   anim.SetBool("TurnLeft", false);
}
if(Input.GetKey(KeyCode.E)){
   anim.SetBool("TurnRight", true);
   transform.Rotate(Vector3.up * (Time.deltaTime * 45.0f),
Space.World);
} else {
   anim.SetBool("TurnRight", false);
}
```

19. Save your script. Then, select the **book** character and, from the **Inspector** view, access the **Basic Controller** component. Leave the **Move Diagonally** and **Mouse Rotate** options unchecked. Also, leave the **Keyboard Rotate** option checked. Finally, play the scene. You should be able to turn left and right by using *Q* and *E* keys, respectively.

How it works...

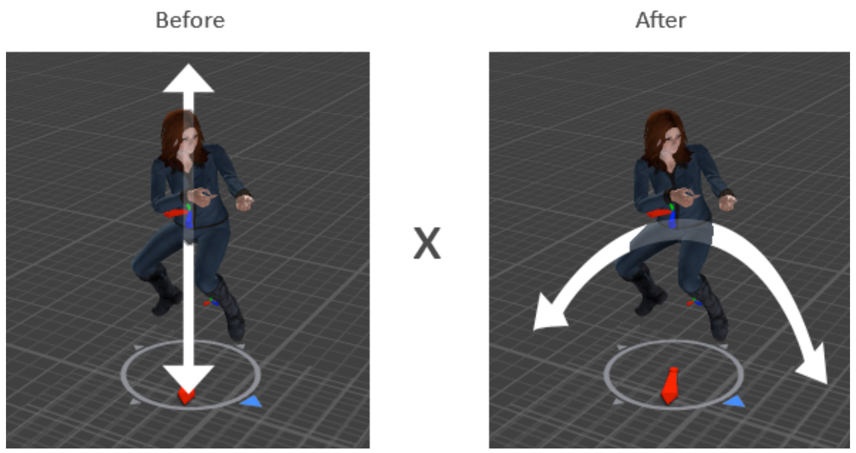
As it should be clear from the recipe, **Sub-state Machines** work in a similar way to Groups or Folders, allowing you to encapsulate a series of state machines into a single entity for easier reference. States from Sub-state Machines can be transitioned from external states, in our case, the **Move** state, or even from different Sub-state Machines.

Regarding the character's rotation, we have overcome the lack of Root Motion by using the command `transform.Rotate(Vector3.up * (Time.deltaTime * -45.0f), Space.World);` to make the character actually turn around when the keys *Q* and *E* are being held down. This command was used in conjunction with `animator.SetBool("TurnLeft", true);` , which triggers the right animation clip.

Transforming the Character Controller via script

Applying **Root Motion** to your character might be a very practical and accurate way to animate it. However, every now and then you might need to manually control one or two aspects of the character movement. Perhaps you have only an in-place animation to work with. Or maybe you want the character's movement to be affected by other variables. In those cases, you will need to override the Root Motion via script.

To illustrate this issue, this recipe makes use of an animation clip for jumping that, originally, moves the character only in the Y-axis. In order to make her move forward or backwards while jumping, we will learn how to access the character's velocity to inform the jump's direction, via script.



Insert image 1362OT_07_38.png

Getting ready

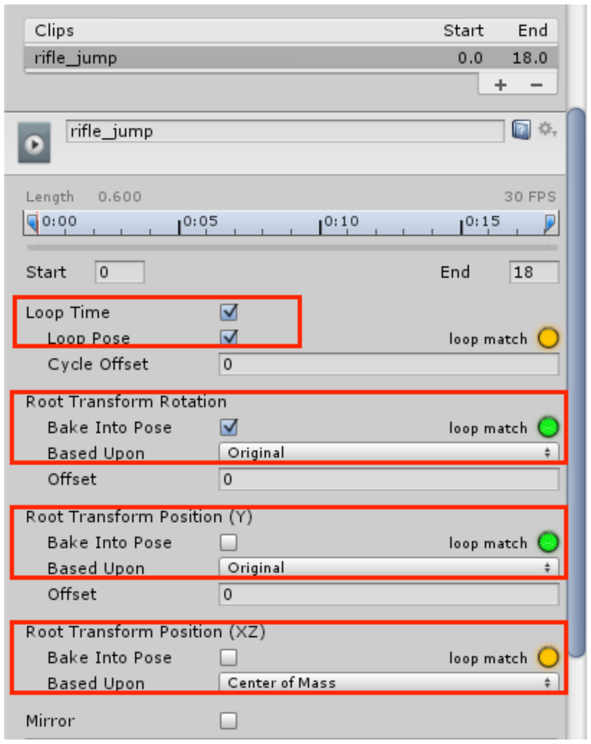
For this recipe, we have prepared a Unity Package named `Jumping`, containing a basic scene that features an animated character. The package can be found inside the `1362_07_05` folder, along with the animation clip `Swat@rifle_jump`.

How to do it...

To apply root motion via script, please follow these steps:

1. Create a new project and import the `Jumping` Unity Package. Then, from the **Project** view, open the **mecanimPlayground** level.
2. Import the file `Swat@rifle_jump.fbx` into the project.
3. We need to configure our animation clip. From the **Original_Character** folder, select the file **Swat@rifle_jump**.
