6
Playing and Manipulating Sounds

In this chapter, we will cover:

- Matching audio pitch to animation speed
- Adding customizable volume controls
- Simulating ambiance with Reverb Zones
- Prevent restarting an already playing AudioClip
- Object waits until audio finished playing before destroying itself
- Using Animation events to play audio clips
- Making a dynamic soundtrack

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 Sim game or the creeping noises in horror games. Think of how those sounds transport you *into* the game.

This chapter is filled with recipes that hopefully will help you implement a better and more efficient sound design to your projects.

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.

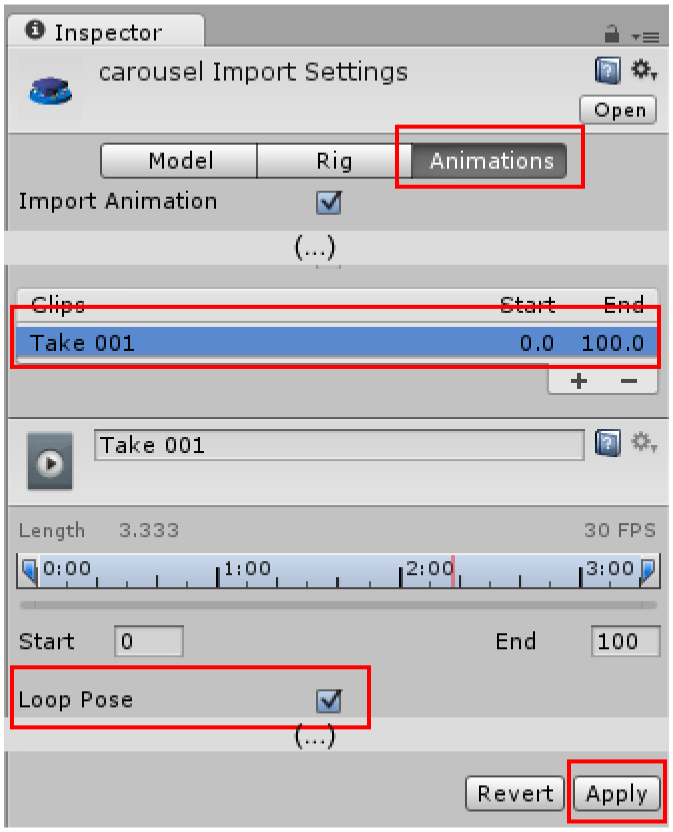
Getting ready

For this recipe, you'll need an animated 3D object and an audio clip. Please use the files *carousel.fbx* and *carouselSound.wav*, available in the folder *0423_06_01*.

How to do it...

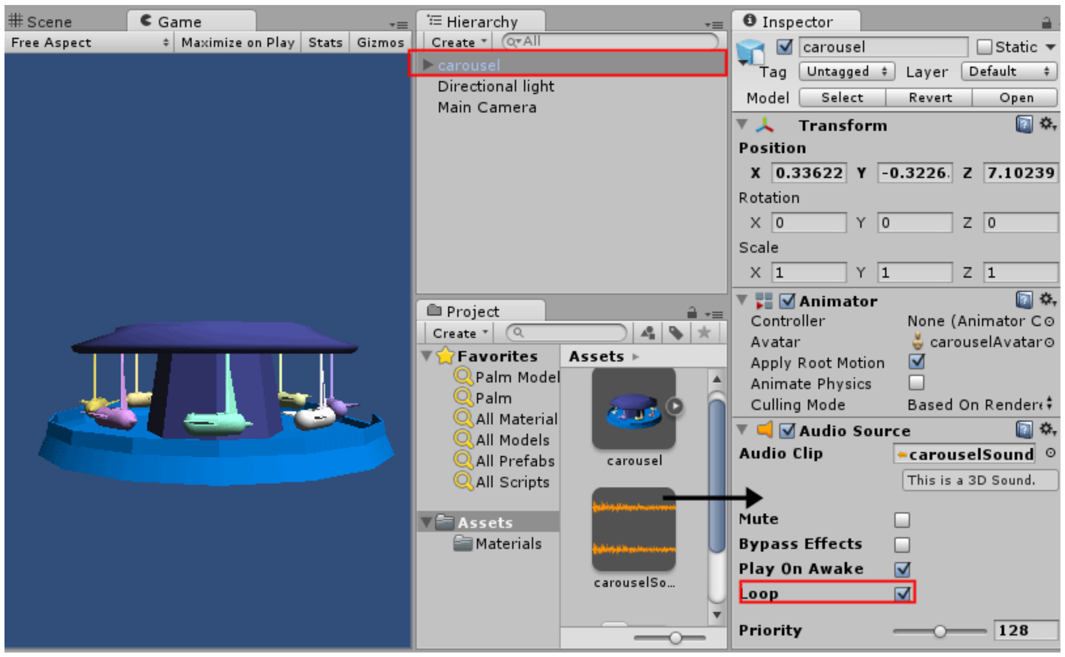
To change the pitch of an audio clip according to the speed of an animated object, please follow these steps:

1. Import the file carousel.fbx into your Unity Project.
2. Select the *carousel.fbx* file in the *Project* view. Then, in the *Inspector* view, check its *Import Settings*. Under *Animation*, select the clip *Take 001* and make sure to check the option *Loop Pose*. Click the button *Apply* to save changes.



0423_06_01.png

3. Add the carousel to the scene by dragging it from the *Project* view into the *Hierarchy* view.
4. Add a *Directional Light* to the scene through the *Create* dropdown menu on top of the *Hierarchy* view.
5. Import the audio clip *carouselSound.wav*.
6. Select the Game Object *carousel* and drag *carouselSound* from the *Project* view into the *Inspector* view, adding it as an *Audio Source* for that object.
7. In the *Audio Source* component of the *carousel*, check the box for the *Loop* option.



