9
Playing and Manipulating Sounds

In this chapter, we will cover:

- Matching audio pitch to animation speed
- Simulating acoustic environment with Reverb Zones
- Preventing AudioClip from restarting if already playing
- Wait for audio to finish before auto-destructing Object
- Adding volume control with Audio Mixers
- Making a dynamic soundtrack with Snapshots
- Balancing in-game audio with Ducking

Introduction

Sound is a very important part of the gaming experience. In fact, we can't stress enough how crucial it is to the player's immersion in a virtual environment. Just think of the engine running in your favorite racing game, the distant urban buzz in a simulator game or the creeping noises in horror games. Think of how those sounds transport you *into* the game.

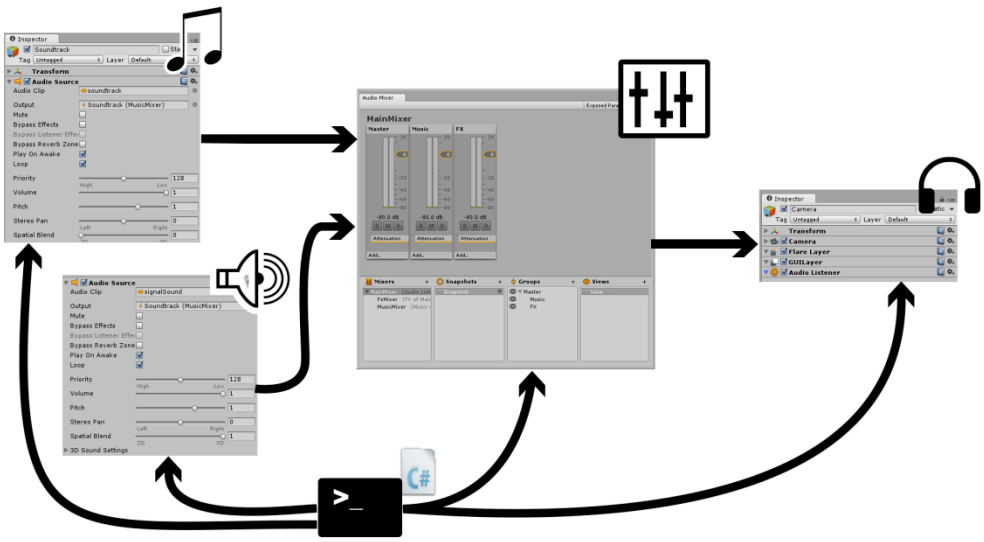
The Big Picture

Before getting on with the recipes, let's step back and have a quick review how sound works on Unity 5.

Audio files can be embedded into game objects through the **Audio Source** component. Unity support **3D sounds**, which means that the location and distance between audio sources and the **Audio Listener** matter in the way the sound is perceived in terms of

loudness and left/right balance - unless the audio source is specified as **2D sound** (which is usually the case for background soundtrack music).

Although all sound is sent to the scene's **Audio Listener** (a component that is usually attached to the **Main Camera**, and that shouldn't be attached simultaneously on more than one object), Unity 5 brings a new player to the audio scene: the **Audio Mixer**. The Audio mixer radically changes the way sound elements can be experienced and worked with. It allows developers to mix and arrange audio very much like musicians and producers do in their Digital Audio Workstations such as *GarageBand* or *ProTools*. It allows you to route audio source clips into specific channels that can have their volumes individually adjusted and processed by customized effects and filters. You can work with multiple Audio Mixers, send a mixer's output to a parent mixer, and save mix preferences as **Snapshots**. Also, you can access mixer parameters from scripting.



Insert image 1362OT_09_01.png

Taking advantage of the new Audio Mixer in many example projects, this chapter is filled with recipes that, hopefully, will help you implement a better and more efficient sound design for your projects, augmenting the player's sense of immersion, transporting him into the game environment, and even improving gameplay.

Matching audio pitch to animation speed

Many artifacts sound higher in pitch when accelerated and lower when slowed down. Car engines, fan coolers, Vinyl - a record player… the list goes on. If you want to simulate

this kind of sound effect in an animated object that can have its speed changed dynamically, follow this recipe.

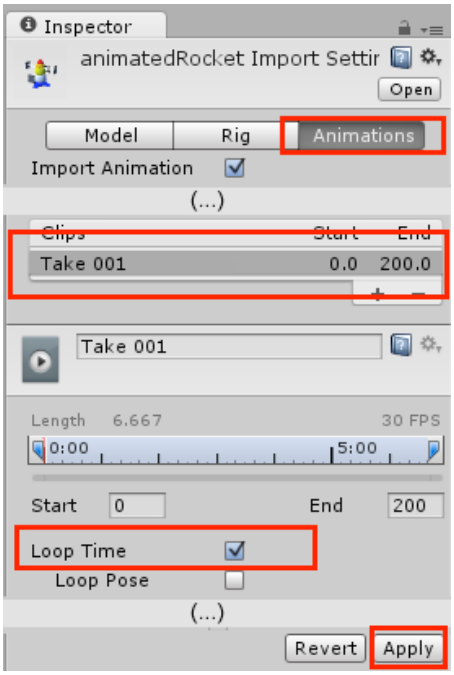
Getting ready

For this recipe, you'll need an animated 3D object and an audio clip. Please use the files `animatedRocket.fbx` and `engineSound.wav`, available in the `1362_09_01` folder.

How to do it...

To change the pitch of an audio clip according to the speed of an animated object, please follow these steps:

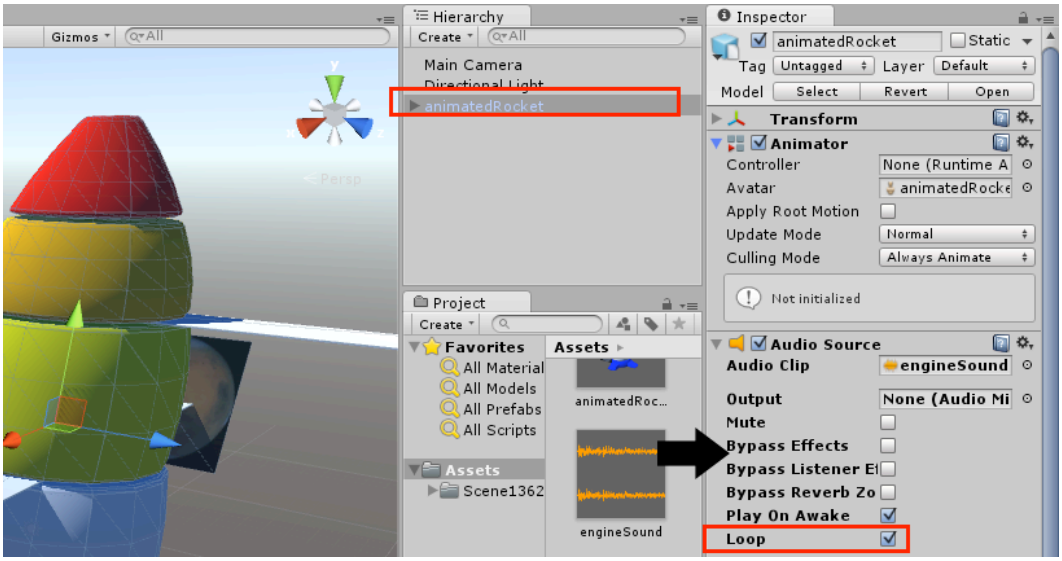
1. Import the file `animatedRocket.fbx` file into your Project.
2. Select the `carousel.fbx` file in the **Project** view. Then, from the **Inspector** view, check its **Import Settings**. Select **Animations,** then select the clip **Take 001** and make sure to check the option **Loop Time**. Click the button **Apply** to save changes.



Insert image 1362OT_09_02.png

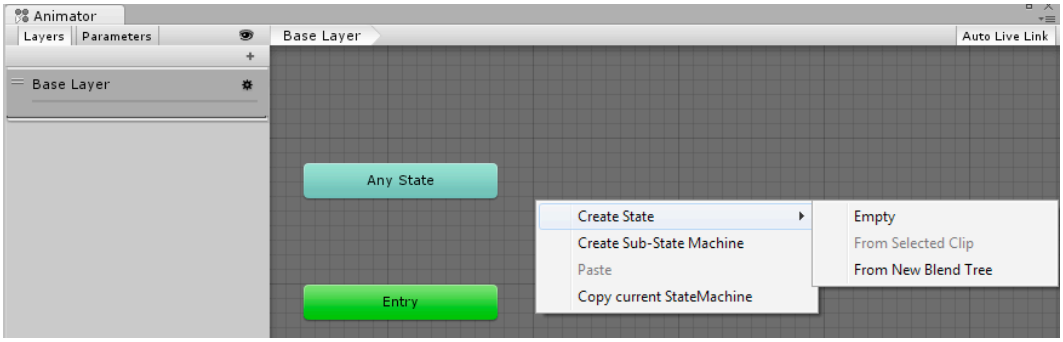
The reason why we didn't need to check **Loop Pose** is because our animation already loops in a seamless fashion. If it didn't, we could have checked that option to automatically create a seamless transition from the last to the first frame of the animation.

3. Add the **animatedRocket** to the scene by dragging it from the **Project** view into the **Hierarchy** view.
4. Import the audio clip `engineSound.wav`.
5. Select the Game Object **animatedRocket**. Then, drag **engineSound** from the **Project** view into the **Inspector** view, adding it as an **Audio Source** for that object.
6. In the **Audio Source** component of the **carousel**, check the box for the **Loop** option.



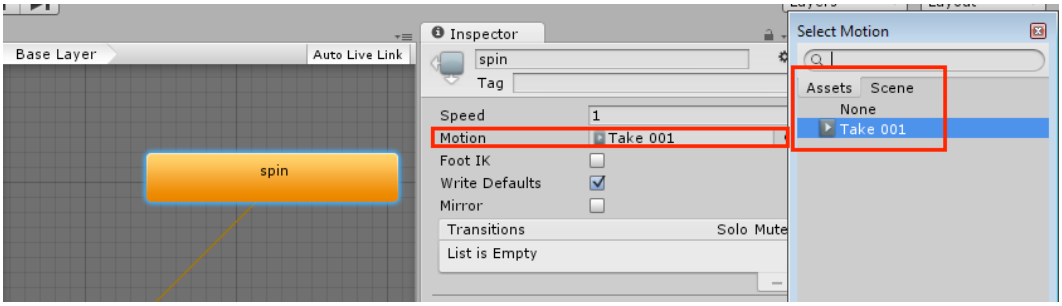
Insert image 1362OT_09_03.png

7. We need to create a **Controller** for our object. In the **Project** view, click the **Create** button and select **Animator Controller**. Name it `rocketController`.
8. Double-click **rocketController** to open the **Animator** window. Then, right-click the gridded area and select the option **Create State | Empty**, on the contextual menu.



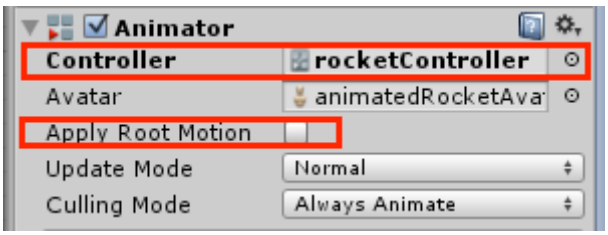
Insert image 1362OT_09_04.png

9. Name the new state `spin` and set **Take 001** as its motion in the **Motion** field.



Insert image 1362OT_09_05.png

10. From the **Hierarchy** view, select the **animatedRocket**. Then, in the **Animator** component (in the **Inspector** view), set **rocketController** as its **Controller** and make sure the option **Apply Root Motion** is unchecked.