4. Activate the **Rig** section. Change **Animation Type** to **Humanoid** and **Avatar Definition** to **Create From this Model**. Confirm by clicking **Apply**.
5. Now activate the **Animations** section. Select the clip **rifle_jump** (from the **Clips** list), click the button **Clamp Range** to adjust the timeline, check the **Loop Time** and **Loop Pose** options. Under **Root Transform Rotation**, check **Bake into**

Pose, select **Baked upon (at Start) | Original**; Under **Root Transform Position (Y)**, leave **Bake into Pose** unchecked, select **Baked upon (at Start) | Original**; Under **Root Transform Position (XZ)**, leave **Bake into Pose** unchecked. Click **Apply** to confirm changes.



Clips   Start   End
rifle_jump   0.0   18.0

rifle_jump

Length 0.600   30 FPS

Start 0   End 18

Loop Time ✓
Loop Pose ✓   loop match
Cycle Offset 0

Root Transform Rotation
Bake Into Pose ✓   loop match
Based Upon Original
Offset 0

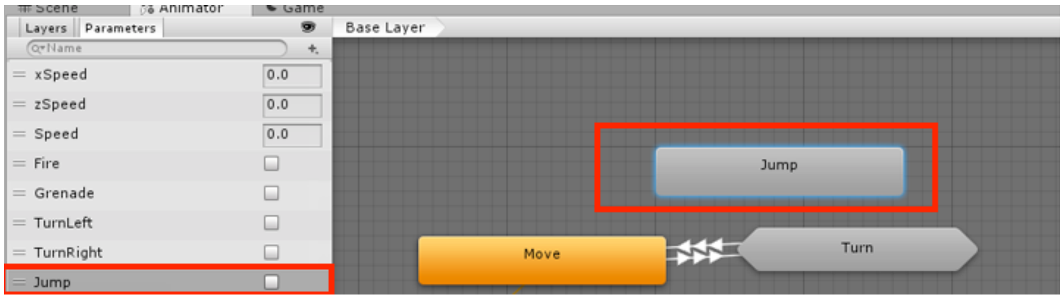
Root Transform Position (Y)
Bake Into Pose ☐   loop match
Based Upon Original
Offset 0

Root Transform Position (XZ)
Bake Into Pose ☐   loop match
Based Upon Center of Mass

Mirror ☐

Insert image 1362OT_07_32.png

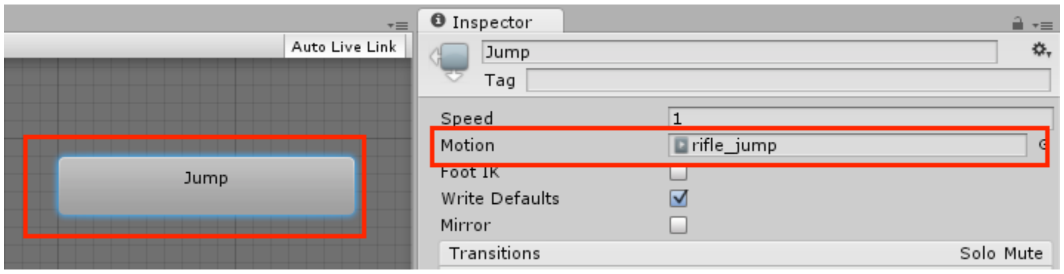
6. From the **Hierarchy** view, select the **book** character. Then, from the **Animator** component in the **Inspector** view, open the **MainCharacter** controller.
7. From the top-left corner of the **Animator** view, activate the **Parameters** section and use the **+** sign to create a new **Parameters (Boolean)** named `Jump`.
8. Right-click the gridded area and, from the context menu, select **Create State | Empty**. Change its name, from the **Inspector** view, to `Jump`.



xSpeed 0.0
zSpeed 0.0
Speed 0.0
Fire
Grenade
TurnLeft
TurnRight
Jump

Insert image 1362OT_07_33.png

9. Select the **Jump** state. Then, from the **Inspector** view, populate it with the **rifle_jump** Motion clip.

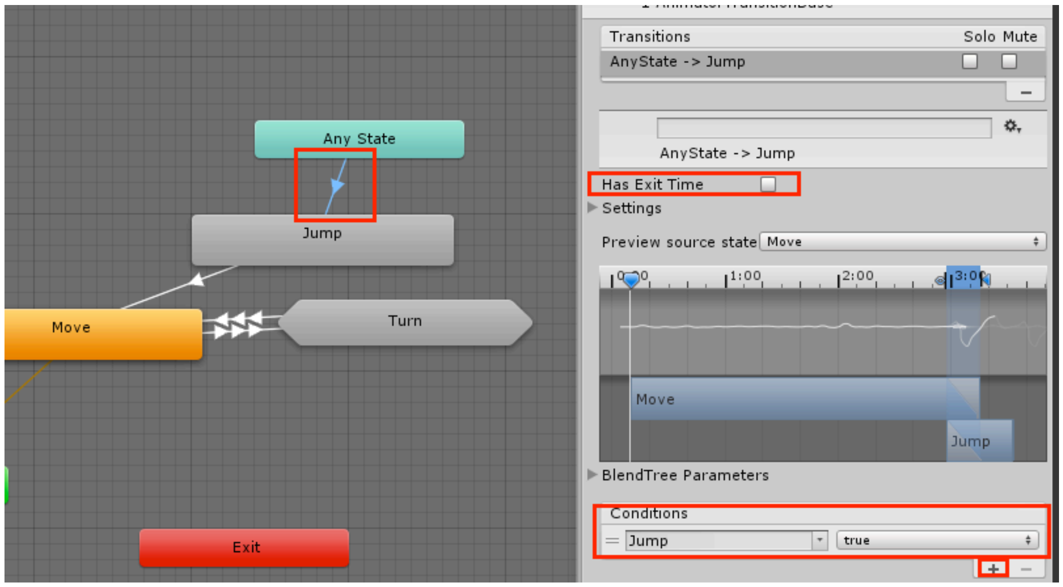


Jump
Tag
Speed 1
Motion rifle_jump
Foot IK
Write Defaults ✓
Mirror

Insert image 1362OT_07_34.png

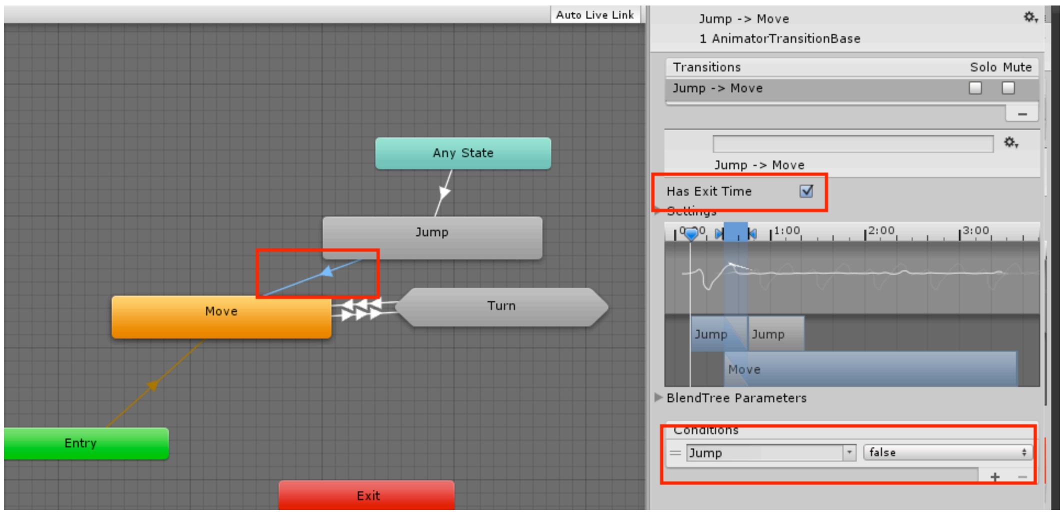
10. Find and right-click the **Any State** state. Then, selecting the **Make Transition** option, create a transition from **Any State** to **Jump**. Select the transition, uncheck **Has Exit Time**, and use the **Jump** variable as Condition (**true**).

11. Now, create a transition from **Jump** to **Move**.



Insert image 1362OT_07_35.png

12. Configure transitions between **Jump** and **Move**, leaving **Has Exit Time** checked, and using the **Jump** variable as Condition (**false**).



Insert image 1362OT_07_36.png

13. From the **Hierarchy** view, select the **book** character. Then, from the **Inspector** view, open the Script from the **BasicController** component.

14. Right before the `Start()` function, add the following code:

```
public float jumpHeight = 3f;
private float verticalSpeed = 0f;
private float xVelocity = 0f;
private float zVelocity = 0f;
```

15. Inside the `Update()` function, find the line containing the code: `if(controller.isGrounded){` , and add the following lines immediatly after it:

```
if (Input.GetKey (KeyCode.Space)) {
   anim.SetBool ("Jump", true);
   verticalSpeed = jumpHeight;
}
```

16. Finally, add a new function following immediately before the final `}` of the code:

```
      void OnAnimatorMove(){
         Vector3 deltaPosition = anim.deltaPosition;
         if (controller.isGrounded) {
            xVelocity = controller.velocity.x;
            zVelocity = controller.velocity.z;
         } else {
            deltaPosition.x = xVelocity * Time.deltaTime;
            deltaPosition.z = zVelocity * Time.deltaTime;
            anim.SetBool ("Jump", false);
         }
         deltaPosition.y = verticalSpeed * Time.deltaTime;
         controller.Move (deltaPosition);
         verticalSpeed += Physics.gravity.y * Time.deltaTime;
         if ((controller.collisionFlags &
CollisionFlags.Below) != 0) {
            verticalSpeed = 0;
         }
      }
```

17. Save your script and play the scene. You should be able to jump around by using the *Space* key. Observe how the character's velocity affects the direction of the jump.

How it works...

Observe that, once this function is added to the script, the field **Apply Root Motion**, on the **Animator** component, changes from a checked box to **Handled by Script**. The reason is that, in order to override the animation clip's original movement, we have placed, inside Unity's `OnAnimatorMove()` function, a series of commands to move our character controller while jumping. The line of code `controller.Move` (deltaPosition); basically replaces the jump's direction from the original animation with the deltaPosition 3D Vector, made of the character's velocity at the instant before the jump (X and Z-axis), and the calculation between the jumpHeight variable and gravity force over time (Y-axis).

Adding rigid props to animated characters

In case you haven't included a sufficient number of props to your character when modeling and animating it, you might want to give her 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.

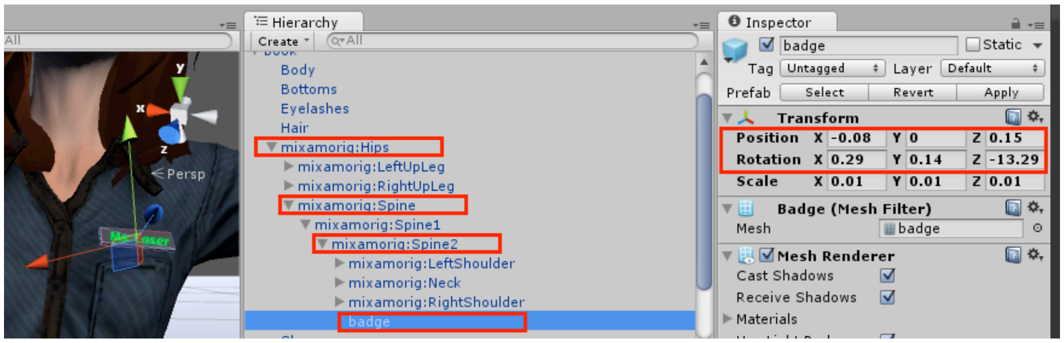
Getting ready

For this recipe, we have prepared a Unity Package named Props, containing a basic scene that features an animated character and a prefab named **badge**. The package can be found inside the 1362_07_06 folder.

How to do it...

To add a rigid prop at runtime to an animated character, follow these steps:

1. Create a new project and import the Props Unity Package. Then, from the **Project** view, open the **mecanimPlayground** level.
2. From the **Project** view, add the **badge** prop to the scene by dragging it into the **Hierarchy** view. Then, make it a child of the **mixamorig:Spine2** transform (use the **Hierarchy** tree to navigate to **book | mixamorig:Hips | mixamorig:Spine | mixamorig:Spine1 | mixamorig:Spine2**). Then, make the **badge** object visible above the character's chest by changing its **Transform Position** to **X:** -0.08, **Y:** 0, **Z:** 0.15; and Rotation to **X:** 0.29, **Y:** 0.14, **Z:**-13.29.



Body
Bottoms
Eyelashes
Hair
mixamorig:Hips
mixamorig:LeftUpLeg
mixamorig:RightUpLeg
mixamorig:Spine
mixamorig:Spine1
mixamorig:Spine2
mixamorig:LeftShoulder
mixamorig:Neck
mixamorig:RightShoulder
badge

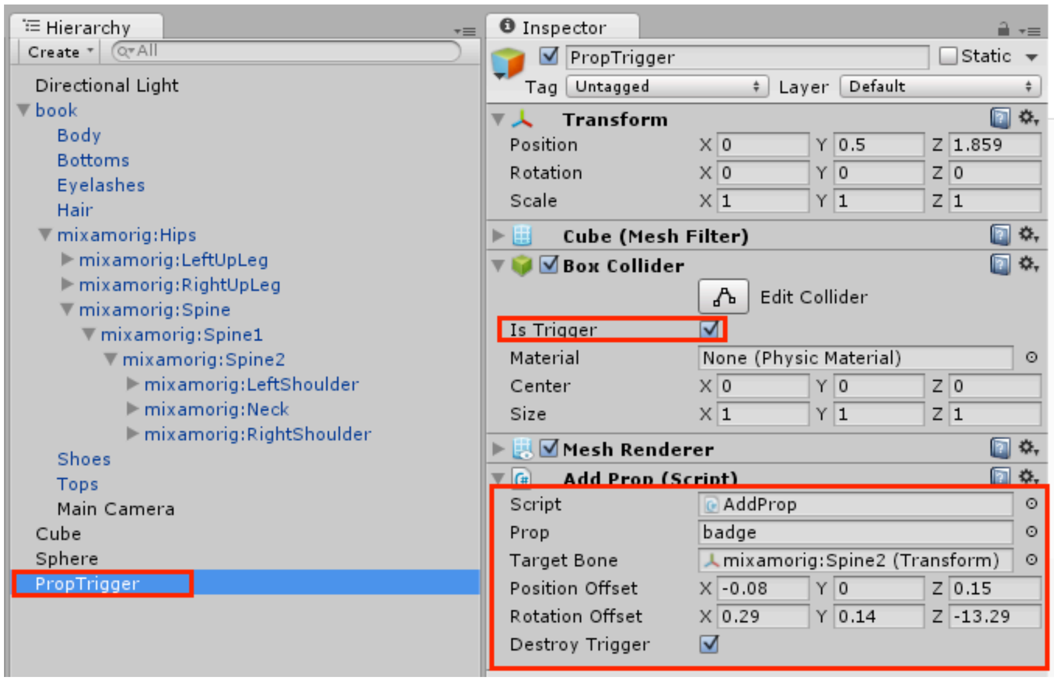
badge Static
Tag Untagged Layer Default
Prefab Select Revert Apply
Transform
Position X -0.08 Y 0 Z 0.15
Rotation X 0.29 Y 0.14 Z -13.29
Scale X 0.01 Y 0.01 Z 0.01
Badge (Mesh Filter)
Mesh badge
Mesh Renderer
Cast Shadows ✓
Receive Shadows ✓
Materials

Insert image 1362OT_07_39.png

3. Take note of the **Position** and **Rotation** values and delete the **badge** object from the scene.
4. Add a new **Cube** to the scene (dropdown **Create** | **3D Object** | **Cube**), rename it **PropTrigger**, and change its Position to **X:** `0`, **Y:** `0.5`, **Z:** `2`.
5. From the **Inspector** view, **Box Collider** component, check the option **Is Trigger**.
6. From the **Project** view, create a new **C# Script** named `AddProp.cs`.
7. Open the script and add the following code:

```
using UnityEngine;
using System.Collections;

public class AddProp : MonoBehaviour {
   public GameObject prop;
   public Transform targetBone;
   public Vector3 positionOffset;
   public Vector3 rotationOffset;
   public bool  destroyTrigger = true;

   void  OnTriggerEnter ( Collider collision  ){

      if (targetBone.IsChildOf(collision.transform)){
         bool  checkProp = false;
         foreach(Transform child in targetBone){
            if (child.name == prop.name)
               checkProp = true;
         }

         if(!checkProp){
            GameObject newprop;
            newprop = Instantiate(prop,
targetBone.position, targetBone.rotation) as GameObject;
            newprop.name = prop.name;
            newprop.transform.parent = targetBone;
            newprop.transform.localPosition +=
positionOffset;
            newprop.transform.localEulerAngles +=
rotationOffset;
            if(destroyTrigger)
               Destroy(gameObject);
         }
      }
   }
}
```

8. Save and close the script.
9. Attach the script **AddProp.cs** to the **PropTrigger** game object.
10. Select the **PropTrigger** and check out its **Add Prop** component. First, populate the **Prop** field with the **badge** prefab. Then, populate **Target Bone** with the **mixamorig:Spine2** transform. Finally, assign the **Position** and **Rotation** values we have previously taken note of to the fields **Position Offset** and **Rotation Offset**, respectively (**Position Offset**: **X**: `-0.08`, **Y**: `0,` **Z**: `0.15`; **Rotation Offset**: **X**: `0.29`, **Y**: `0.14`, **Z**:`-13.29`).



Hierarchy
Create All

- Directional Light
- book
  - Body
  - Bottoms
  - Eyelashes
  - Hair
  - mixamorig:Hips
    - mixamorig:LeftUpLeg
    - mixamorig:RightUpLeg
    - mixamorig:Spine
      - mixamorig:Spine1
        - mixamorig:Spine2
          - mixamorig:LeftShoulder
          - mixamorig:Neck
          - mixamorig:RightShoulder
  - Shoes
  - Tops
  - Main Camera
- Cube
- Sphere
- PropTrigger

Inspector
PropTrigger Static
Tag Untagged Layer Default

Transform

Position	X 0	Y 0.5	Z 1.859
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Cube (Mesh Filter)

Box Collider
Edit Collider

Is Trigger	✓		
Material	None (Physic Material)		
Center	X 0	Y 0	Z 0
Size	X 1	Y 1	Z 1

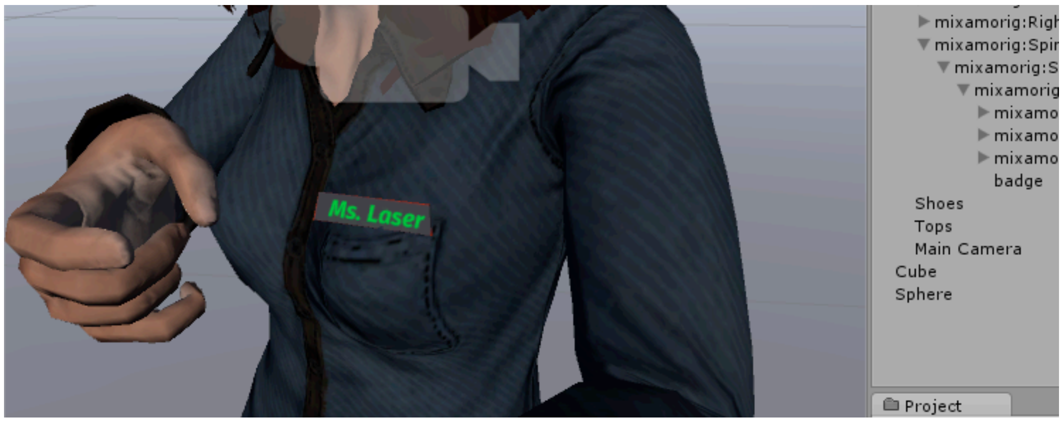
Mesh Renderer

Add Prop (Script)

Script	AddProp		
Prop	badge		
Target Bone	mixamorig:Spine2 (Transform)		
Position Offset	X -0.08	Y 0	Z 0.15
Rotation Offset	X 0.29	Y 0.14	Z -13.29
Destroy Trigger	✓		

Insert image 1362OT_07_40.png

11. Play the scene. Using the *WASD* keyboard control scheme, direct the character to the **PropTrigger**. Colliding with it will add the **badge** to the character.



Insert image 1362OT_07_41.png

How it works...

Once it's been triggered by the character, the script attached to the **PropTrigger** instantiates the assigned prefab, making it a child of the bones they have been "placed into". The **Position Offset** and **Rotation Offset** can be used to fine-tune the exact position of the prop (relative to its parent transform). As 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.

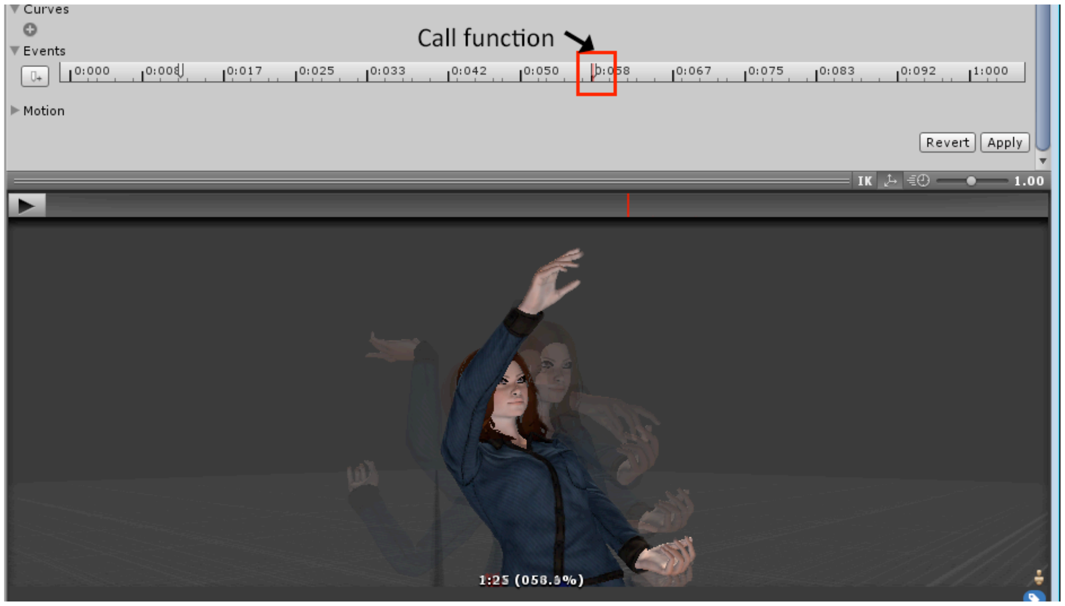
There's more...

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

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

Using Animation Events to throw an object

Now that your animated character is ready, you might want to coordinate some of her actions with her animation states. In this recipe, we will exemplify this by making the character throw an object whenever the appropriate animation clip reaches the right time. To do so, we will make use of **Animation Events**, which basically mean triggering a function from the animation clip's timeline. This feature, recently introduced to the **Mecanim** system, should feel familiar to those experienced with the **Add Event** feature of the classic **Animation** panel.



Insert image 1362OT_07_42.png

Getting ready

For this recipe, we have prepared a Unity Package named `Throwing`, containing a basic scene that features an animated character and a prefab named **EasterEgg**. The package can be found inside the `1362_07_07` folder.

How to do it...

To make an animated character throw an Easter egg (!), follow these steps:

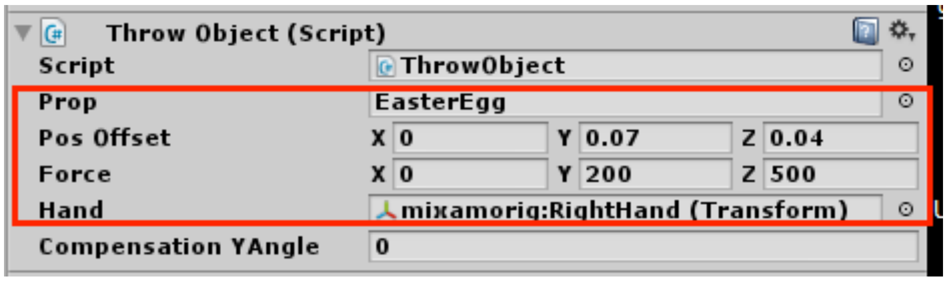
1. Create a new project and import the `Throwing` Unity Package. Then, from the **Project** view, open the **mecanim** level.
2. Play the level and press *F* on your keyboard. The character will move as if she is throwing something with her right hand.
3. From the **Project** view, create a new **C# Script** named `ThrowObject.cs.`
4. Open the script and add the following code:

```
using UnityEngine;
using System.Collections;
```

```
public class ThrowObject : MonoBehaviour {
   public GameObject prop;
   private GameObject proj;
   public Vector3 posOffset;
   public Vector3 force;
   public Transform hand;
   public float compensationYAngle = 0f;

   public void Prepare () {

      proj = Instantiate(prop, hand.position,
hand.rotation) as GameObject;
      if(proj.GetComponent<Rigidbody>())
         Destroy(proj.GetComponent<Rigidbody>());
      proj.GetComponent<SphereCollider>().enabled = false;

      proj.name = "projectile";
      proj.transform.parent = hand;
      proj.transform.localPosition = posOffset;
      proj.transform.localEulerAngles = Vector3.zero;
   }

   public void Throw () {

      Vector3 dir = transform.rotation.eulerAngles;
      dir.y += compensationYAngle;
      proj.transform.rotation = Quaternion.Euler(dir);
      proj.transform.parent = null;
      proj.GetComponent<SphereCollider>().enabled = true;

      Rigidbody rig = proj.AddComponent<Rigidbody>();
      Collider projCollider = proj.GetComponent<Collider>
();
      Collider col = GetComponent<Collider> ();
      Physics.IgnoreCollision(projCollider, col);
      rig.AddRelativeForce(force);
   }
}
```

5. Save and close the script.
6. Attach the script **ThrowObject.cs** to the character's game object, named **book**.
7. Select the **book** object. From the **Inspector** view, check out its **Throw Object** component. Then, populate the **Prop** field with the prefab named **EasterEgg**;

Populate **Hand** with **mixamorig:RightHand.** Also, change **Pos Offset** to **X**: `0`; **Y**: `0.07`; **Z**: `0.04`. Finally, change Force to **X**: `0`; **Y**: `200`; **Z**: `500`.

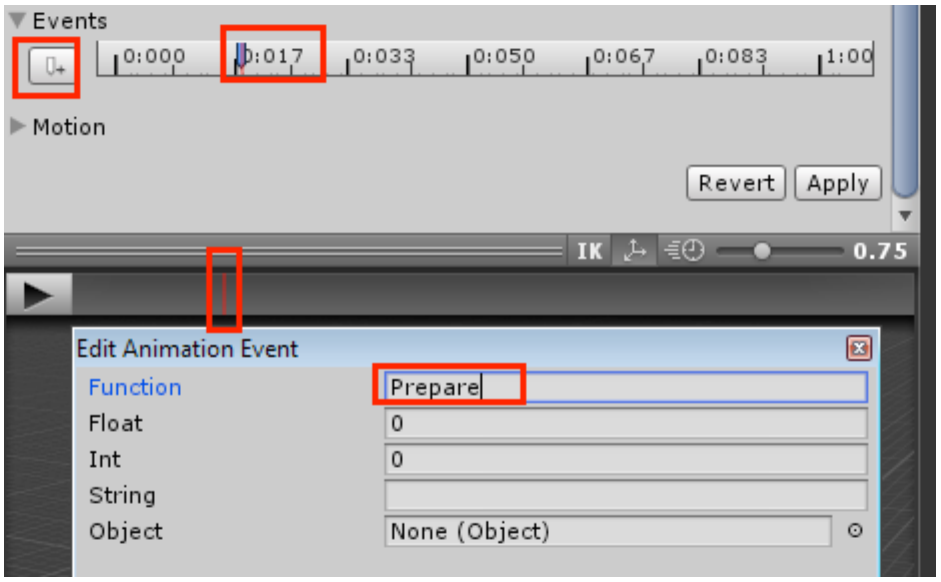


Throw Object (Script)

Script	ThrowObject
Prop	EasterEgg
Pos Offset	X 0 Y 0.07 Z 0.04
Force	X 0 Y 200 Z 500
Hand	mixamorig:RightHand (Transform)
Compensation YAngle	0

Insert image 1362OT_07_43.png

8. From the **Project** view, select the **Swat@toss_grenade** file. Then, from the **Inspector** view, access the **Animation** section and scroll down to the **Events** section.
9. Expand the **Events** section. Drag the playhead to approximately **0:17 (017.9%)** of the animation timeline. Then, click the button with the *marker +* icon to add an **Animation Event**. From the **Edit Animation Event** window, set **Function** as `Prepare`. Close the window.



Edit Animation Event

Function	Prepare
Float	0
Int	0
String	
Object	None (Object)

Insert image 1362OT_07_44.png

10. Add a new Animation Event at approximately **1:24 (057.1%)** of the animation timeline. This time, from the **Edit Animation Event** window, set **Function** as `Throw`. Close the window.
11. Click the **Apply** button to save changes.
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 the moments we have set our **Events** to, the `Prepare()` and `throw()` functions are called. The former instantiates a prefab, now named **projectile** into the character's hand (**Projectile Offset** values are used to fine-tune its position), also making it respect the character's hierarchy. Also, it disables the prefab's collider and destroys its `Rigidbody` component, provided it has one. The latter function enables the projectile's collider, adding a `Rigidbody` component to it, and making it independent from the character's hand. Finally, it adds a relative force to the projectile's `Rigidbody` component, so it will behave as if thrown by the character. The **Compensation YAngle** can be used to adjust the direction of the grenade, if necessary.