Hierarchy: carousel, Directional light, Main Camera

carousel — Static
Tag: Untagged — Layer: Default
Model: Select, Revert, Open

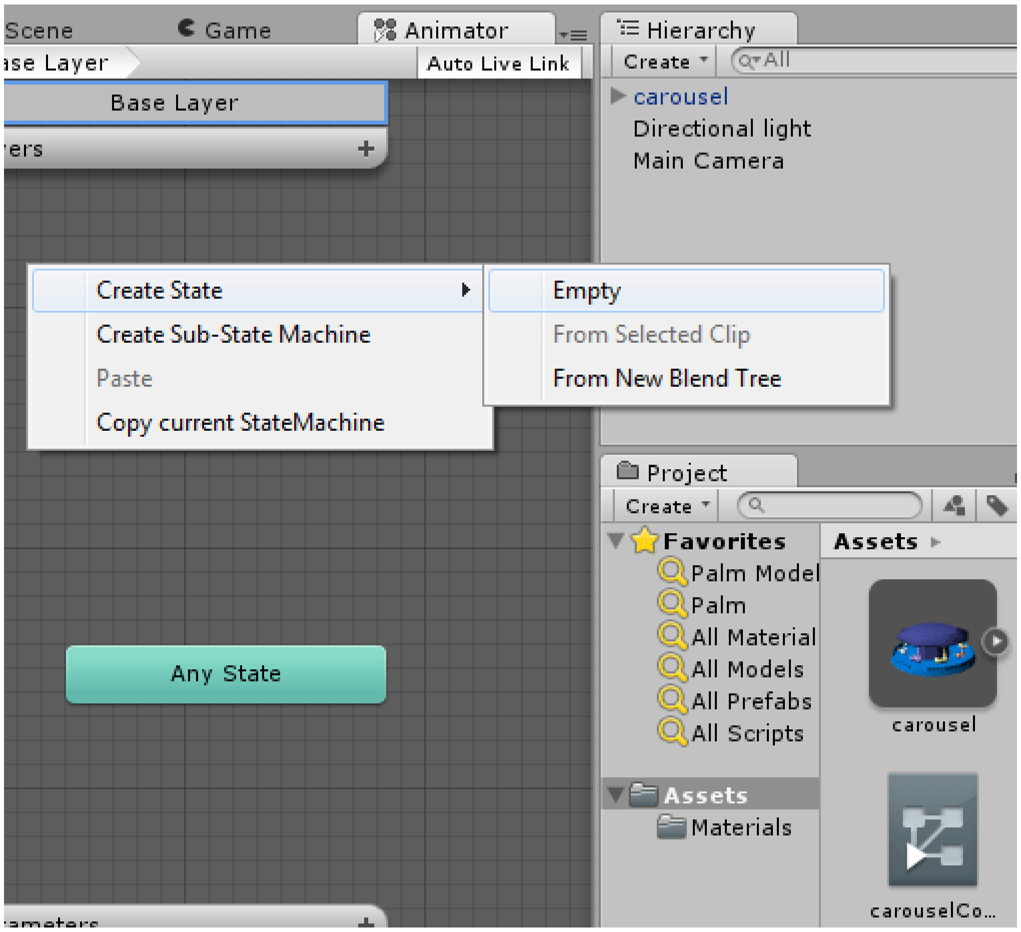
Transform
Position: X 0.33622, Y -0.3226, Z 7.10239
Rotation: X 0, Y 0, Z 0
Scale: X 1, Y 1, Z 1

Animator
Controller: None (Animator Co
Avatar: carouselAvatar
Apply Root Motion: ✓
Animate Physics: ☐
Culling Mode: Based On Render

Audio Source
Audio Clip: carouselSound
This is a 3D Sound.
Mute: ☐
Bypass Effects: ☐
Play On Awake: ✓
Loop: ✓
Priority: 128

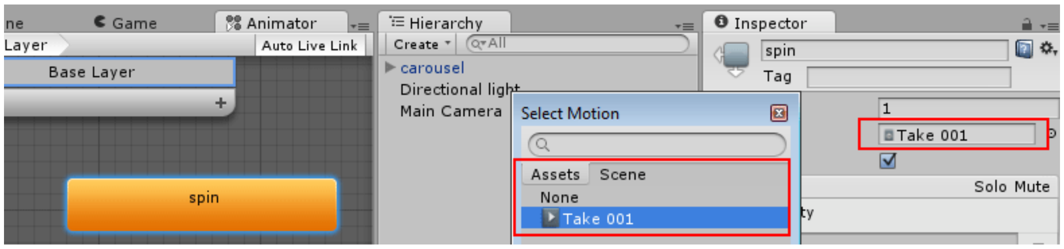
0423_06_02.png

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



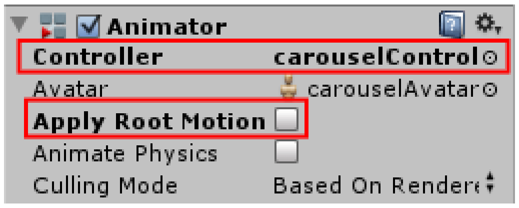
0423_06_03.png

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



0423_06_04.png

11. From the *Hierarchy* view, select the *carousel*. Then, in the Animator component (in the Inspector view), set *CarouselAnimator* as its *Controller* and uncheck the option *Apply Root Motion*.



Animator

Controller	carouselControl
Avatar	carouselAvatar
Apply Root Motion	☐
Animate Physics	☐
Culling Mode	Based On Render

0423_06_05.png

12. In the *Project* window, create a new *C# Script* and rename it as *carouselScript*.
13. Open the script in your editor and replace everything with the following code:

```
using UnityEngine;
using System.Collections;

public class carouselScript : MonoBehaviour
{
   public float speed = 0.0f;
   public float minSpeed = 0.0f;
   public float maxSpeed = 2.0f;
   public float animationSoundRatio = 1.0f;
   private Animator animator;
   void Start(){
   animator =  GetComponent<Animator>();
   }
    void Update(){
        animator.speed = speed;
        audio.pitch speed * animationSoundRatio;
    }

    void OnGUI(){
      Rect rect = new Rect(10, 10, 100, 30);
        speed = GUI.HorizontalSlider(rect, speed, minSpeed,
maxSpeed);
    }

}
```

14. Save your script and add it as a component to the *carousel*.
15. Play the scene and change the animation speed, along with the audio pitch, using the slide bar.

How it works...

The idea behind the script and its implementation are actually quite straightforward: it creates a slidebar from which the user can change the speed of the animator component. Then, it updates the audio pitch based on that number.

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.

Adding customizable volume controls

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 create a sound volume control GUI that can be switched from a single volume bar to independent Music and Effects control.

Getting ready

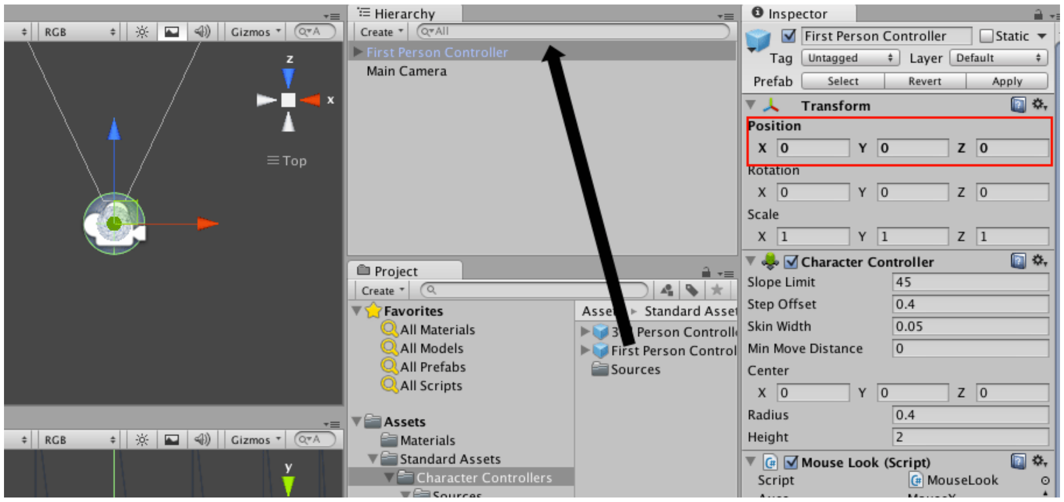
For this recipe, you'll need the audio files *soundFX.wav* and *soundtrack.mp3*, available in the folder *0423_06_02*.