Insert image 1362OT_09_06.png

11. In the **Project** view, create a new **C# Script** and rename it to `ChangePitch`.
12. Open the script in your editor and replace everything with the following code:

```
using UnityEngine;

public class ChangePitch : MonoBehaviour{

   public float accel = 0.05f;
   public float minSpeed = 0.0f;
   public float maxSpeed = 2.0f;
   public float animationSoundRatio = 1.0f;
   private float speed = 0.0f;
   private Animator animator;
   private AudioSource audioSource;

   void Start(){
      animator =   GetComponent<Animator>();
      audioSource = GetComponent<AudioSource>();
      speed = animator.speed;
      AccelRocket (0f);
   }

   void Update(){
      if (Input.GetKey (KeyCode.Alpha1))
         AccelRocket(accel);

      if (Input.GetKey (KeyCode.Alpha2))
         AccelRocket(-accel);
   }

   public void AccelRocket(float accel){
      speed += accel;
      speed = Mathf.Clamp(speed,minSpeed,maxSpeed);
   animator.speed = speed;
      float soundPitch = animator.speed * animationSoundRatio;
      audioSource.pitch = Mathf.Abs(soundPitch);
   }
}
```

13. Save your script and add it as a component to the **animatedRocket**.
14. Play the scene and change the animation speed by pressing *1* (accelerate) and *2* (decelerate) on your alphanumeric keyboard. The audio pitch will change accordingly.

How it works...

At `Start()`, besides storing the **Animator** and **AudioSource** components in variables, we get the initial `speed` from the Animator and call the function `AccelRocket()`, passing `0` as argument, only for that function to calculate the resulting **pitch** for the AudioSource. During `Update()`, the lines of code `if(Input.GetKey (KeyCode.Alpha1))` and `if(Input.GetKey (KeyCode.Alpha2))` detect whenever the keys *1* or *2* are being pressed on the alphanumeric keyboard to call the `AccelRocket()` function, passing as argument a float variable `accel`. The `AccelRocket()` function, in its turn, increments `speed` with the received argument (the float variable `accel`) . However, it uses the command `Mathf.Clamp()` to limit the new speed value between the minimum and maximum speed as set by the user. Then, it changes the **Animator** speed and **AudioSource** pitch according to the new `speed` absolute value (the reason for making it an absolute value is keeping the pitch a positive number even when the animation is reversed by a negative speed value). Also, please note that setting the animation speed, and therefore the sound pitch, to `0` will cause the sound to stop, making it clear that stopping the object's animation also prevents the engine sound from playing.

There's more...

Here is some information on how to fine tune and customize this recipe:

Changing the Animation / Sound Ratio

If you want the audio clip pitch to be either more or less affected by the animation speed, change the value of the **Animation / Sound Ratio** parameter.

Accessing the function from other scripts

The `AccelRocket()` function was made public so it can be accessed from other scripts. As an example, we have included the script `ExtChangePitch.cs` in `1362_09_01` folder. Try attaching this script to the **Main Camera** object and use it to control the speed by clicking on the left and right mouse buttons.

Simulating acoustic environments with Reverb Zones

Once you have created your level's geometry, and the scene is looking just the way you want it to, you might want your sound effects to correspond to that look. Sound behaves differently depending upon the environment in which it is projected, so it can be a good idea to make it reverberate accordingly. In this recipe, we will address this acoustic effect by using **Reverb Zones**.

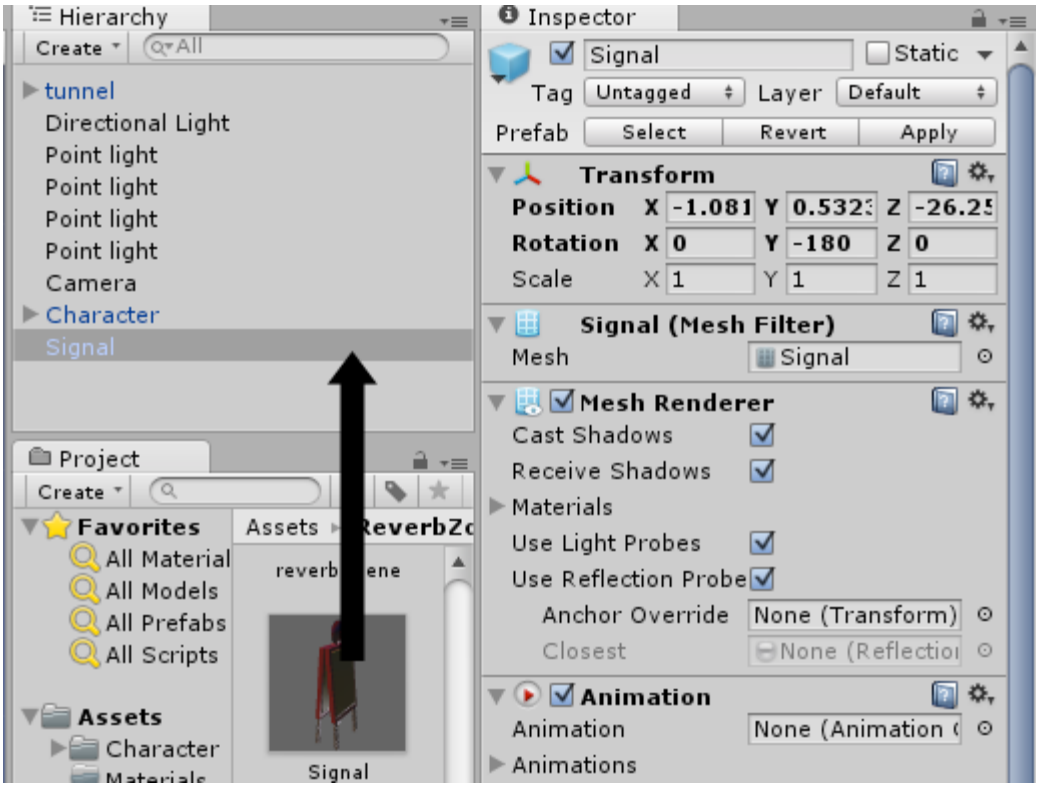
Getting ready

For this recipe, we have prepared the `ReverbZone.unitypackage` file, containing a basic level named `reverbScene` and the `Signal` prefab. The package is in the folder `1362_09_02`.

How to do it...

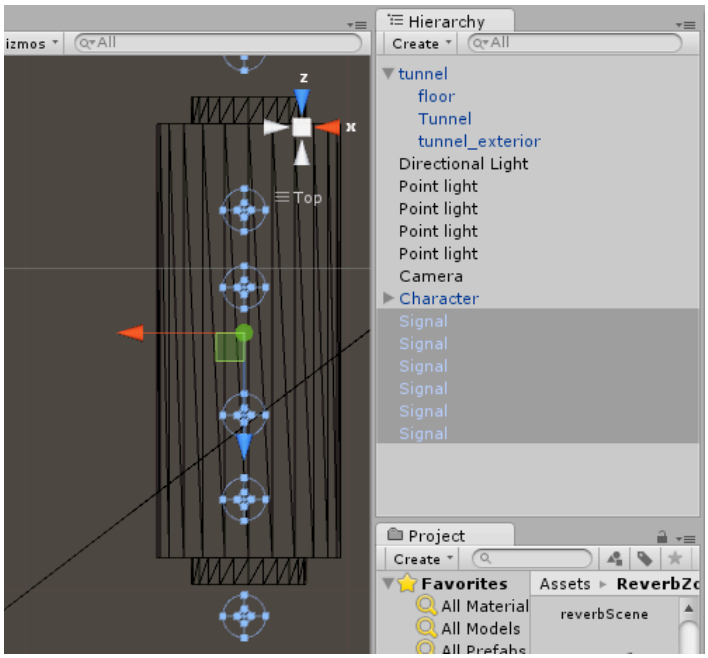
Follow these steps to simulate the sonic landscape of a tunnel:

1. Import the package `ReverbZone` into your Unity Project.
2. In the **Project** view, open the **reverbScene** level, inside the **ReverbZones** folder. This is a basic scene featuring a controllable character and a tunnel.
3. Now, drag the **Signal** prefab from the **Project** view into the **Hierarchy**. Thais should add a sound-emitting object to the scene. Place it in the center of the tunnel.



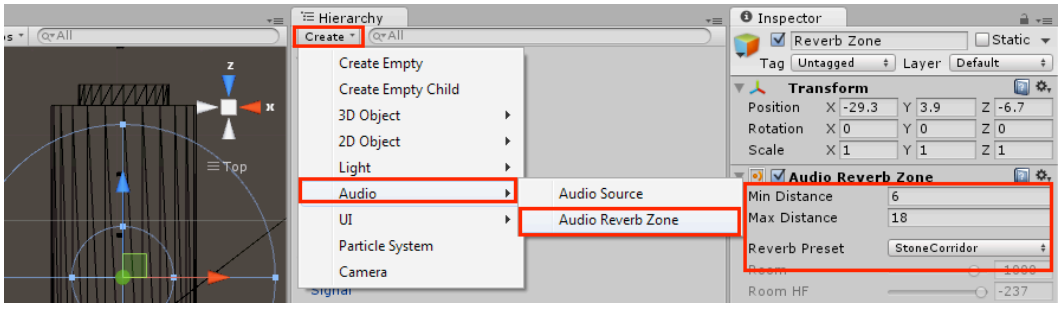
Insert image 1362OT_09_07.png

4. Make five copies of the **Signal** Game Object and distribute them across the tunnel (leaving a copy just outside each entrance).



Insert image 1362OT_09_08.png

5. In the **Hierarchy** view, click **Create | Audio | Audio Reverb Zone** to add a **Reverb Zone** to the scene. Then, place it in the center of the tunnel.
6. Select the **Reverb Zone** Game Object. In the **Inspector** view, change the **Reverb Zone** component parameters to these values**: Min Distance: 6***; Max Distance: 18**; Preset: StoneCorridor*.



Insert image 1362OT_09_09.png

7. Play the scene and walk through the tunnel using the *WASD* keys (pressing *SHIFT* to run). You should hear the audio reverberate when inside the **Reverb Zone** area.

How it works...

Once positioned, the **Audio Reverb Zone** applies an audio filter to all audio sources within its radius.

There's more...

More options for you to try:

Attaching the Audio Reverb Zone Component to Audio Sources

Instead of creating an **Audio Reverb Zone** Game Object, you could attach it to the sound emitting object (in our case, the **Signal**) as a component through the menu **Component | Audio | Audio Reverb Zone**. In this case, the Reverb Zone would be individually set up around the object.

Making your own Reverb settings

Unity comes with several **Reverb Presets**. We have used **StoneCorridor**, but your scene could ask for something less intense (like **Room**) or more radical (like **Psychotic**). If those presets still won't be able to recreate the effect you have in mind, change it to **User** and edit its parameters as you wish.

Preventing AudioClip from restarting if already playing

In a game, there may be several different events that cause a sound to start playing. If the sound is already playing, then in almost all cases, we don't wish to restart the sound. This recipe includes a test, so that an **AudioSource** component is only sent a `Play()` message if it is currently not playing.