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 she 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 a Unity Package named `Ragdoll`, containing a basic scene that features an animated character and two prefabs, already placed into the scene, named **Landmine** and **Spawnpoint**. The package can be found inside the `1362_07_08` folder.

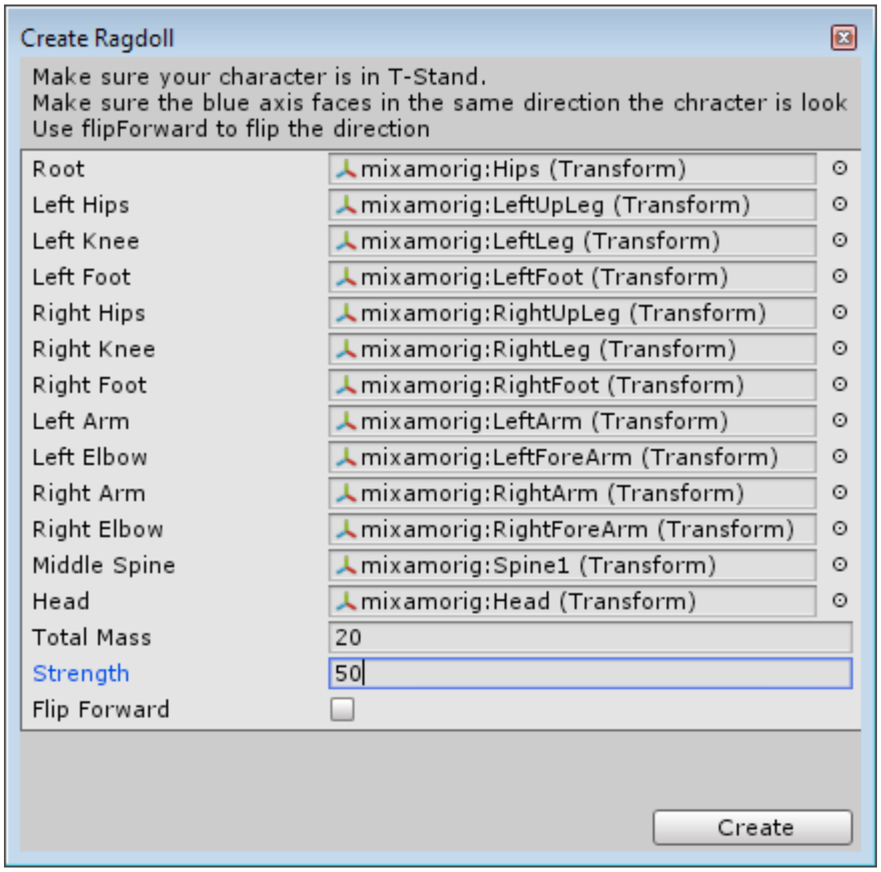
How to do it...

To apply ragdoll physics to your character, follow these steps:

1. Create a new project and import the `Ragdoll` Unity Package. Then, from the **Project** view, open the **mecanimPlayground** level.
2. You should see the animated book character and two discs: the **Landmine** and the **Spawnpoint**.

3. First, let’s set up our **Ragdoll**. Access the menu **GameObject | 3D Object | Ragdoll...** The **Ragdoll wizard** should pop-up.
4. Assign the transforms as follows:

Root	mixamorig:Hips
Left Hips	mixamorig:LeftUpLeg
Left Knee	mixamorig:LeftLeg
Left Foot	mixamorig:LeftFoot
Right Hips	mixamorig:RightUpLeg
Right Knee	mixamorig:RightLeg
Right Foot	mixamorig:RightFoot
Left Arm	mixamorig:LeftArm
Left Elbow	mixamorig:LeftForeArm
Right Arm	mixamorig:RightArm
Right Elbow	mixamorig:RightForeArm
Middle Spine	mixamorig:Spine1
Head	mixamorig:Head
Total Mass	20
Strength	50


Create Ragdoll

Make sure your character is in T-Stand.
Make sure the blue axis faces in the same direction the chracter is look
Use flipForward to flip the direction

Root	mixamorig:Hips (Transform)
Left Hips	mixamorig:LeftUpLeg (Transform)
Left Knee	mixamorig:LeftLeg (Transform)
Left Foot	mixamorig:LeftFoot (Transform)
Right Hips	mixamorig:RightUpLeg (Transform)
Right Knee	mixamorig:RightLeg (Transform)
Right Foot	mixamorig:RightFoot (Transform)
Left Arm	mixamorig:LeftArm (Transform)
Left Elbow	mixamorig:LeftForeArm (Transform)
Right Arm	mixamorig:RightArm (Transform)
Right Elbow	mixamorig:RightForeArm (Transform)
Middle Spine	mixamorig:Spine1 (Transform)
Head	mixamorig:Head (Transform)
Total Mass	20
Strength	50
Flip Forward	☐

Insert image 1362OT_07_45.png

5. From 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 {

    void Start () {
        DeactivateRagdoll();
     }

     public void ActivateRagdoll(){
       gameObject.GetComponent<CharacterController>
().enabled = false;
```

```
      gameObject.GetComponent<BasicController> ().enabled =
false;
      gameObject.GetComponent<Animator> ().enabled = false;
      foreach (Rigidbody bone in
GetComponentsInChildren<Rigidbody>()) {
             bone.isKinematic = false;
             bone.detectCollisions = true;
      }
      foreach (Collider col in
GetComponentsInChildren<Collider>()) {
             col.enabled = true;
      }
      StartCoroutine (Restore ());

    }
   public void DeactivateRagdoll(){

      gameObject.GetComponent<BasicController>().enabled =
true;
      gameObject.GetComponent<Animator>().enabled = true;
      transform.position =
GameObject.Find("Spawnpoint").transform.position;
      transform.rotation =
GameObject.Find("Spawnpoint").transform.rotation;
      foreach(Rigidbody bone in
GetComponentsInChildren<Rigidbody>()){
           bone.isKinematic = true;
           bone.detectCollisions = false;
       }
      foreach (CharacterJoint joint in
GetComponentsInChildren<CharacterJoint>()) {
          joint.enableProjection = true;
      }
      foreach(Collider col in
GetComponentsInChildren<Collider>()){
          col.enabled = false;
      }

   gameObject.GetComponent<CharacterController>().enabled =
true;

    }

   IEnumerator Restore(){
      yield return new WaitForSeconds(5);
      DeactivateRagdoll();
   }
```

```
        }
```