How to do it...

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

1. Import the required *soundtrack* audio file.
2. In the *Project* view, select the *soundtrack* file. Make sure the *3D Sound* option of the *Audio Importer* (in the *Inspector* view) is unselected. If not, unselect it.
3. Let's now import the *soundFX* audio clip. This time, we will make sure to leave the 3D Sound option checked.
4. Make sure the *First Person Controller* prefab is available in your Project. You can do that by importing it through the menu *Assets > Import Package... > Character Controller*.

5. Add the *First Person Controller* to your scene by dragging it from the *Project* view to the *Hierarchy* view. Then, in the *Inspector* view, reset its position to *X: 0, Y:0, Z:0*.



First Person Controller ☐ Static
Tag Untagged Layer Default
Prefab

Transform

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

Character Controller

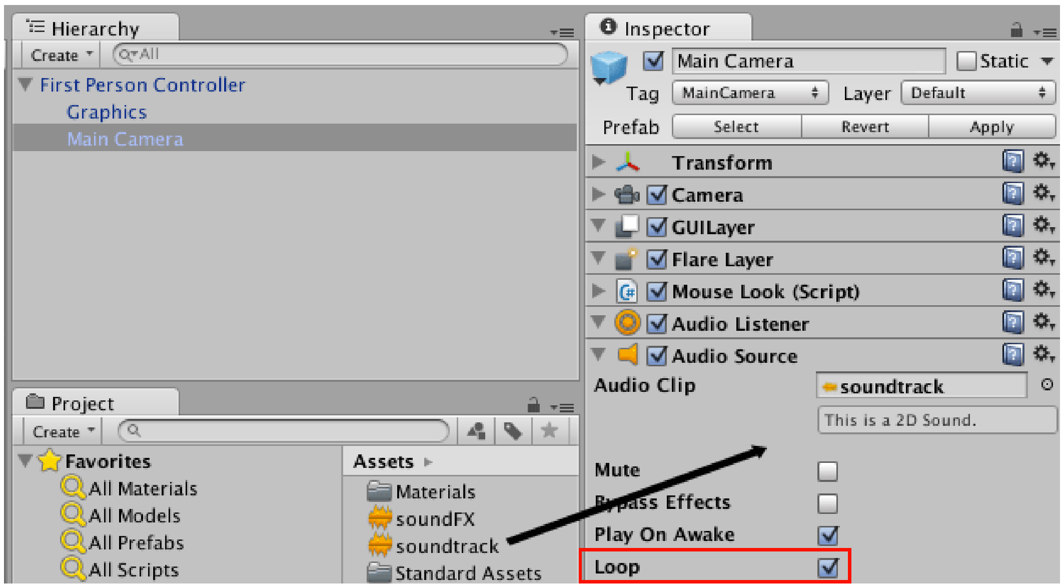
Slope Limit	45
Step Offset	0.4
Skin Width	0.05
Min Move Distance	0
Center	X 0 Y 0 Z 0
Radius	0.4
Height	2

Mouse Look (Script)

Script MouseLook

0423_06_06.png

6. Delete the original *Main Camera* from the scene.
7. Expand the *First Person Controller* Object in the *Hierarchy* view and select the *Main Camera* child.
8. Drag the *soundtrack* from the *Project* view to bottom of *the Main Camera Inspector* view, adding it as an *Audio Source* component. Check the option *Loop*.



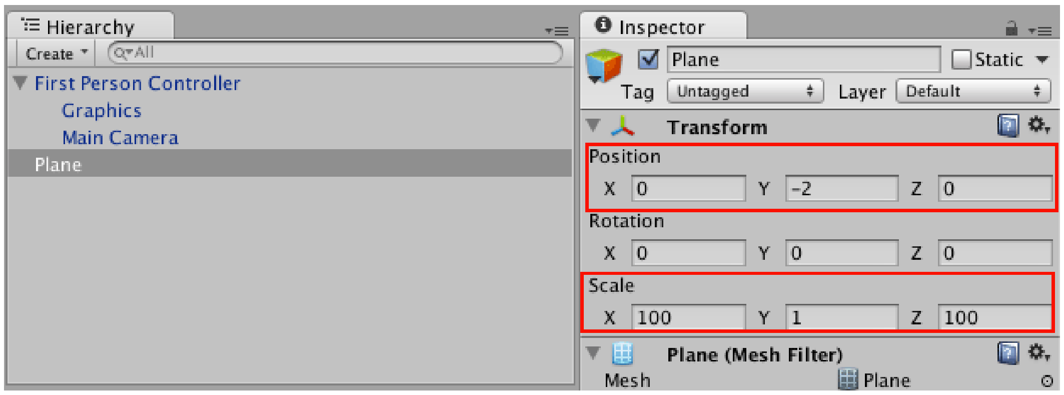
Main Camera ☐ Static
Tag: MainCamera — Layer: Default
Prefab: Select | Revert | Apply

Audio Clip: soundtrack
This is a 2D Sound.

Mute ☐
Bypass Effects ☐
Play On Awake ✓
Loop ✓

0423_06_07.png

9. Using the *Create* dropdown of the *Hierarchy* view, add a *Plane* to your scene.
10. In the *Transform* component of the *Inspector* view, change the plane's *Scale* X and Z values to 100. Also, change its Position to X:0, Y: -2, Z:0. The *Plane* object will be the ground for our scene.



Plane ☐ Static
Tag: Untagged — Layer: Default

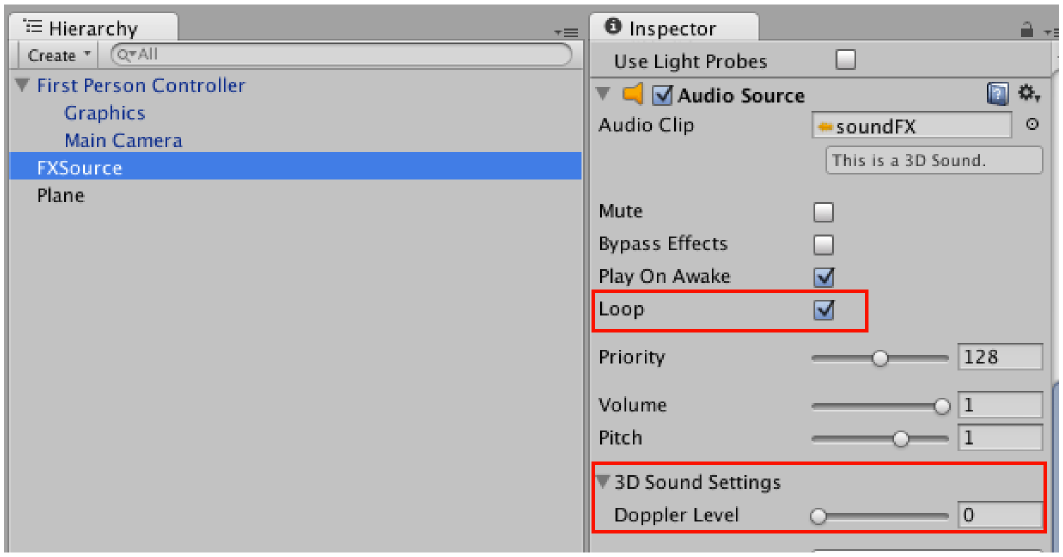
Position: X 0 Y -2 Z 0
Rotation: X 0 Y 0 Z 0
Scale: X 100 Y 1 Z 100