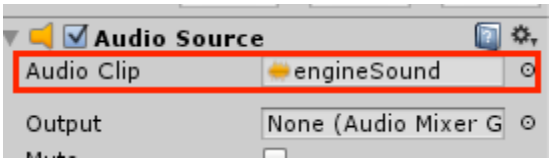
Getting ready

Try this with any audio clip that is one second or longer in duration. We have included the `engineSound` audio clip inside the `1362_09_03`.

How to do it...

To prevent an AudioClip from restarting, follow these steps:

1. Create an empty GameObject and rename it to **AudioObject**. Then, add an **Audio Source** component to this object (in the menu **Component | Audio | Audio Source**).
2. Import the `engineSound` audio clip and drag it from the **Project** view to populate the **Audio Clip** parameter of the **Audio Source** component of **AudioObject**.

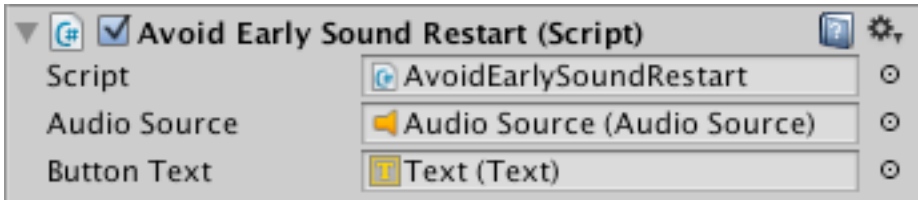


Insert image 1362OT_09_10.png

3. Create a UI button named **PlaySoundButton** on screen, and attach the following script to this button:

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

public class AvoidEarlySoundRestart : MonoBehaviour {
    public AudioSource audioSource;
    public Text message;

    void Update(){
        string statusMessage = "Play sound";
        if(audioSource.isPlaying )
            statusMessage = "(sound playing)";

        message.text = statusMessage;
    }

    // button click handler
    public void PlaySoundIfNotPlaying(){
        if( !audioSource.isPlaying )
            audioSource.Play();
    }
}
```

4. With **PlaySoundButton** selected in the **Hierarchy**, drag **AudioObject** into the **Inspector** for the public **AudioSource** variable, and drag the **Text** child of **PlaySoundButton** for the public **ButtonText:**



Insert image 1362OT_09_11.png

5. With **PlaySoundButton** selected in the **Hierarchy**, create a new On Click event handler, dragging the **PlaySoundButton** into the Object slot, and selecting the function **PlaySoundIfNotPlaying().**

How it works...

Audio Source components have a public readable property **isPlaying**, which is a Boolean true/false flag indicating if the sound is currently playing. The text of the button is set to display "`Play Sound`" when the sound is not playing, and "(`sound playing`)" when it is. When the button is clicked, the method `PlaySoundIfNotPlaying()` is called. This method uses an "IF" statement ensuring that a `Play()` message is only sent to the Audio Source component if its **isPlaying** is false.

See also

We will see more details in the next recipe.

Waiting audio to finish before auto-destructing Object

An event may occur (such as an object pickup or the killing of an enemy) that we wish to notify to the player by playing an audio clip and an associated visual object (such as an explosion particle system, or a temporary object in the location of the event). However, as soon as the clip has finished playing, we wish the visual object to be removed from the scene. This recipe provides a simply way to link the ending of a playing audio clip with the automatic destruction of its containing object.

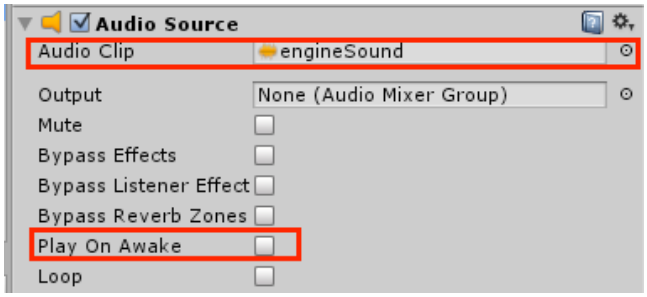
Getting ready

Try this with any audio clip that is one second or longer in duration. We have included the `engineSound` audio clip inside the `1362_09_04`.

How to do it...

To wait for the audio to finish before destroying an Object, follow these steps

1. Create an empty GameObject and rename it to **AudioObject**. Then, add an **Audio Source** component to this object (in the menu **Component | Audio | Audio Source**).
2. Import the `engineSound` audio clip and drag it from the **Project** view to populate the **AudioClip** parameter of the **AudioSource** component of **AudioObject**, and de-select the component's **Play On Awake** checkbox.



Insert image 1362OT_09_12.png

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

```
using UnityEngine;
using System.Collections;

public class AudioDestructBehaviour : MonoBehaviour {
   private AudioSource audioSource;

   void Start(){
      audioSource = GetComponent<AudioSource>();
   }

   private void Update(){
      if( !audioSource.isPlaying )
         Destroy(gameObject);
   }
}
```

4. In the **Inspector**, now disable (un-check) scripted component `AudioDestructBehaviour` of **AudioObject** (when needed, it will be re-enabled via C# code).

Insert image 1362OT_09_13.png

5. Create a new C# file named **ButtonActions** containing the following code:

```
using UnityEngine;
using System.Collections;

public class ButtonActions : MonoBehaviour{
   public AudioSource audioSource;
   public AudioDestructBehaviour audioDestructScriptedObject;

   public void PlaySound(){
      if( !audioSource.isPlaying )
         audioSource.Play();
   }