7. Save and close the script.
8. Attach the script **RagdollCharacter.cs** to the **book** Game Object. Then, select the **book** character and, from the top of the **Inspector** view, change its tag to **Player**.
9. From the **Project** view, create a new **C# Script** named `Landmine.cs`.
10. Open the script and add the following code:

```
        using UnityEngine;
        using System.Collections;

        public class Landmine : MonoBehaviour {
           public float range = 2f;
           public float force = 2f;
           public float up = 4f;
           private bool active = true;

           void  OnTriggerEnter ( Collider collision  ){
              if(collision.gameObject.tag == "Player" && active){
                 active = false;
                 StartCoroutine(Reactivate());

           collision.gameObject.GetComponent<RagdollCharacter>().Ac
        tivateRagdoll();
                 Vector3 explosionPos = transform.position;
                    Collider[] colliders =
        Physics.OverlapSphere(explosionPos, range);
                 foreach (Collider hit in colliders) {
                    if (hit.GetComponent<Rigidbody>())

           hit.GetComponent<Rigidbody>().AddExplosionForce(force,
        explosionPos, range, up);
                    }
                 }
            }
           IEnumerator Reactivate(){
              yield return new WaitForSeconds(2);
              active = true;
           }
        }
```

11. Save and close the script.
12. Attach the script to the **Landmine** Game Object.
13. 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 and no longer in control of its body movements, akin to a ragdoll.

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` script's function `ActivateRagdoll()`. It also applies an explosion force to our ragdoll character, throwing it outside the explosion site.

There's more...

Instead of resetting the character's transform settings, you could have destroyed its 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`.

Rotating the character's torso to aim

When playing a third person character, you might want her to aim her weapon at some target that is not directly in front of her without making her change her 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 spine by moving the mouse, allowing for adjustments in the character's aim. We will also use this opportunity to cast a ray from the character's weapon and display a crosshair over the nearest object on target. Please note that this approach will work with cameras standing behind third-person controlled characters.

Getting ready

For this recipe, we have prepared a Unity Package named `AimPointer`, containing a basic scene that features a character armed with a laser pointer. The package, which also includes the crossAim sprite, to be used as a crosshair for aiming, can be found inside the 1362_07_09 folder.

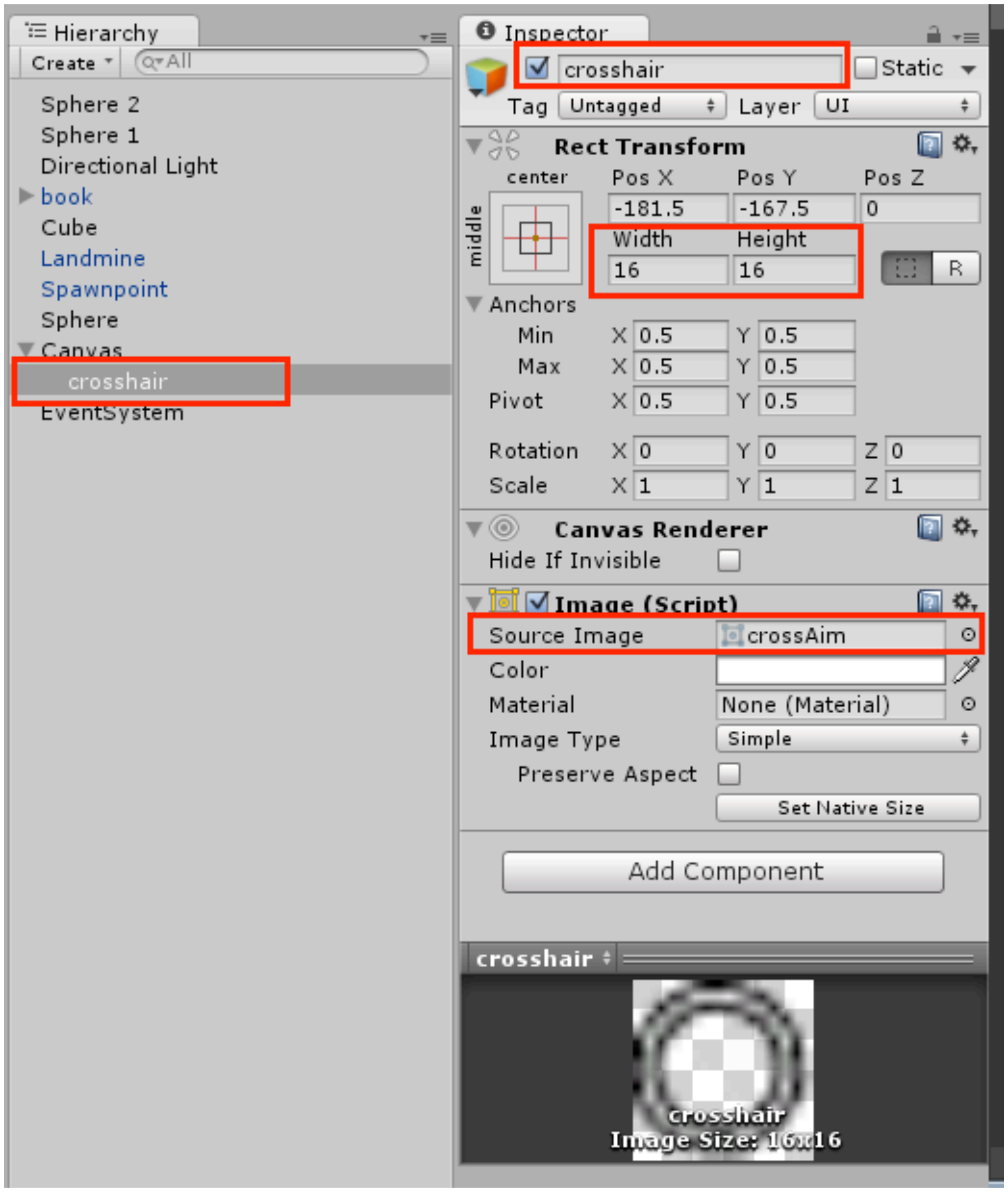
How to do it...