Mesh: Plane

0423_06_08.png

11. Using the *Create* dropdown of the *Hierarchy* view, add a *Cube* to your scene and rename it to *FXSource*. Set its Position to X:0, y:0, Z:6.

12. Drag the *soundFX* from the *Project* view to bottom of *the FXSource Inspector* view, adding it as an *Audio Source* component.
13. In the *Audio Source* component of the *FXSource*, check the *Loop* option and change the *Doppler Level* to 0.



Use Light Probes ☐
Audio Source ✓
Audio Clip: soundFX
Mute ☐
Bypass Effects ☐
Play On Awake ✓
Loop ✓
Priority 128
Volume 1
Pitch 1
3D Sound Settings
Doppler Level 0

0423_06_09.png

16. Add a *Directional Light* to the scene.
17. In the *Project* view, create a new *C# Script* and rename it *volumeControlScript*.
18. Open the script in your editor and replace everything with the following code:

```
using UnityEngine;
using System.Collections;
[RequireComponent(typeof(AudioSource))]
public class volumeControlScript : MonoBehaviour
{
    bool separateSoundtrack = true;
    float minVolume = 0.0f;
    float maxVolume = 1.0f;
    float initialVolume = 1.0f;
    float soundtrackVolume = 1.0f;
    bool displaySliders = false;

    void Start()
    {
        if (separateSoundtrack)
        {
            audio.ignoreListenerVolume = true;
        }
    }
    void Update()
    {
        AudioListener.volume = initialVolume;
        if (separateSoundtrack)
        {
            audio.volume = soundtrackVolume;
        }
        else
        {
            audio.volume = initialVolume;
        }
    }

    void OnGUI()
    {

        Event e = Event.current;
        if (e.type == EventType.KeyUp && e.keyCode ==
KeyCode.Escape)
        {
            displaySliders = !displaySliders;
        }

        if (displaySliders)
        {
            if (!separateSoundtrack)
            {
                GUI.Label(new Rect(10, 0, 100, 30), "Volume");
                initialVolume = GUI.HorizontalSlider(new
Rect(10, 20, 100, 30), initialVolume, minVolume, maxVolume);
            }
            else
            {
                GUI.Label(new Rect(10, 0, 100, 30), "Sound
FX");
                initialVolume = GUI.HorizontalSlider(new
Rect(10, 20, 100, 30), initialVolume, minVolume, maxVolume);
                GUI.Label(new Rect(10, 40, 100, 30), "Music");
                soundtrackVolume = GUI.HorizontalSlider(new
Rect(10, 60, 100, 30), soundtrackVolume, minVolume,
maxVolume);
            }
        }
    }
}
```

19. Save your script and attach it to the *Main Camera* by dragging it from the *Project* view to camera game object in the *Hierarchy* view.
20. Play the scene and hit *Escape* in your keyboard. You'll see the volume slide bars on the top left of the Game viewport.

How it works...

By default, every sound in the scene has its volume controlled by the *Volume* parameter of the camera's *Audio Listener* component. With our script, we assign the *Volume* slide bar value to that parameter. Also, we create a separate volume bar for the soundtrack, making it independent from the camera's *Audio Listener*.

There's more...

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

A single volume bar

If you need to simplify even more the volume controls, you can use a single volume bar by leaving the option *Separate Soundtrack* unchecked.

Don't wait for the ESC key

If you want your volume bars to be displayed as soon as, make sure to tick the checkbox for the parameter named *Display Sliders*.

See also

Making a dynamic soundtrack

Pausing the game

Simulating a tunnel environment 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 that look. Sound behaves differently depending on the environment 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.

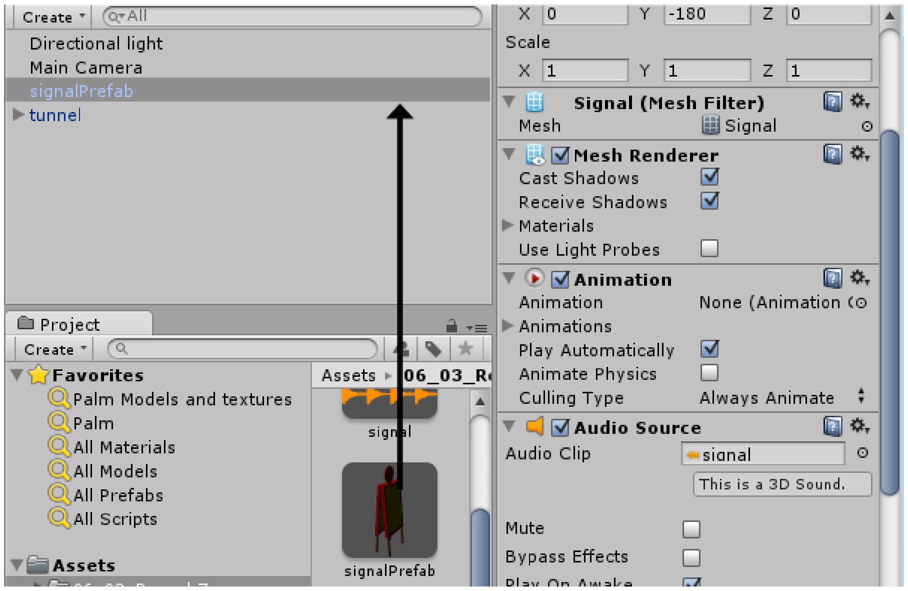
Getting ready

For this recipe, we have prepared a package containing a basic level named *reverbZoneLevel* and the *signal* prefab. The package is in the folder *0423_06_04*.

How to do it...

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

1. Import the package *reverbZones* into your Unity Project.
2. In the *Project* view, open the level *reverbZoneLevel*, inside the folder *06_03 ReverbZones*. This is a basic scene featuring a First Person camera and a tunnel.
3. Now drag *signalPrefab* from the *Project* view into the *Hierarchy* view. That should add a sound-emitting object to the scene. Place it in the center of the tunnel.