   public void DestroyAfterSoundStops(){
      audioDestructScriptedObject.enabled = true;
   }
}
```

6. Create a UI button named **PlaySoundButton** on screen, with button text `Play Sound`, and attach the **ButtonActions** script to this button.
7. With **PlaySoundButton** selected in the **Hierarchy**, create a new On Click event handler, dragging the **PlaySoundButton** into the Object slot, and selecting the function **PlayMessage().**
8. With the **PlaySoundButton** selected in the **Hierarchy**, drag **AudioObject** into the **Inspector** for the public **Audio Source** variable **AudioObject**. Also, drag **AudioObject** into the **Inspector** for the public **Script** variable **AudioDestructScriptedObject.**

Insert image 1362OT_09_14.png

9. Create a second UI button named **DestoryWhenSoundFinishedButton** on screen, with button text `Destroy When Sound Finished`, and attach the **ButtonActions** script to this button.
10. With **DestoryWhenSoundFinishedButton** selected in the **Hierarchy**, create a new **On Click** event handler, dragging the **PlaySoundButton** into the **GO** slots, and selecting function **SendAudioSourceDestroyAfterPlayingMessage()**.
11. Just as you did with the other button, now the **DestoryWhenSoundFinishedButton** selected in the **Hierarchy**, drag **AudioObject** into the **Inspector** for the public **Script** variable **MyAudioDestructObect**.

How it works...

The GameObject named **AudioObject** contains an Audio Source component, which stores and manages the playing of audio clips. **AudioObject** also contains a scripted component, which is an instance of the class **AudioDestructBehaviour**. This script is initially disabled. When enabled, every frame this object (via its `Update()` method) tests whether the audio source is not playing (`!audio.isPlaying`). As soon as the audio is found to be not playing, then the GameObect is destroyed.

There are two UI buttons created. Button **PlaySoundButton** calls the method `PlaySound()`. This method will start playing the audio clip, if it is not already playing.

The second button **DestoryWhenSoundFinishedButton** calls the method `DestoryAfterSoundStops()`. This method enables the scripted component **AudioDestructBehaviour** in GameObject **AudioObject** - so that that GameObject will be destroyed once the sound has finished playing.

See also

- *Preventing AudioClip from restarting if already playing*

Adding volume control with Audio Mixers

Sound volume adjustment can be a very important feature, especially if your game is a standalone. After all, it can be very frustrating having to access the operational system volume control. In this recipe, we will use the mew **Audio Mixer** feature to create independent volume controls for Music and Sound FX.

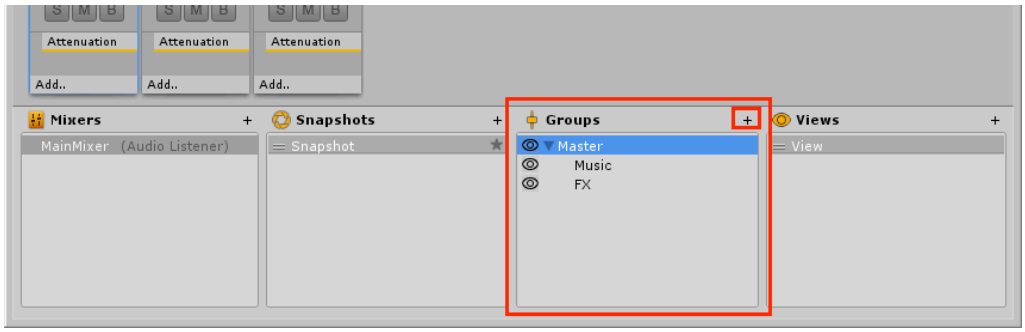
Getting ready

For this recipe, we have provided a Unity package named `Volume.unitypackage`, containing an initial scene featuring soundtrack music and sound effects. The file is available inside the `1362_09_05` folder.

How to do it...

To add volume control sliders to your scene, follow these steps:

1. Import `Volume.unitypackage` into your project.
2. Open the scene **Volume** (available in the folder **Assets | Volume**). Play the scene and walk towards the semitransparent green wall in the tunnel, using the *WASD* keys (pressing *SHIFT* to run). You should be able to listen to:
    - A looping soundtrack music.
    - Bells ringing.
    - A robotic speech whenever the character collides with the wall.
3. From the **Project** view, use the **Create** dropdown menu to add an **Audio Mixer** to the project. Name it **MainMixer**. Double click it to open the **Audio Mixer** window.
4. From the **Groups** view, highlight **Master** and click the + sign to add a child to the **Master** group. Name it **Music**. Then, highlight **Master** again and add a new child group named **FX**.

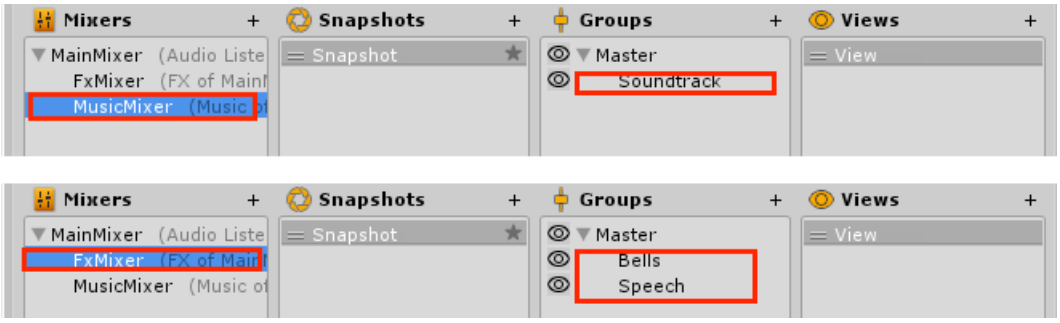


Insert image 1362OT_09_15.png

5. From the **Mixers** view, highlight **MainMixer** and click the + sign to add a new **Mixer** to the project. Name it **MusicMixer**. Then, drag it into the **MainMixer** and select the group **Music** as its **Output**. Repeat the operation to add a mixer named **FxMixer** to the project, selecting the **FX** group as output.

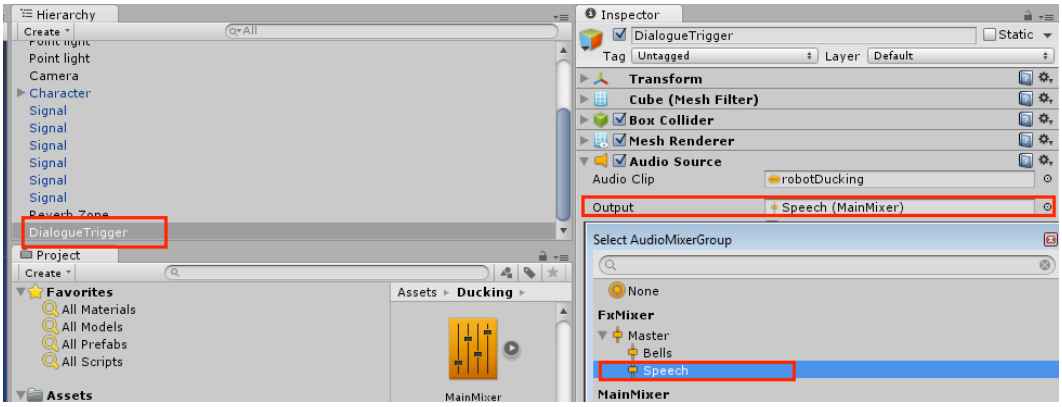
Insert image 1362OT_09_16.png

6. Now select the **MusicMixer**. Select its **Master** group and add a child named **Soundtrack**. Then, select **FxMixer** and add two children to its **Master** group: one named **Speech**, and another named **Bells**.



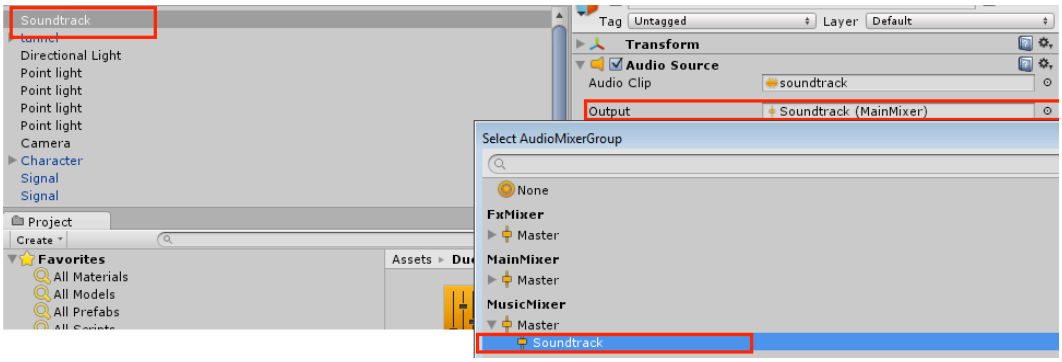
Insert image 1362OT_09_17.png

7. From the **Hierarchy** view, select the **DialogueTrigger** object. Then, in the **Inspector** view, **Audio Source** component, Change its **Output** track to **FxMixer | Speech**.



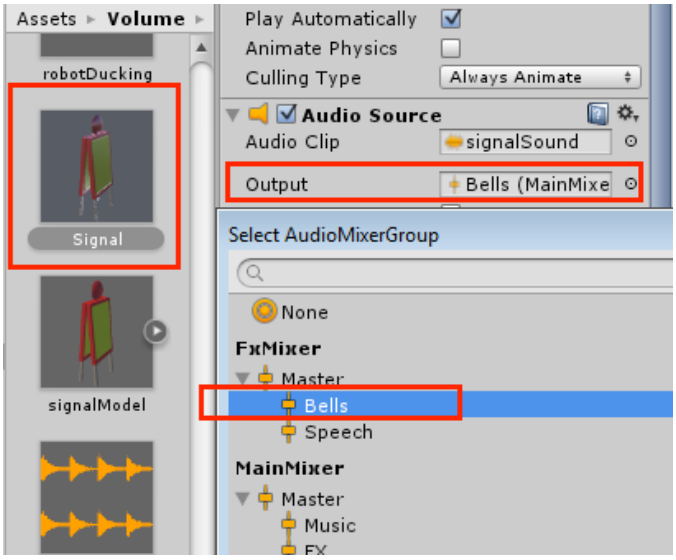
Insert image 1362OT_09_18.png

8. Now, select the **Soundtrack** game object. From the **Inspector** view, find the **Audio Source** component and change its **Output** track to **MusicMixer | Soundtrack**.



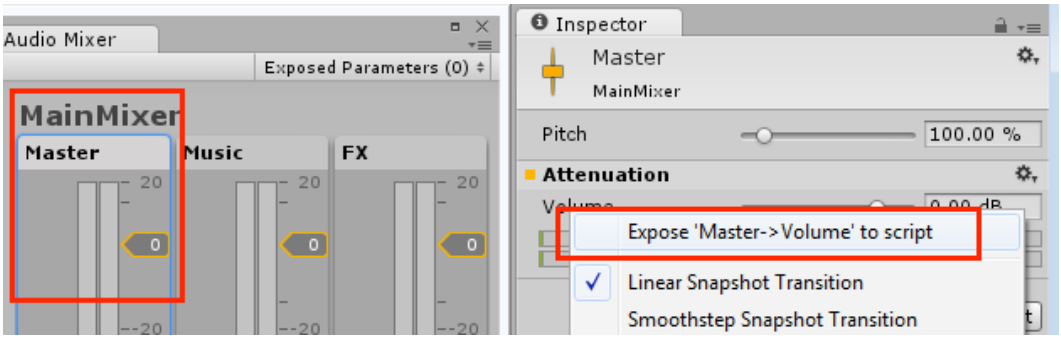
Insert image 1362OT_09_19.png

9. Finally, from the **Assets** folder in the **Project** view, select the **Signal** prefab and, from the **Inspector** view, access its **Audio Source** component and change its **Output** to **FxMixer | Bells**.



Insert image 1362OT_09_20.png

10. From the **Audio Mixer** window, choose **MainMixer** and select its **Master** track. Then, from the **Inspector** view, right-click **Volume**, in the **Attenuation** component, and from the context menu, select **Expose Master -> Volume to script**. Repeat the operation for the **Music and FX** tracks.



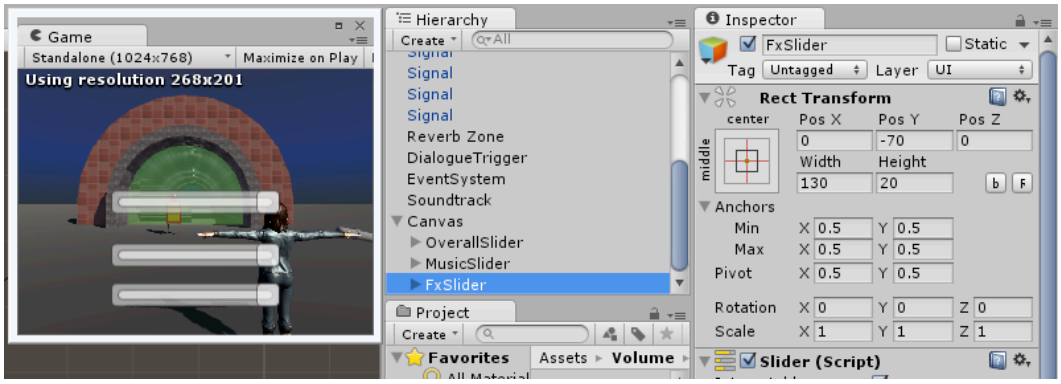
Insert image 1362OT_09_21.png

11. From the top of the **Audio Mixer**, with the **MainMixer** selected, access the dropdown menu **Exposed Parameters**, right-click **MyExposedParam** and rename it to `OverallVolume`. Then, rename **MyExposedParam1** as `MusicVolume` and **MyExposedParam2** as `FxVolume`.
12. From the **Project** view, create a new **C# Script** and rename it to `VolumeControl`.
13. Open the script in your editor and replace everything with the following code:

```
using UnityEngine;
using UnityEngine.Audio;
using UnityEngine.EventSystems;
using System.Collections;

public class VolumeControl : MonoBehaviour{
    public AudioMixer myMixer;
    private Canvas canvas;
    private EventSystem eventSystem;