1. Create a new project and import the AimPointer Unity Package. Then, from the **Project** view, open the **mecanim** level. You should see an animated character named **book** holding the **pointerPrefab** object.
2. From the **Project** view, create a new **C# Script** named MouseAim.cs.
3. Open the script and add the following code:

```
using UnityEngine;
using System.Collections;

public class MouseAimClean : MonoBehaviour {

   public Transform spine;
   private float xAxis = 0f;
   private float yAxis = 0f;
   public Vector2 xLimit = new Vector2(-30f,30f);
   public Vector2 yLimit= new Vector2(-30f,30f);
   public Transform weapon;
   public GameObject crosshair;
   private Vector2 aimLoc;

   public void LateUpdate(){

      yAxis += Input.GetAxis ("Mouse X");
      yAxis = Mathf.Clamp (yAxis, yLimit.x, yLimit.y);
      xAxis -= Input.GetAxis ("Mouse Y");
      xAxis = Mathf.Clamp (xAxis, xLimit.x, xLimit.y);
      Vector3 corr = new Vector3(xAxis,yAxis,
spine.localEulerAngles.z);
      spine.localEulerAngles = corr;
      RaycastHit hit;
      Vector3 fwd =
weapon.TransformDirection(Vector3.forward);
      if (Physics.Raycast (weapon.position, fwd, out hit))
{
         print (hit.transform.gameObject.name);
         aimLoc =
Camera.main.WorldToScreenPoint(hit.point);
         crosshair.SetActive(true);
         crosshair.transform.position = aimLoc;
      } else {
```

```
                crosshair.SetActive(false);
            }
            Debug.DrawRay (weapon.position, fwd, Color.red);
        }
    }
```

4. Save and close the script.
5. From the **Hierarchy** view, create a new **UI | Image** game object. Then, from the **Inspector** view, change its name to `crosshair`. Also, in **Rect Transform**, set its **Width** and **Height** to `16` and populate **Source Image** field with the **crossAim** sprite.



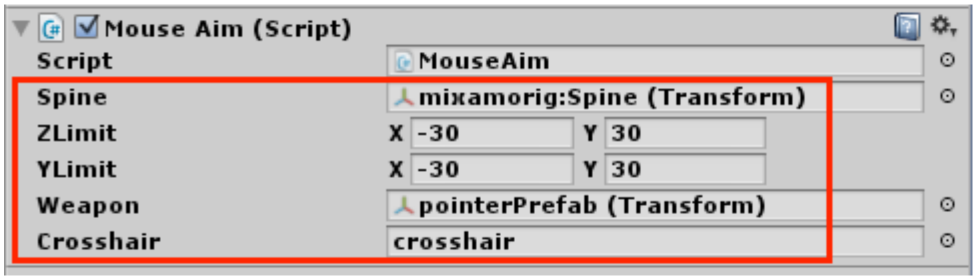
Hierarchy: Sphere 2, Sphere 1, Directional Light, book, Cube, Landmine, Spawnpoint, Sphere, Canvas, crosshair, EventSystem

Inspector: crosshair, Static
Tag Untagged, Layer UI
Rect Transform: center, middle; Pos X -181.5, Pos Y -167.5, Pos Z 0; Width 16, Height 16
Anchors: Min X 0.5 Y 0.5; Max X 0.5 Y 0.5
Pivot X 0.5 Y 0.5
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Canvas Renderer: Hide If Invisible
Image (Script): Source Image crossAim; Color; Material None (Material); Image Type Simple; Preserve Aspect
crosshair
crosshair
Image Size: 16x16

Insert image 1362OT_07_46.png

6. Attach the script **MouseAim.cs** to the **book** game object.

7. Select the **book** game object and from the **Inspector** view, **Mouse Aim** component, populate the **Spine** field with **mixamorig:Spine**; **Weapon** field with **pointerPrefab**; **Crosshair** field with the **crosshair** UI game object.

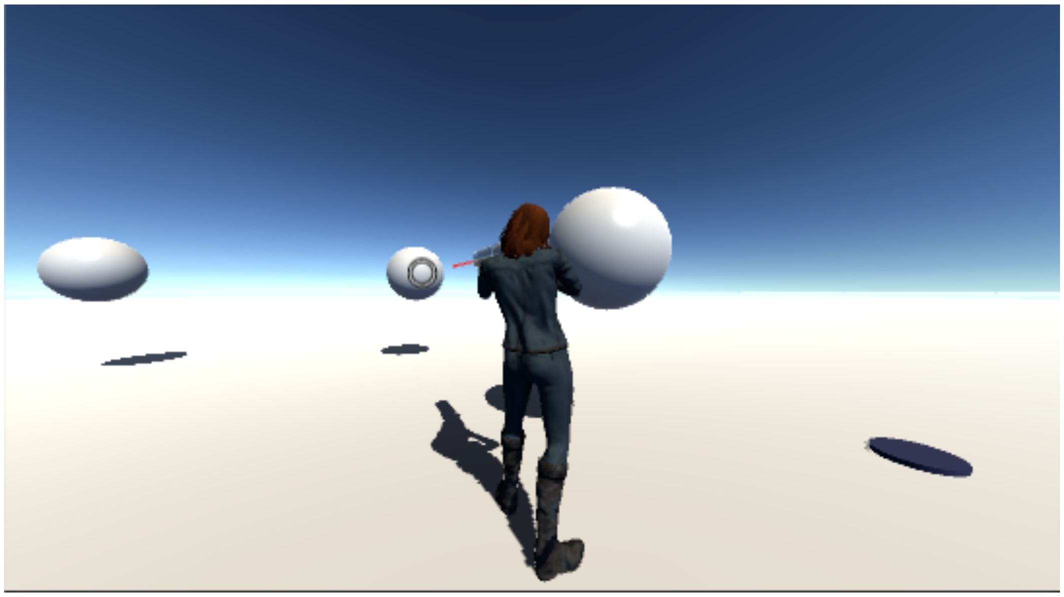


Mouse Aim (Script)

Script	MouseAim
Spine	mixamorig:Spine (Transform)
ZLimit	X -30 Y 30
YLimit	X -30 Y 30
Weapon	pointerPrefab (Transform)
Crosshair	crosshair

Insert image 1362OT_07_47.png

8. Play the scene. You should now be able to rotate the character's torso by moving the mouse. Even better, the crosshair GUI texture will be displayed on the top of the object that is being aimed at by the pointer.



Insert image 1362OT_07_48.png

How it works...

You might have noticed that all the code for rotating the character's spine is inside the `LateUpdate` function, as opposed to the more common `Update` function. The reason for that is to make sure that all the transform manipulation will be executed after the original animation clip is played, overriding it.

Regarding the spine rotation, our script adds the horizontal and vertical speed of the mouse to the `xAxis` and `yAxis` float variables. Those variables are then constrained within the specified limits, avoiding distortions to the character's model. Finally, the `spine` object transform rotation for axis X and Y are set to `xAxis` and `yAxis`, respectively. The Z axis is preserved from the original animation clip.

Additionally, our script uses a `Raycast` command to detect if there is any object's collider within the weapon's aim, in which case a crosshair will be drawn on screen.

There's more...

Since this recipe's script was tailored for cameras standing behind third-person controlled characters, we have included a more generic solution to the problem - in fact, a similar approach to the one presented in *Unity 4.x Cookbook*, *Packt Publishing*. An alternate script named `MouseAimLokkAt`, which can be found inside the `1362_07_09` folder, starts by converting our bi-dimensional mouse cursor screen's coordinates to three-dimensional world space coordinates (stored in a `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 `minY` and `maxY` angles, otherwise causing distortions to the character model. 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. Another addition is the option to freeze the X-axis rotation, in case you just want the character to rotate the torso laterally, but not look up or down. Again, this script uses a `Raycast` command to detect objects in front of the weapon's aim, drawing a crosshair on screen when they are present.