X 0 Y -180 Z 0
Scale
X 1 Y 1 Z 1

Signal (Mesh Filter)
Mesh Signal

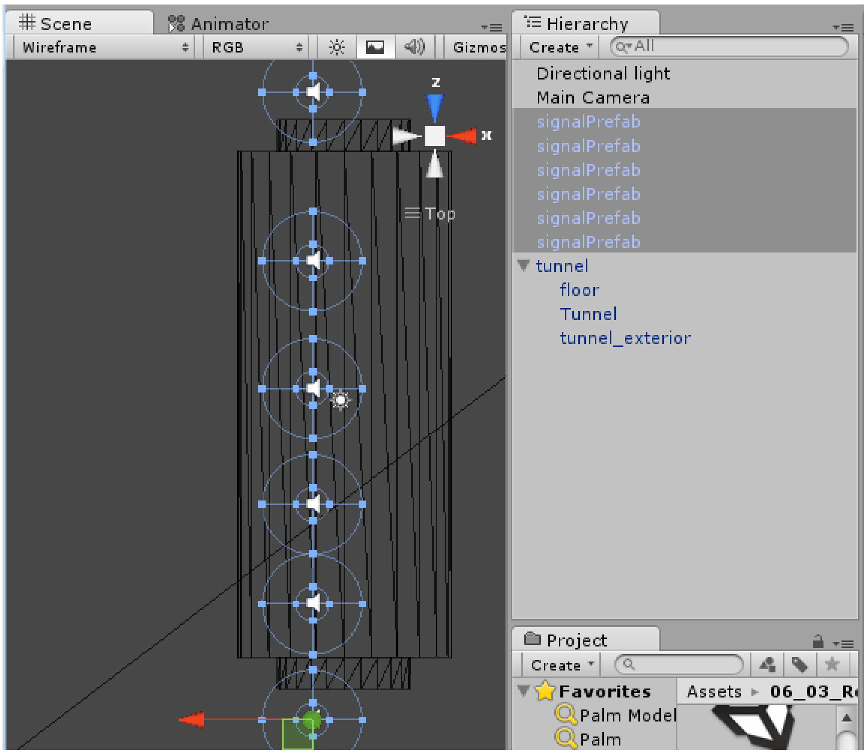
Mesh Renderer
Cast Shadows ✓
Receive Shadows ✓
Materials
Use Light Probes ☐

Animation
Animation None (Animation C
Animations
Play Automatically ✓
Animate Physics ☐
Culling Type Always Animate

Audio Source
Audio Clip signal
This is a 3D Sound.
Mute ☐
Bypass Effects ☐
Play On Awake ✓

0423_06_10.png

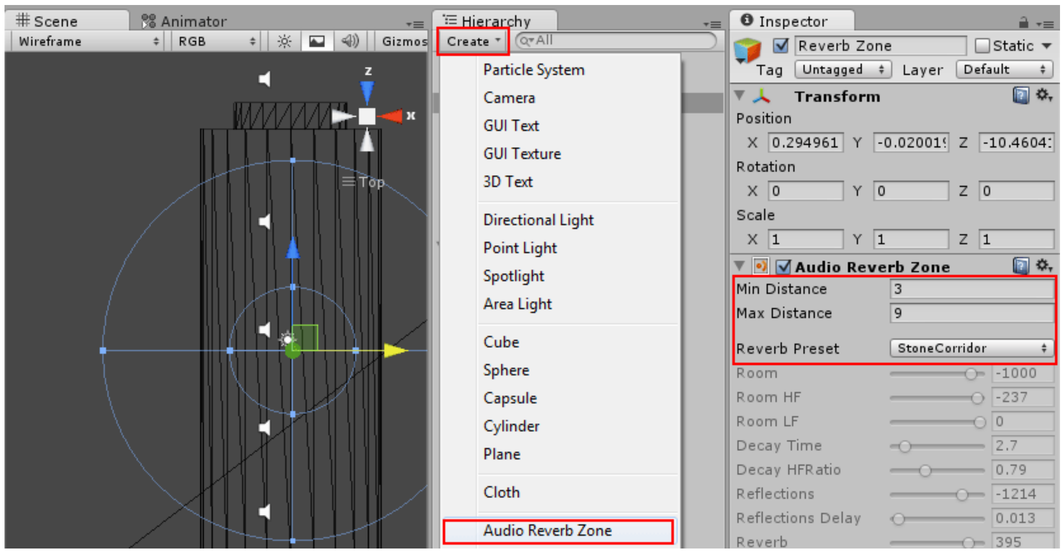
4. Make five copies of the *signalPrefab* Game Object and distribute them across the tunnel (leaving a copy just outside each entrance).



Directional light
Main Camera
signalPrefab
signalPrefab
signalPrefab
signalPrefab
signalPrefab
signalPrefab
tunnel
floor
Tunnel
tunnel_exterior

0423_06_11.png

5. In the *Hierarchy* view, click *Create* to add an *Audio 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: 3; Max Distance: 9; Preset: StoneCorridor.*



Reverb Zone Static
Tag Untagged Layer Default

Transform
Position X 0.294961 Y -0.02001 Z -10.4604
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Audio Reverb Zone
Min Distance 3
Max Distance 9
Reverb Preset StoneCorridor
Room -1000
Room HF -237
Room LF 0
Decay Time 2.7
Decay HFRatio 0.79
Reflections -1214
Reflections Delay 0.013
Reverb 395

0423_06_12.png

7. Play the scene and walk through the tunnel. 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:

Attach 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 *signalPrefab*) 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.

Make 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.

See also

Simulating underwater ambiance with audio filters (Pro Only)

Prevent restarting an already playing AudioClip

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 not currently playing.

Getting ready

Try this with any audio clip that is one second or longer in duration.

How to do it...

1. Create an empty GameObject named *AudioObject*, and add an audio source component to this object
2. Drag an audio clip file from the Project panel to populate the AudioClip parameter of the AudioSource component of *AudioObject*
3. Add the following script class to the Main Camera:

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

public class AvoidSoundRestart : MonoBehaviour{
   public AudioSource myAudioSourceObject;

   private void OnGUI(){
      string statusMessage = "audio source - not playing";
      if( myAudioSourceObject.isPlaying )
          statusMessage = "audio source - playing";

      GUILayout.Label( statusMessage );

      bool buttonWasClicked = GUILayout.Button("send Play() 
message");
      if( buttonWasClicked )
```

```
          PlaySoundIfNotPlaying();
    }

    private void PlaySoundIfNotPlaying(){
       if( !myAudioSourceObject.isPlaying )
          myAudioSourceObject.Play();
    }
}
```

4. With the Main Camera selected in the Hierarchy, drag *AudioObject* into the Inspector for the public AudioSource variable

How it works...

AudioSource components have a public readable property *isPlaying*, which is a Boolean true/false flag indicating if the sound is currently playing. Method PlaySoundIfNotPlaying() includes an 'if' statement ensuring that a Play() message is only sent to the AudioSource component if its *isPlaying* is false.

See also

- Object waits until audio finished playing before destroying itself
- Using Animation events to play audio clips

Object waits until audio finished playing before destroying itself

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 associate 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.

How to do it...

1. Create an empty GameObject named *AudioObject*, and add an audio source component to this object
2. Drag an audio clip file from the Project panel to populate the AudioClip parameter of the AudioSource component of *AudioObject*, and de-select the components “Play On Awake” checkbox.
3. Add the following script class to *AudioObject*:

```
// file: AudioDestructBehaviour.cs

using UnityEngine;
using System.Collections;

public class AudioDestructBehaviour : MonoBehaviour {
   public void DestroyAfterPlaying(){
      selfDestruct = true;
   }

   private bool selfDestruct;

   private void Update()
   {
      if( selfDestruct && !audio.isPlaying )
         Destroy(gameObject);
   }
}
```

4. Add the following script class to the Main Camera:

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

public class AvoidSoundRestart : MonoBehaviour{
   public AudioDestructBehaviour myAudioDestructObect;

   private void OnGUI(){
      bool playButtonWasClicked = GUILayout.Button("play
audio");
      bool destroyButtonWasClicked = GUILayout.Button("destry
audio");

      if( playButtonWasClicked )
         myAudioDestructObect.audio.Play();

      if( destroyButtonWasClicked )
         myAudioDestructObect.DestroyAfterPlaying();
   }
}
```

5. With the Main Camera selected in the Hierarchy, drag *AudioObject* into the Inspector for the public AudioSource variable *myAudioDestructObect.*

How it works...

The GameObject named *AudioObject* contains an AudioSource 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 object has a public method DestroyAfterPlaying(), which when called will set the Boolean true/false flag *selfDestruct* to true. Every frame (via the Update() method) this object tests the value of this flag, and if the flag is true, and the audio source is not playing (!audio.isPlaying) then the GameObect is destroyed. If the audio source is playing, then no action is taken.

The Main Camera scripted object *PlayDestroyButtonGUI* offers 2 buttons to the user, one to send a Play() message to the audio object, the second sends a DestroyAfterPlaying() message. The first button will start (or restart) the playing of the AudioClip inside *AudioObject*, the second will result in the Boolean true/false flag *selfDestruct* being set to true, and therefore the destruction of *AudioObject* as soon as the sound is no longer playing.

See also

- Prevent restarting an already playing AudioClip
- Using Animation events to play named audio clips

Using Animation events to play audio clips

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

Getting ready

You'll find two water droplet sounds in folder *0423_06_06*.

How to do it...

1. Set up a new scene by moving the Main Camera to (0, 1, -3) and add a directional light
2. Create a cube GameObject at (0,0,0) named *AnimatedCube*, and add an AudioSource component to this cube
   NOTE: this cube should be visible at the bottom center of the Game panel with current Unity default new scene settings – if it isn't visible in the Game panel, then move it so that it is!
3. Add the following script class to *AnimatedCube*:

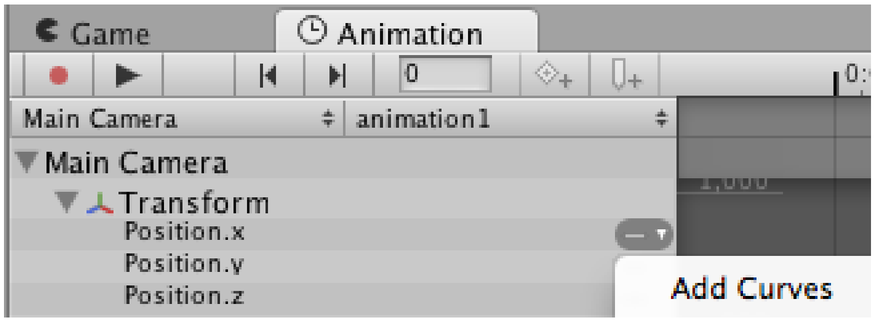
```
// file: CubeAnimMethods.cs
using UnityEngine;
using System.Collections;

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

4. Ensure an Animation panel is visible.

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

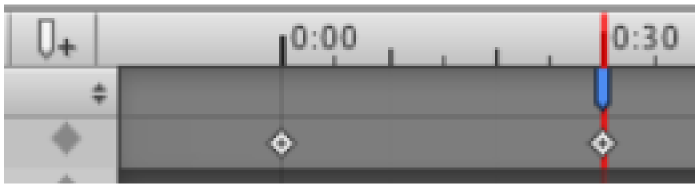
5. First ensure *AnimatedCube* is selected in the Hierarchy, then in the Animation panel create a new animation clip named *animation1*.
6. In the Animation panel, select Add Curves for component: Main Camera | Transform | Position.x

NOTE: Once any position axis is selected, then changes to any of the three dimensions are recorded – red/green/blue diamonds are displayed to indicate property values are being recorded, and a new key-frame (indicated by a small diamond) should have been created at frame 0



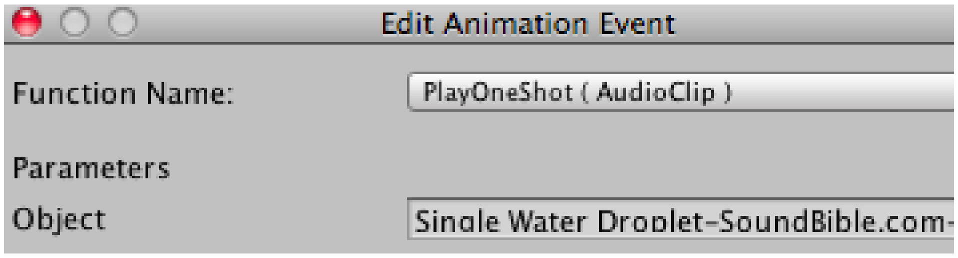
0423_06_13.png

6. Move the play-head to frame 30 (either drag the red line, or change the 0 to 30 in the frame number textbox), and add a key-frame at frame 30 by clicking on the diamond tool with the '+' sign (next to the frame number textbox)
7. Move the cube a to position (1,0,0) (to the right of the camera view)
8. Ensuring the play-head is still at frame 30, add an animation event, by clicking the pencil tool with the '+' sign (next to the new key-frame tool)



0423_06_14.png

9. Edit the blue animation event by clicking it, and from the popup window select the method PlayOneShot(), once selected drag your audio clip into the Parameters slot:



Edit Animation Event

Function Name: PlayOneShot (AudioClip)

Parameters

Object Single Water Droplet-SoundBible.com-

0423_06_15.png

10. Change the repeat setting of the animaton from 'Default' to 'Ping Pong'

How it works...

An animation in an animation clip of a GameObject can send a message to call methods from scripted components of the same GameObject. In this recipe example, the method *PlayOneShot*() was called from an animation event in the animation we created for GameObject *AnimatedCube*. Since this method requires an AudioClip parameter, the animation event editor popup window created a parameter slot into which we could drag the audio clip we wished to be player at that frame.

There's more...

Some details you don't want to miss:

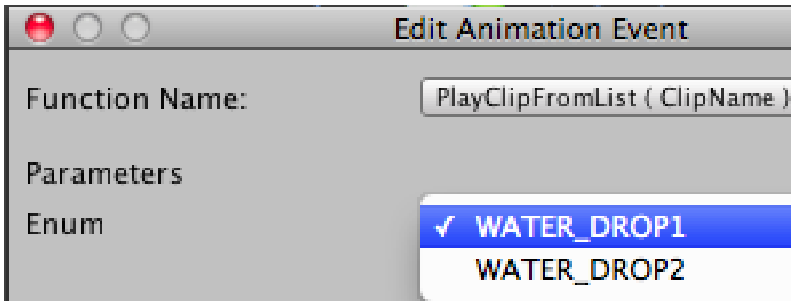
Offering a combo dropdown list of audio clips to choose from

The use of an enumerated type and an extra method, can remove the need for the person editing the animation to have to locate and drag the sound clip into the parameter slot. This approach might be useful either to simplify animation editing for a less technical team member, or to ensure only one from a limited set of audio clips may be played for a given GameObject. The approach works by making the animation event method message parameter an integer from a restricted range (via nicely named enumerations), and then using a 'switch' statement to select one of the public audio clips to actually play. These animation clips variables will then need to be assigned by being dragged into the scripted component via the Inspector as is usual for public variables.

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

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

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

   public enum ClipName {
      WATER_DROP1,
      WATER_DROP2
   }

   public void PlayClipFromList(ClipName clipInt) {
      AudioClip clip = GetClip( clipInt );
```

```
        audio.PlayOneShot(clip);
    }

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



Function Name: PlayClipFromList (ClipName)

Parameters

Enum ✓ WATER_DROP1 WATER_DROP2

0423_06_16.png

See also

- Prevent restarting an already playing AudioClip
- Object waits until audio finished playing before destroying itself

Making a dynamic soundtrack

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 when the player reaches specific targets. Also, we will have the option of fading in and out the sound.

Getting ready

For this recipe, we have prepared a basic level and some soundtrack audio files in *.ogg* format. They are contained inside the Unity package named *DynamicSoundtrack*, which can be found in the folder *0423_06_07*.

How to do it...

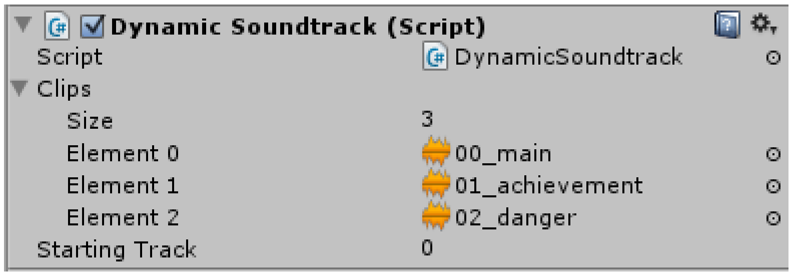
To make a dynamic soundtrack, follow these steps:

1. Import the package *DynamicSoundtrack* into your Unity Project. Also, import the audio files *00_main*, *01_achievement* and *02_danger*.
2. Open the level named *SoundtrackScene*. It should include a basic terrain, a 3rd person character controller and three spheres named *Music Sphere*.
3. In the *Project* view, create a new *C# Script* and rename it as *DynamicSoundtrack.*
4. Open the script in your editor and replace everything with the following code:

```
using UnityEngine;
using System.Collections;

public class DynamicSoundtrack : MonoBehaviour {

public AudioClip[] clips;
public int startingTrack = 0;
private int currentTrack;
private int nextTrack;
private bool  isFadingOut = false;
private float fadeOutTime = 1.0f;
private bool  isFadingIn = false;
private float fadeInTime = 1.0f;
private bool  waitSequence = true;
private bool  keepTime = false;
private float targetVolume = 1.0f;
private float oldVolume = 0.0f;
private float fadeOutStart = 0.0f;
private float fadeInStart = 0.0f;

void  Start (){
audio.clip = clips[startingTrack];
audio.Play();
currentTrack = startingTrack;
}

void  Update (){
if(isFadingOut){
   if(audio.volume > 0){
      float elapsOut = Time.time - fadeOutStart;
      float indOut = elapsOut / fadeOutTime;
        audio.volume = oldVolume - (indOut * oldVolume);
    } else {
      isFadingOut = false;
      StartCoroutine(PlaySoundtrack());
      }
}

if(isFadingIn){
   if (audio.volume < targetVolume) {
        float elapsIn = Time.time - fadeInStart;
      float indIn = elapsIn / fadeInTime;
      audio.volume = indIn;
    } else {
      audio.volume = targetVolume;
      isFadingIn = false;
    }
   }
}

public void  ChangeSoundtrack ( int newClip, bool
waitForSequence , bool keepPreviousTime , float trackVolume ,
float fadeIn , float fadeOutPrevious  ){
nextTrack = newClip;
waitSequence = waitForSequence;
keepTime = keepPreviousTime;
targetVolume = trackVolume;
fadeInTime = fadeIn;

if(newClip != currentTrack){
   currentTrack = newClip;
   if(fadeOutPrevious !=0){
      oldVolume = audio.volume;
      fadeOutStart = Time.time;
      fadeOutTime = fadeOutPrevious;
      isFadingOut = true;
   } else {
      StartCoroutine(PlaySoundtrack());
   }
   }
}
IEnumerator PlaySoundtrack (){
if(waitSequence)
   yield return new WaitForSeconds(audio.clip.length -
(audio.timeSamples / audio.clip.frequency));

if(fadeInTime !=0){
   audio.volume = 0;
   fadeInStart = Time.time;
   isFadingIn = true;
}
float StartingPoint = 0.0f;
if(keepTime)
   StartingPoint = audio.timeSamples;