    void Start(){
        canvas = gameObject.GetComponent<Canvas> ();
        eventSystem = GameObject.Find
("EventSystem").GetComponent<EventSystem> ();
        canvas.enabled = false;
        eventSystem.enabled = false;
    }
    void Update() {
```

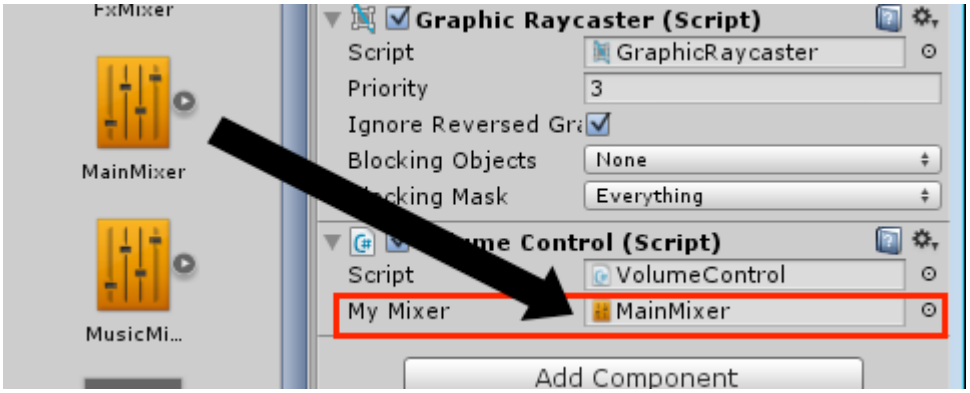
```
        if (Input.GetKeyUp (KeyCode.Escape)) {
            canvas.enabled = !canvas.enabled;
            eventSystem.enabled = !eventSystem.enabled;
        }
    }
    public void ChangeMusicVol(float vol){
        myMixer.SetFloat ("MusicVolume", Mathf.Log10(vol) * 20f);
    }
    public void ChangeFxVol(float vol){
        myMixer.SetFloat ("FxVolume", Mathf.Log10(vol) * 20f);
    }
    public void ChangeOverallVol(float vol){
        myMixer.SetFloat ("OverallVolume", Mathf.Log10(vol) * 20f);
    }
}
```

14. From the **Hierarchy** view, use the **Create** dropdown menu to add a **Slider** to the scene (**Create | UI | Slider**). Note that it will automatically add a **Canvas** to the scene.
15. Rename the slider as **OverallSlider**. Duplicate it and rename the new copy to **MusicSlider**. Then, in the **Inspector** view, **Rect Transform** component, change its **Pos Y** parameter to `-40`.
16. Duplicate **MusicSlider** and rename the new copy to **FxSlider**. Then, change its **Pos Y** parameter to `-70`.



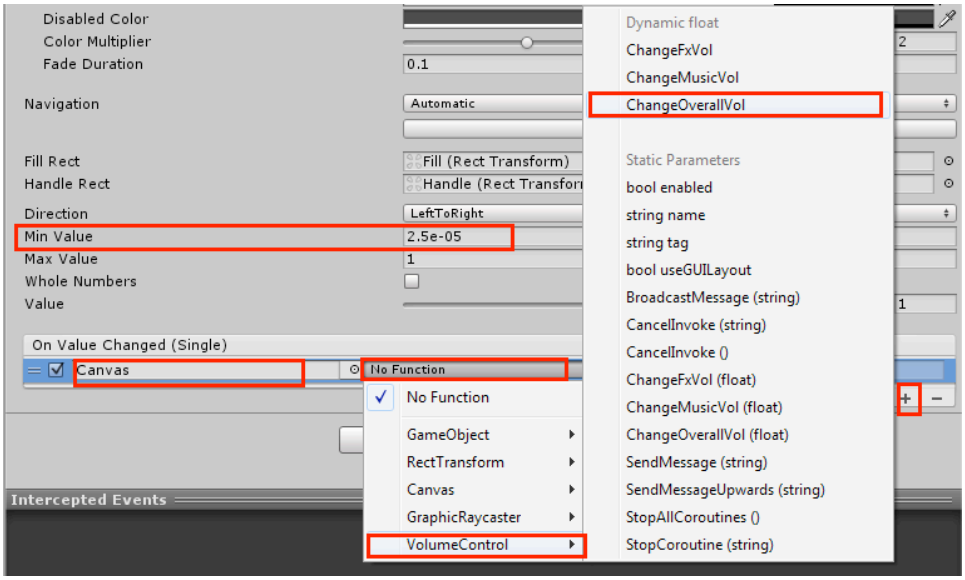
Insert image 1362OT_09_22.png

17. Select the **Canvas** game object and add the **VolumeControl** script to it. Then, populate the **MyMixer** field of **Volume Control** with the **MainMixer**.



Insert image 1362OT_09_23.png

18. Select the **OverallSlider**. From the **Inspector** view, **Slider** component, change the **Min Value** to `0.000025` (or **2.5e-05**). Then, below the **On Value Changed** list, click the **+** sign to add an action. From **Hierarchy**, drag **Canvas** into the **Object** slot and, using the function dropdown menu, choose **VolumeControl | ChangeOverallVolume**.



Insert image 1362OT_09_24.png

19. Repeat the previous step with **MusicSlider** and **FxSlider**, except this time choosing **ChangeMusicVol** and **ChangeFxVol**, respectively, from the dropdown menu.
20. Play the scene. You should be able to access the sliders when pressing *Escape* on your keyboard and adjust volume settings from there.

How it works...

The new **Audio Mixer** works in a similar fashion to Digital Audio Workstations such as Logic and Sonar. Through Audio Mixers, you can organize and manage Audio elements by routing them into specific groups that can have individual audio tracks to be tweaked around, allowing for adjustments in volume level and sound effects.

By organizing and routing our audio clips into two groups (**Music** and **FX**), we established the **MainMixer** as a unified controller for volume. Then, we have used the **Audio Mixer** to expose the volume levels for each track of the **MainMixer**, making them accessible to our script.

Also, we have set up a basic GUI featuring three sliders that, when in use, will pass their float values (between 0.000025 and 1) as arguments to three specific functions in our script: `ChangeMusicVol`, `ChangeFxVol`, and `ChangeOverallVol`. Those functions, on their turn, use the `SetFloat` command to effectively change the volume levels at runtime. However, before passing on the new volume levels, the script converts linear values (between 0.000025 and 1) to decibel levels used by the Audio Mixer. This conversion is calculated through the mathematical function *log(x) * 20*.

> For a full explanation on issues regarding the conversion of linear values to decibel levels and vice-versa, check out Aaron Brown's excellent article at `http://www.playdotsound.com/portfolio-item/decibel-db-to-float-value-calculator-making-sense-of-linear-values-in-audio-tools/`.

It's worth mentioning that the `VolumeControl` script also includes code to enable and disable the **GUI** and the **EventSystem**, depending upon whether the player hits the Escape key to activate/deactivate the volume control sliders.

A very important note: do not change the volume of any MainMixer's tracks; leave them at 0 dB. The reason is that our `VolumeControl` script sets 0 as their maximum volume level. For general adjustments, use the secondary Mixers MusicMixer and FxMixer.

There's more...

Here is some extra information on Audio Mixers:

Playing with Audio Production

There are many creative uses for exposed parameters. We could, for instance, add effects such as **Distortion**, **Flange**, and **Chorus** to audio channels, allowing users to operate virtual sound tables / mixing boards.

See also

- *Making a dynamic soundtrack with Snapshots*
- *Balancing soundtrack volume with Ducking*

Making a dynamic soundtrack with Snapshots

Dynamic soundtracks are the ones that change according to what is happening to the player in the game, musically reflecting that place or moment of the character's adventure. In this recipe, we will implement a soundtrack that changes twice: the first time when entering a tunnel, and the second time when coming out of its end. To achieve that, we will use the new **Snapshot** feature of the **Audio Mixer**.

Snapshots are a way of saving the state of your Audio Mixer, keeping your preferences for volume levels, audio effects, and more. We can access those states through script, creating transitions between mixes, bringing up the desired sonic ambience for each moment of the player's journey.

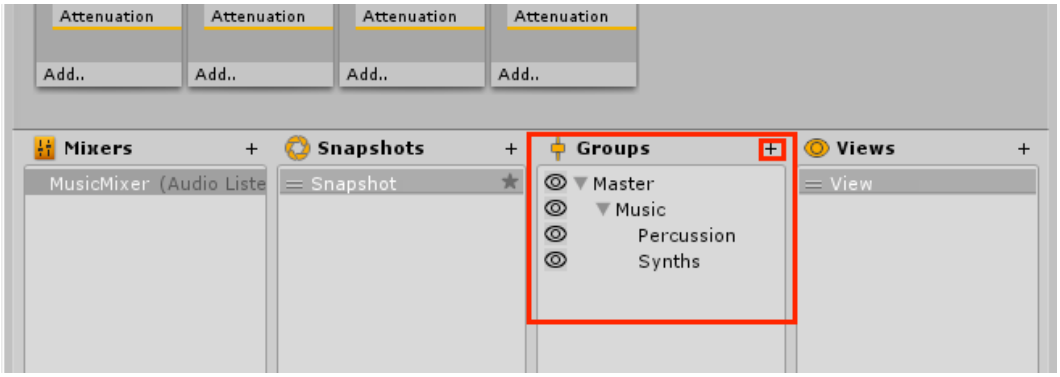
Getting ready

For this recipe, we have prepared a basic level, contained inside the Unity package named `DynamicSoundtrack`, and two soundtrack audio clips in `.ogg` format: `Theme01_Percussion` and `Theme01_Synths`. All files can be found in the `1362_09_07` Folder.

How to do it...

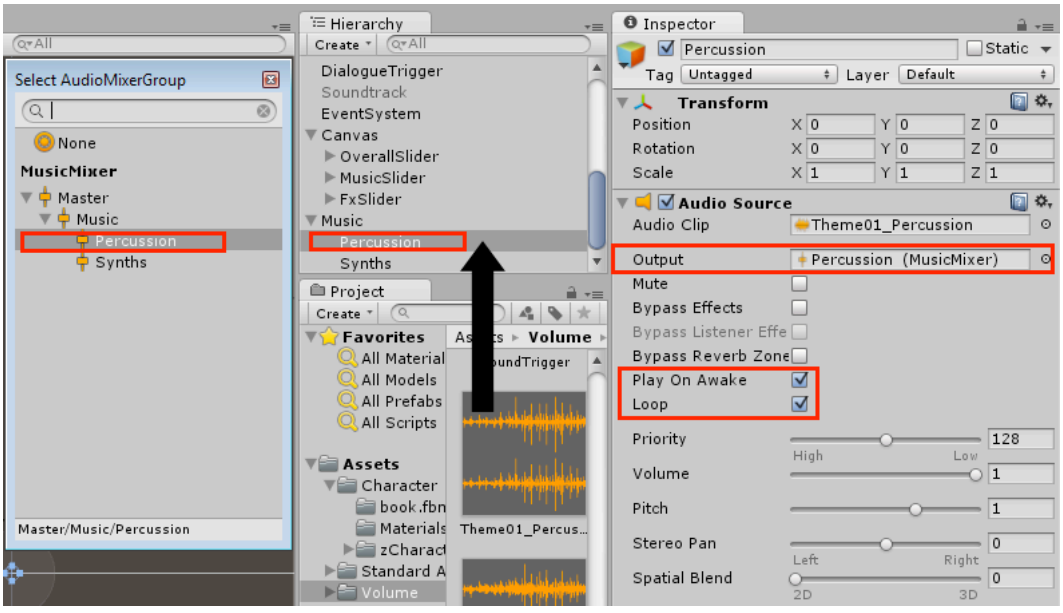
To make a dynamic soundtrack, follow these steps:

1. Import the package `DynamicSoundtrack` into your Unity Project.
2. Open the level named **DynamicSoutrack**.
3. From the **Project** view, use the **Create** dropdown menu to add an **Audio Mixer** to the project. Name it **MusicMixer**. Double click it to open the **Audio Mixer** window.
4. From the **Groups** view, highlight **Master** and click the **+** sign to add a child to the **Master** group. Name it as **Music**. Then, add two child groups to **Music**: **Percussion** and **Synths**.



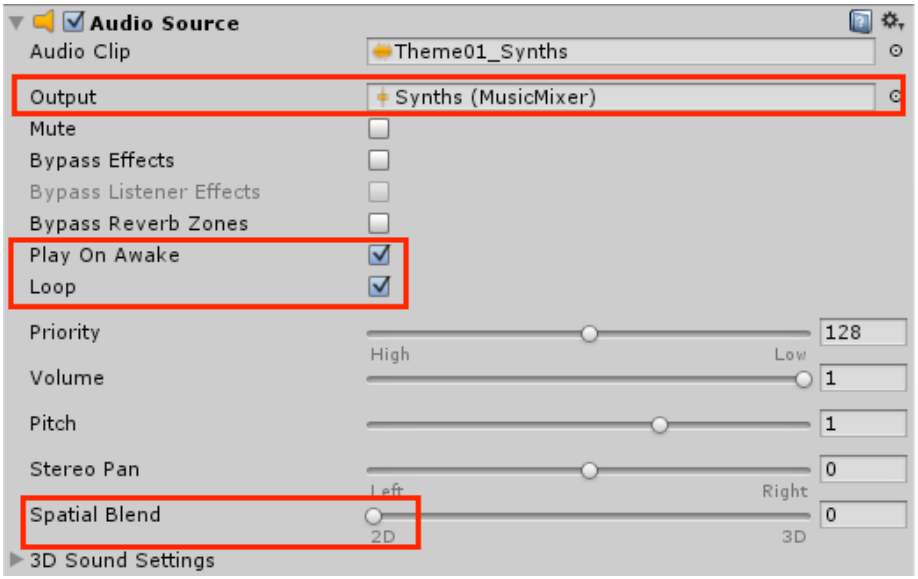
Insert image 1362OT_09_25.png

5. From the **Hierarchy** view, create a new **Empty** game object. Name it **Music**. Then, add two **Empty Child** game objects to it. Name them as **Percussion** and **Synth**.
6. From the **Project** view, drag the **Audio Clip** **Theme01_Percussion** into the **Percussion** game object in **Hierarchy**. Select **Percussion** and, in the **Inspector** view, access the **Audio Source** component. Change its **Output** to **Percussion (MusicMixer)**, make sure the **Play On Awake** option is checked, check the **Loop** option, and make sure its **Spatial Blend** is set to **2D**.



Insert image 1362OT_09_26.png

7. Now drag **Theme01_Synths** into the **Synths** game object. From the **Inspector** view, change its **Output** to **Synths (MusicMixer)**, make sure the **Play On Awake** option is checked, check the **Loop** option, and make sure its **Spatial Blend** is set to **2D**.

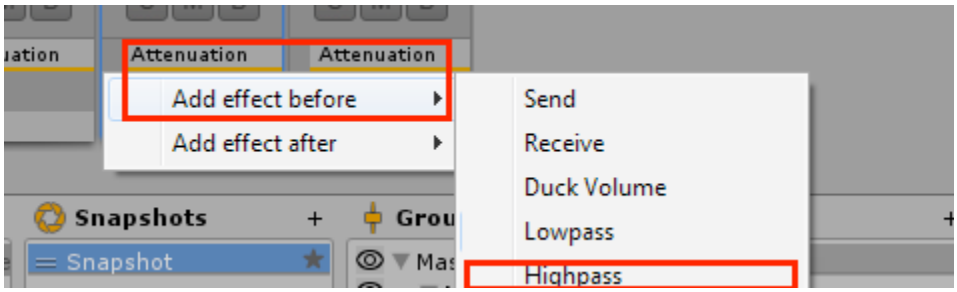


Insert image 1362OT_09_27.png

8. Open the **Audio Mixer** and play the scene - we will now use the mixer to set the soundtrack for the start of the scene. With the scene playing, click the button **Edit in Play Mode**, on the top of the Audio Mixer. Then, drop the volume on the **Synths** track down to **-30 dB**.

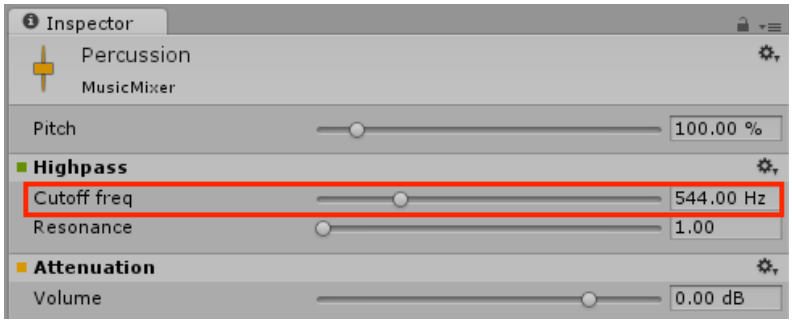
Insert image 1362OT_09_28.png

9. Now select the **Percussion** track. Right-click **Attenuation** and add the **High-pass** effect before it.



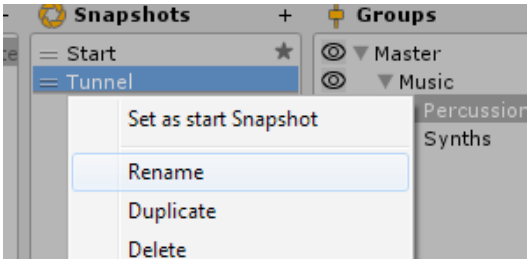
Insert image 1362OT_09_29.png

10. From the **Inspector** view, change the **Cutoff frequency** of the **High-pass** effect to **544.00 Hz**.



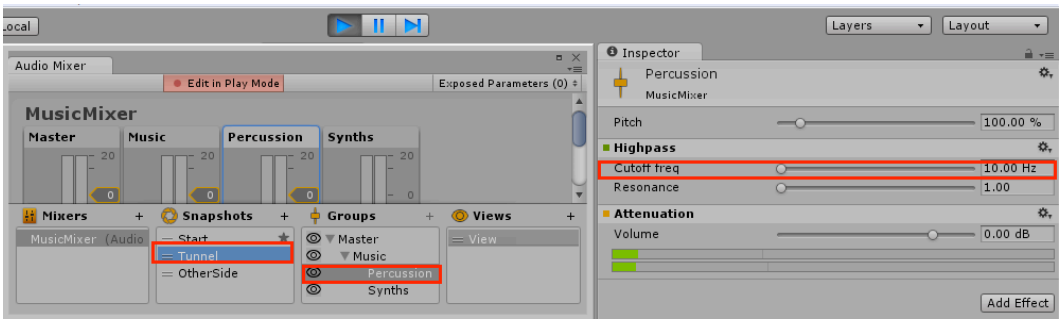
Insert image 1362OT_09_30.png

11. Every change so far has been assigned to the current **Snapshot**. From the **Snaphots** view, right-click the current **Snapshot** and rename it to **Start**. Then, right-click **Start** and select the option **Duplicate**. Rename the new snapshot as **Tunnel**.



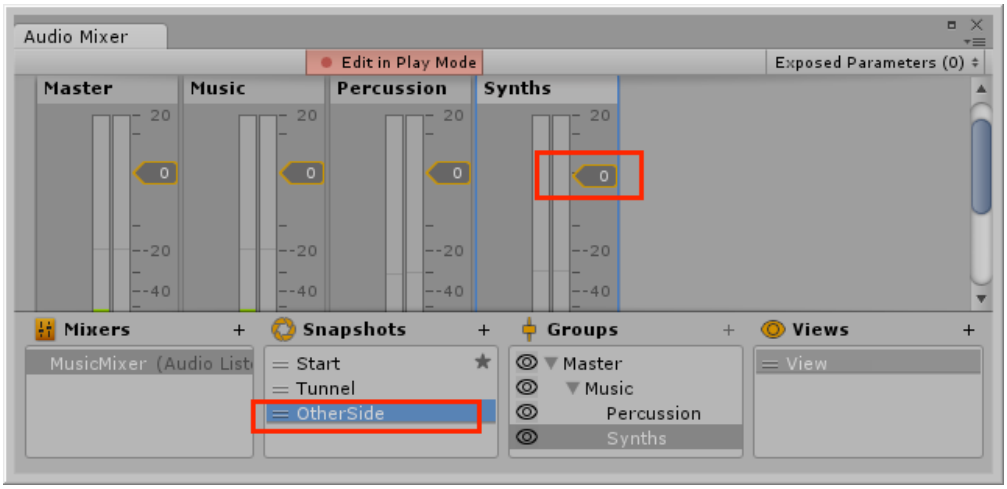
Insert image 1362OT_09_31.png

12. Select the **Tunnel** snapshot. Then, from the **Inspector** view, change the **Cutoff frequency** of the **Highpass** effect to **10.00 Hz**.



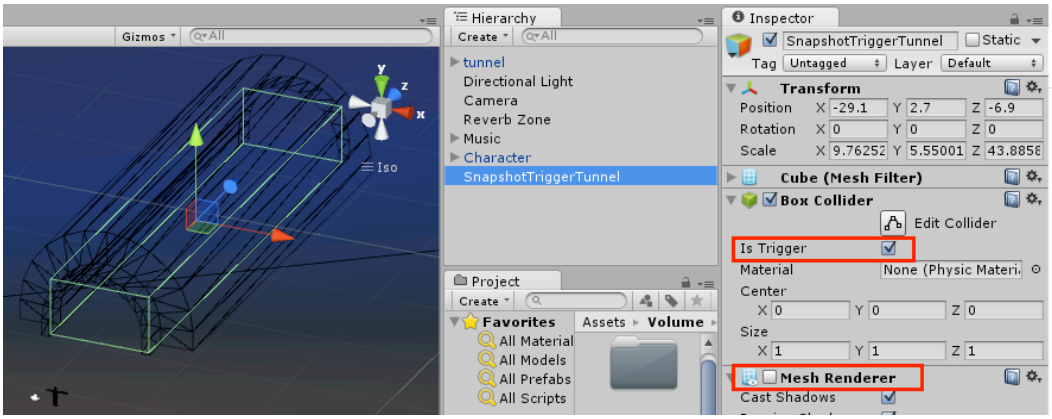
Insert image 1362OT_09_32.png

13. Switch between snapshots **Tunnel** and **Start**. You'll be able to hear the difference.
14. Duplicate the **Tunnel** snapshot, rename it to **OtherSide** and select it.
15. Raise the volume of the **Synths** track up to **0 dB**.



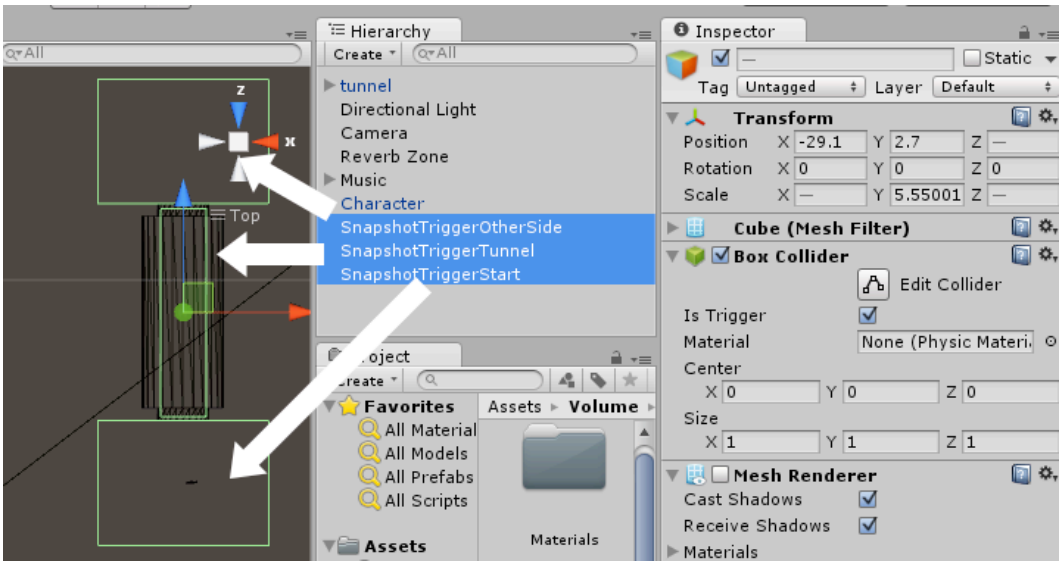
Insert image 1362OT_09_33.png

16. Now that we have our three Snapshots, it´s time to create triggers to make transitions among them. From the **Hierarchy** view, use the **Create** dropdown menu to add a **Cube** to the scene (**Create | 3D Object | Cube**).
17. Select the new Cube and rename it to `SnapshotTriggerTunnel`. Then, from the **Inspector** view, access the **Box Collider** component and check the option **Is Trigger**. Also, uncheck its **Mesh Renderer** component. Finally, adjust its size and position to the scene tunnel's interior.



Insert image 1362OT_09_34.png

18. Make two copies of `SnapshotTriggerTunnel` and rename them to `SnapshotTriggerStart` and `SnapshotTriggerOtherSide`. Then, adjust their size and position so that they occupy the areas before the tunnel's entrance (where the character is), and after its other end.



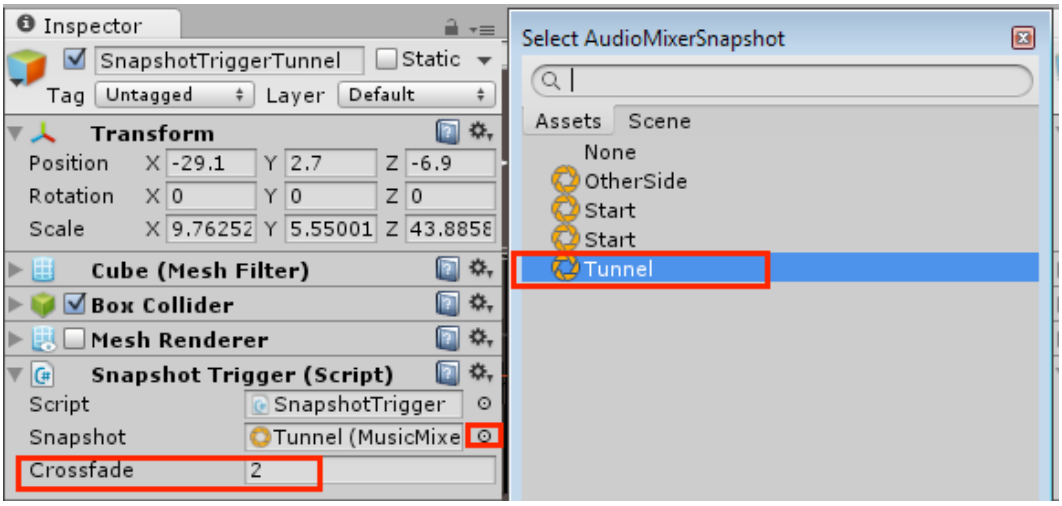
Insert image 1362OT_09_35.png

19. In the **Project** view, create a new **C# Script** and rename it to `SnapshotTrigger`.
20. Open the script in your editor and replace everything with the following code:

```
using UnityEngine;
```

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

public class SnapshotTrigger : MonoBehaviour{
    public AudioMixerSnapshot snapshot;
    public float crossfade;

    private void OnTriggerEnter(Collider other){
        snapshot.TransitionTo (crossfade);
    }
}
```

21. Save your script and attach it to `SnapshotTriggerTunnel` , `SnapshotTriggerStart`, and `SnapshotTriggerOtherSide`.
22. Select `SnapshotTriggerTunnel`. Then, from the **Inspector** view, access the **Snapshot Trigger** component, setting **Snapshot** as **Tunnel** and **Crossfade** as `2`.



Insert image 1362OT_09_36.png

23. Make changes to `SnapshotTriggerStart` and `SnapshotTriggerOtherSide`, setting their **Snapshots** to **Start** and **OtherSide**, respectively.
24. Test the scene. The background music should change as the character moves from its starting point, through the tunnel, and into the other side.

How it works...

The **Snapshot** feature allows you to save **Audio Mixer** states (including all volume levels, every filter setting, and so on) so you can change, at runtime, those mixing

preferences, making the audio design more suitable for specific locations or gameplay settings. For this recipe, we have created three **Snapshots** for different moments in the player's journey: before entering the tunnel, inside the tunnel, and outside the tunnel. We have used the **Highpass** filter to make the initial Snapshot less intense. We have also turned the **Synths** track volume up to emphasize the open environment outside the tunnel. Hopefully, changes in the audio mix will collaborate setting the right mood for the game.

To activate our snapshots, we have placed **trigger colliders** featuring our **Snapshot Trigger** component, in which we set the desired Snapshot and the time, in seconds, that it takes to make the transition (a crossfade) between the previous snapshot and the next. In fact, the function in our script is really this straightforward: the line of code `snapshot.TransitionTo (crossfade)` simply starts a transition lasting `crossfade` seconds to the desired `snapshot`.

There's more...

Here is some information on how to fine tune and customize this recipe:

Reducing the need for multiple audio clips

You might have noticed how different the `Theme01_Percussion` audio clip sounds when the **Cutoff frequency** of the **High-pass** filter is set as `10.00 Hz`. The reason for that is because the high-pass filter, as its name suggests, cuts off lower frequencies of the audio signal. In this case, it attenuated the bass drum down to inaudible levels, while keeping the shakers audible. The opposite effect could be achieved through the **Lowpass** filter. A major benefit is the opportunity of virtually having two separate tracks into the same audio clip.

Dealing with Audio File Formats and compression rates

To avoid loss of audio quality, you should import your sound clips using the appropriate file format, depending upon your target platform. If you are not sure which format to use, please check out Unity's documentation on the subject at `http://docs.unity3d.com/Documentation/Manual/AudioFiles.html`.

Applying snapshots to background noise

Although we have applied Snapshots to our music soundtrack, background noise can also benefit immensely. If your character travels across places that are significantly different, transitioning from open spaces to indoor environments, you should consider applying snapshots to your environment audio mix. Be careful, however, to create separate Audio Mixers for Music and Environment - unless you don't mind having musical and ambient sound tied to the same Snapshot.

Getting creative with effects

In this recipe, we have mentioned the **High-pass** and **Low-pass** filters. However, there are many effects that can make audio clips sound radically different. Experiment! Try applying effects such as **Distortion**, **Flange**, and **Chorus**. In fact, we encourage you to try every effect, playing with their settings. The creative use of these effects can bring out different expressions to a single audio clip.

See also

- *Adding volume control with Audio Mixers*
- *Balancing soundtrack volume with Ducking*

Balancing soundtrack volume with Ducking

As much as background music can be important in establishing the right atmosphere, there will be times when other audio clips should be emphasized, and the music volume, turned down for the duration of that clip. This effect is known as *Ducking*. Maybe you need it for dramatic effect (simulating hearing loss after an explosion took place), or maybe you want to make sure the player listens to a specific bit of information. In this recipe, we will learn how to emphasize a piece of dialog by ducking the audio whenever a specific sound message is played. For that effect, we will use the new **Audio Mixer** to send information between tracks.

Getting ready

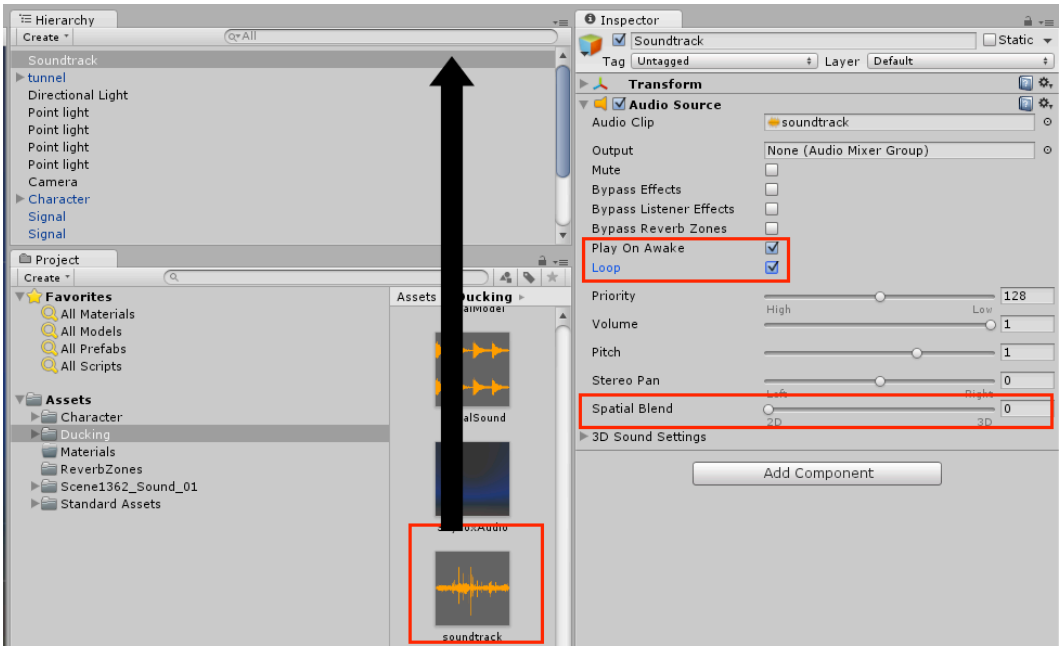
For this recipe, we have provided the audio clip `soundtrack.mp3` and a Unity package named `Ducking.unitypackage`, containing an initial scene. All files are available inside the `1362_09_05` folder.

How to do it...

To apply Audio Ducking to your soundtrack, follow these steps:

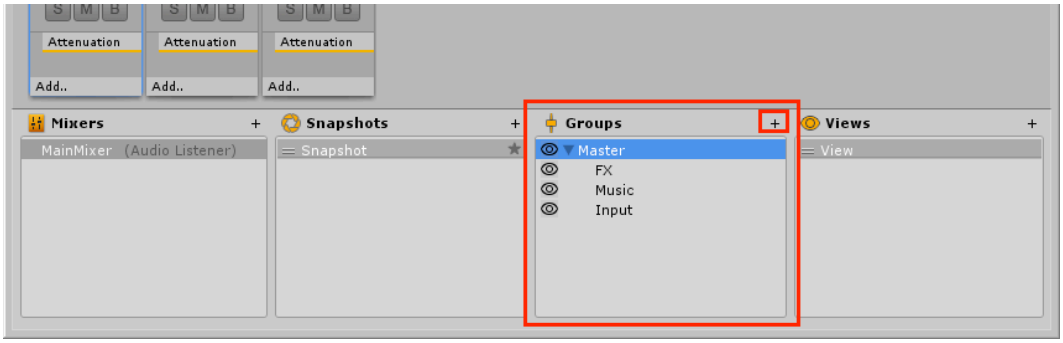
1. Import `Ducking.unitypackage` and `soundtrack.mp3` into your project.
2. Open the scene **Ducking** (available in the folder **Assets | Ducking**). Play the scene and walk towards the semitransparent green wall in the tunnel, using the *WASD* keys (pressing *SHIFT* to run). You should hear the **robotDucking** audio clip play as the character collides with the wall.
3. From the **Create** dropdown on the top of the **Hierarchy** view, choose **Create Empty** to add a new game object to the scene. Name it **Soundtrack**.

4. From the **Assets | Ducking** folder, drag the `soundtrack` audio clip into the **Soundtrack** game object. Then, select the **Soundtrack** object and, from the **Inspector** view, **Audio Source** component, check the option **Loop** and make sure the option **Play On Awake** is checked, and **Spatial Blend** set to **2D**.



Insert image 1362OT_09_36.png

5. Test the scene again. The soundtrack music should be playing.
6. From the **Project** view, use the **Create** dropdown menu to add an **Audio Mixer** to the project. Name it **MainMixer**. Double click it to open the **Audio Mixer** window.
7. From the **Groups** view, highlight **Master** and click the + sign to add a child to the **Master** group. Name it **Music**. Then, highlight **Master** again and add a new child group named **FX**. Finally, add a third child to the **Master** group, named **Input**.

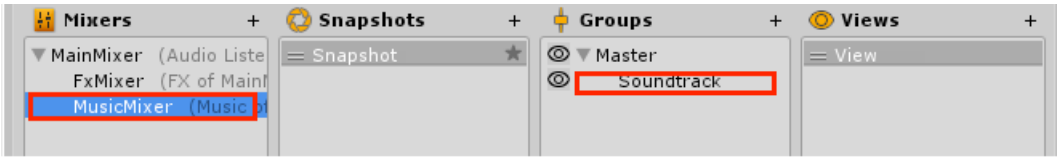


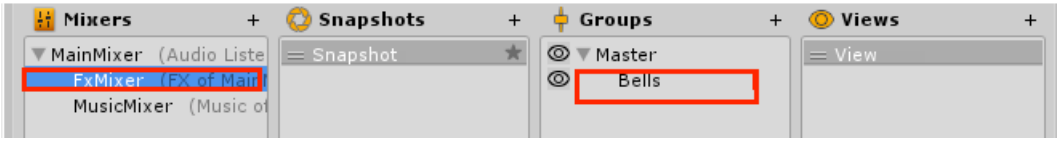
Insert image 1362OT_09_38.png

8. From the **Mixers** view, highlight **MainMixer** and click the **+** sign to add a new **Mixer** to the project. Name it **MusicMixer**. Then, drag it into the **MainMixer** and select the group **Music** as its **Output**. Repeat the operation to add a mixer named **FxMixer** to the project, selecting the **FX** group as output.