audio.clip = clips[nextTrack];
audio.timeSamples = Mathf.RoundToInt(StartingPoint);
audio.Play();
}
}
```

NOTE: In case you are wondering why we are using timeSamples instead of time, that's because the former is more accurate when working with compressed audio files. To find out its actual time, we have used the expression audio.timeSamples / audio.clip.frequency. As this is not currently documented in Unity's Scripting Reference, we thank audio engineer Aldo Naletto for the tip.

5. Save your script and attach it to the *Main Camera* by dragging it from the *Project* view to the *Main Camera* game object in the *Hierarchy* view.
6. Select the *Main Camera* and, in the *Inspector* view, access the *Dynamic Soundtrack* component and change the *Size* parameter of the *Clips* variable to 3. Then, drag the sound files *0_Main*, *1_Achievement* and *2_Danger* from the *Project* view into the appropriate slots. Also, type in *0* into the slot named *Starting Track*.

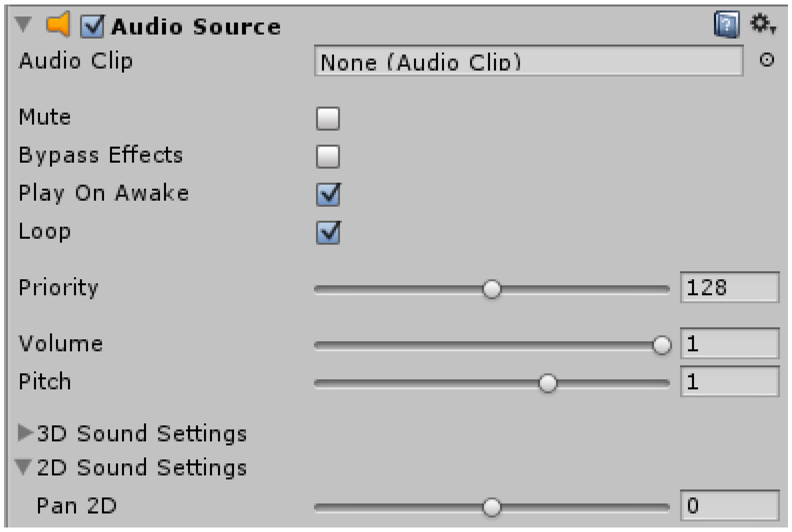


Dynamic Soundtrack (Script)

Script	DynamicSoundtrack
Clips	
Size	3
Element 0	00_main
Element 1	01_achievement
Element 2	02_danger
Starting Track	0

0423_06_17.png

7. Now, access the *Audio Source* component and make sure the option *Loop* is on.



Audio Source

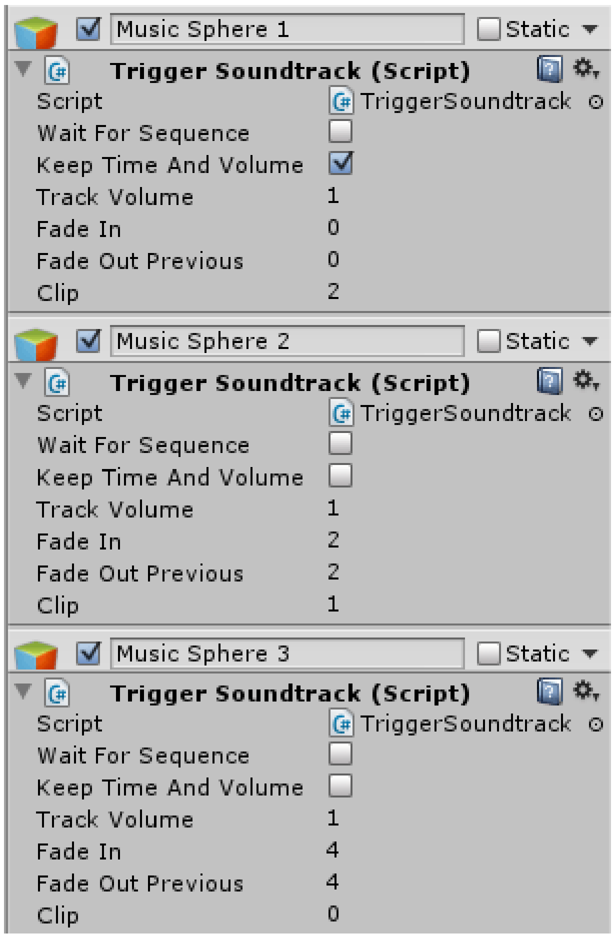
Audio Clip	None (Audio Clip)
Mute	☐
Bypass Effects	☐
Play On Awake	✓
Loop	✓
Priority	128
Volume	1
Pitch	1
▶3D Sound Settings	
▼2D Sound Settings	
Pan 2D	0

0423_06_18.png

8. In the *Project* view, create a new C# *Script* and name it *TriggerSoundtrack*.
9. Open the script in your editor and replace everything with the following code:

```
using UnityEngine;
using System.Collections;

public class TriggerSoundtrack : MonoBehaviour {
public bool  waitForSequence = true;
public bool  keepTimeAndVolume = false;
public float trackVolume = 1.0f;
public float fadeIn = 0.0f;
public float fadeOutPrevious = 0.0f;
public int clip;

void  OnTriggerEnter ( Collider other  ){
     if (other.gameObject.CompareTag ("Player")) {

Camera.main.GetComponent<DynamicSoundtrack>().ChangeSoundtrack
(clip,waitForSequence,keepTimeAndVolume, trackVolume,
fadeIn,fadeOutPrevious);
      }
}
}
```

10. Save your script and attach it to the each one of the *Music Spheres* by dragging it from the *Project* view to the *Main Camera* game object in the *Hierarchy* view.
11. Select each *Music Sphere* and, in the *Inspector* view, change the *Trigger Soundtrack* parameters as in the images below.



✓ Music Sphere 1	☐ Static
Trigger Soundtrack (Script)	
Script	TriggerSoundtrack
Wait For Sequence	☐
Keep Time And Volume	✓
Track Volume	1
Fade In	0
Fade Out Previous	0
Clip	2
✓ Music Sphere 2	☐ Static
Trigger Soundtrack (Script)	
Script	TriggerSoundtrack
Wait For Sequence	☐
Keep Time And Volume	☐
Track Volume	1
Fade In	2
Fade Out Previous	2
Clip	1
✓ Music Sphere 3	☐ Static
Trigger Soundtrack (Script)	
Script	TriggerSoundtrack
Wait For Sequence	☐
Keep Time And Volume	☐
Track Volume	1
Fade In	4
Fade Out Previous	4
Clip	0

0423_06_19.png

12. Play your scene and direct the character towards each *Music Sphere*. The background music will change accordingly.

How it works...

We have created two different scripts. The one attached to the *Main Camera*, *DynamicSoundtrack* , is responsible for keeping a list of the audio files that make up all the soundtrack for the level. Also, it contains all the functions that control the audio playback, volume transition, etc. The second one, *TriggerSoundtrack*, is attached to the

Music Spheres and triggers soundtrack changes based on the preferences expressed in that component's parameters. They are:

- Wait for Sequence: Leave this option checked if you want to wait the end of the previous audio clip before playing the new part.
- Keep Time And Volume: Leave it checked to start a new audio clip from the same point where the previous clip was at. Also keeps the volume from the previous clip.
- Track Volume: The volume for the new audio clip (from 0.0 to 1.0).
- Fade In: The amount of seconds that will take for the new audio clip's volume to fade in.
- Fade Out Previous: The amount of seconds that will take for the previous audio clip's volume to fade out.

There's more...

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

Hide the triggers

If having milestone objects as triggers feels too obvious for you and your players, you can always make it invisible by disabling the *Mesh Renderer* component.

Audio File Formats and compression issues

To avoid loss of audio quality, you should import your sound clips using the appropriate file format, depending on 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`

Use 2D sound

To make the sound volume and balance independent of the Audio Source position, make sure your audio clip is not set up as *3D Sound* (you can check it out the *Import Settings* in the *Inspector* view, by selecting the file in the *Project* view).