Insert image 1362OT_09_39.png

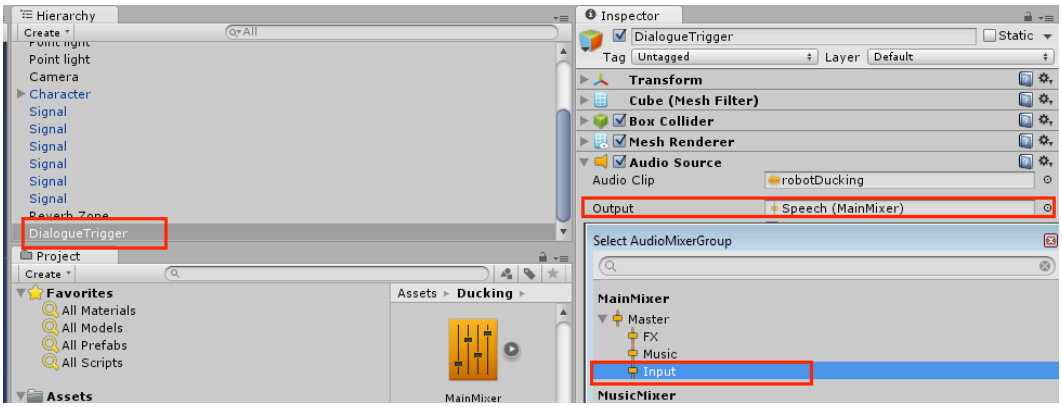
9. Now select the **MusicMixer**. Select its **Master** group and add a child named **Soundtrack**. Then, select **FxMixer** and add a child named **Bells**.





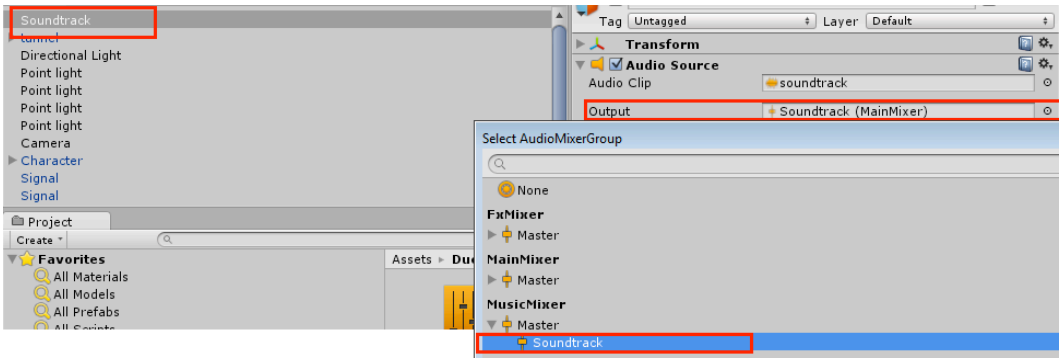
Insert image 1362OT_09_40.png

10. From the **Hierarchy** view, select the **DialogueTrigger** object. Then, in the **Inspector** view, **Audio Source** component, Change its **Output** track to **MainMixer | Input**.



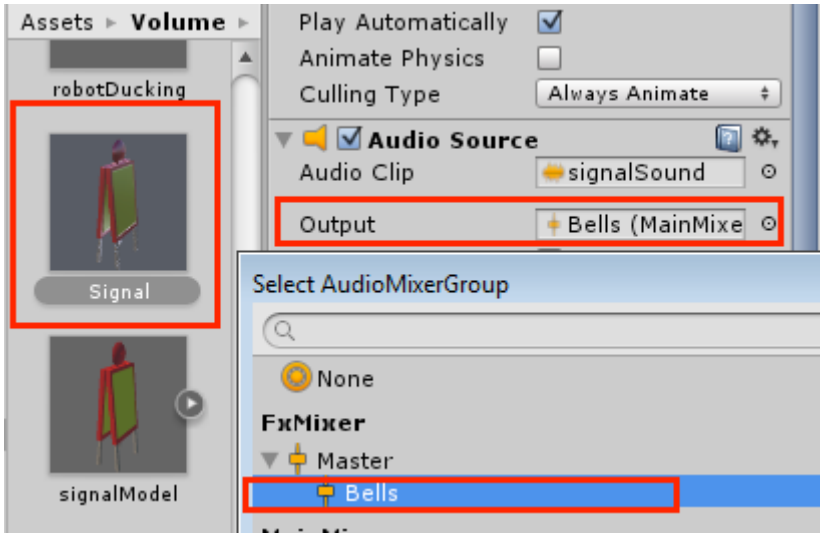
Insert image 1362OT_09_41.png

11. Now select the **Soundtrack** game object and, in the **Inspector** view, **Audio Source** component, change its **Output** track to **MusicMixer | Soundtrack**.



Insert image 1362OT_09_42.png

12. Finally, from the **Assets** folder in the **Project** view, select the **Signal** prefab and, from the **Inspector** view, access its **Audio Source** component and change its **Output** to **FxMixer | Bells**.

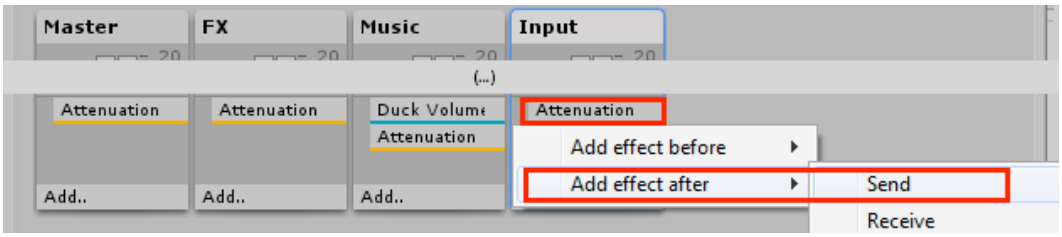


Insert image 1362OT_09_43.png

13. Open the **Audio Mixer** window. Choose **MainMixer**, select the **Music** track controller, right-click **Attenuation** and, using the context menu, add the **Duck Volume** effect before **Attenuation**.

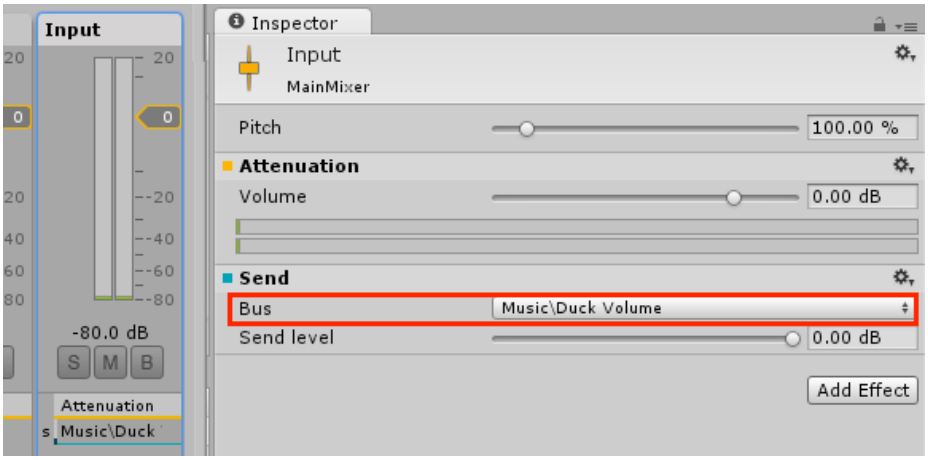
Insert image 1362OT_09_44.png

14. Now Select the **Input** track, right-click **Attenuation** and, using the context menu, add **Send** after **Attenuation**.



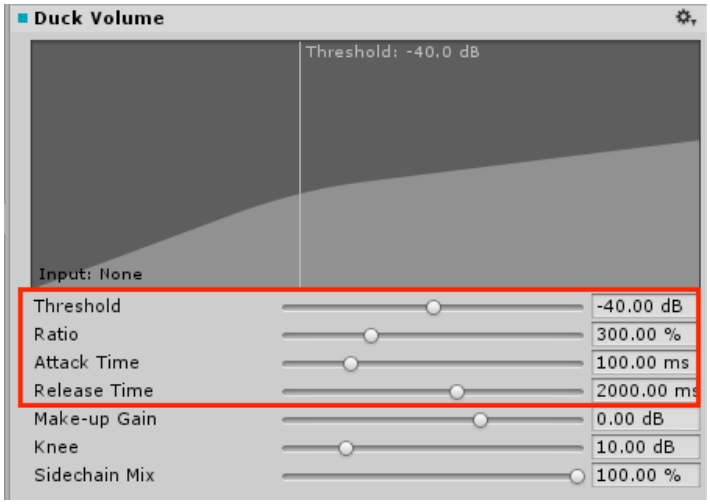
Insert image 1362OT_09_45.png

15. With **Input** track still selected, go to the **Inspector** view, and change the **Bus** setting in **Send** to **Music\Duck Volume** and its **Send** level to `0.00 db`.



Insert image 1362OT_09_46.png

16. Select the **Music** track. From the Inspector view, change the settings on Duck Volume as follows: **Threshold: -40.00 db**; **Ratio: 300.00 %**; **Attack Time: 100.00 ms**; **Release Time: 2000.00 ms**.

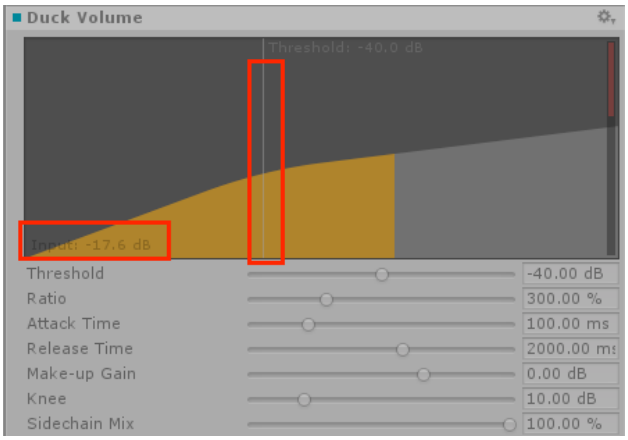


Insert image 1362OT_09_47.png

17. Test the scene again. Entering the trigger object will cause the soundtrack volume to drop considerably, recovering the original volume in 2 seconds.

How it works...

In this recipe, we have created, in addition to Music and Sound FX, a group named Input, to which we have routed the audio clip that triggers the **Duck Volume** effect attached to our Music track. The Duck Volume effect changes the track's volume whenever it receives an input that is louder than indicated in its Threshold setting. In our case, we have sent the Input track as input, and adjusted the settings so the volume would be reduced as soon as 0.1 seconds after the input had been received, turning back to its original value of 2 seconds after the input has ceased. The amount of volume reduction was determined by our Ratio of 300.00 percent. Playing around with the setting values will give you a better idea on how each parameter affects the final result. Also, make sure to visualize the graphic as the trigger sound is played. You will be able to see how the Input sound passes the threshold, triggering the effect.



Insert image 1362OT_09_48.png

Also, please note that we have organized our tracks so other sound clips (other than speech) would not affect the volume of the music - but every music clip would be affected by audio clips sent to the Input track.

See also

- Adding volume control with Audio Mixers
- Making a dynamic soundtrack with Snapshots