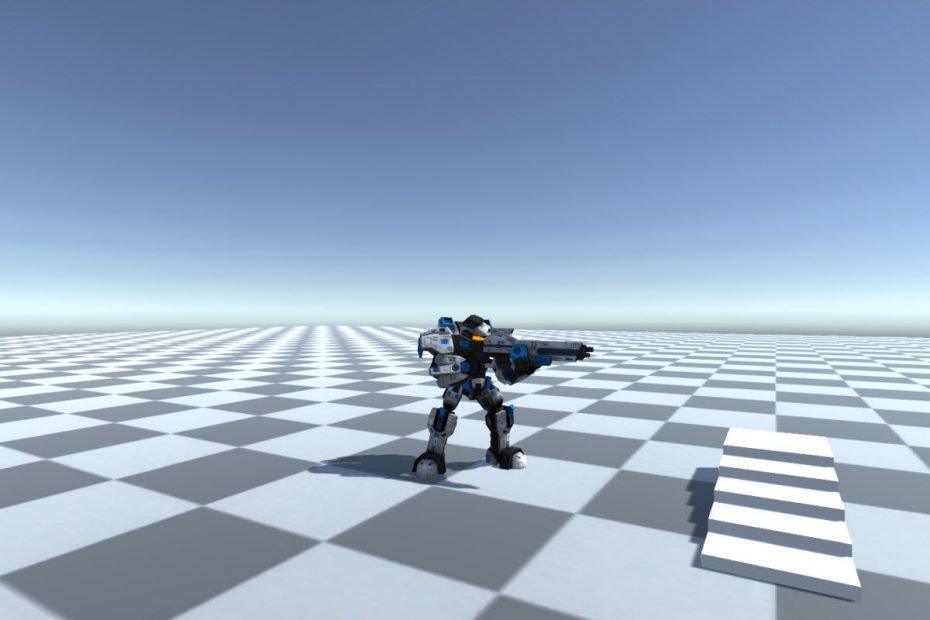
# Implementing Audio using C# in Unity



Shamim Akhtar

## Introduction

Clone and get the source code for the base project from <https://github.com/shamim-akhtar/pgge/tree/Week07>

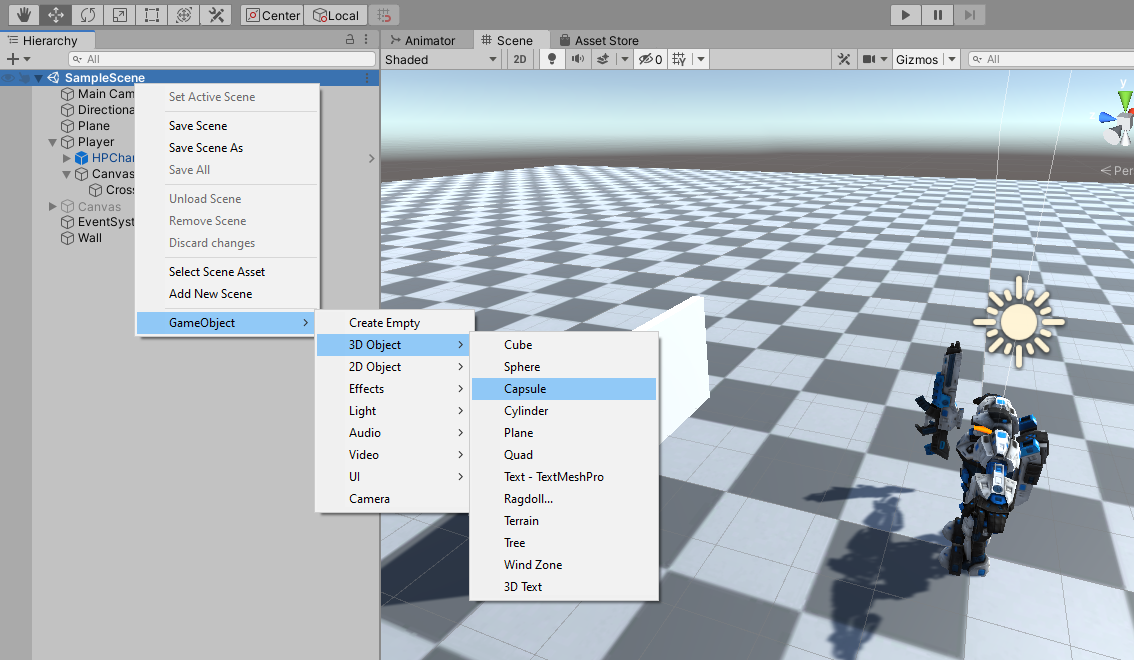
The project is a continuation from your last worksheet. In this section, we will learn about interfaces and then use Unity audio to play sound.

## Implementing Shooting Bullets

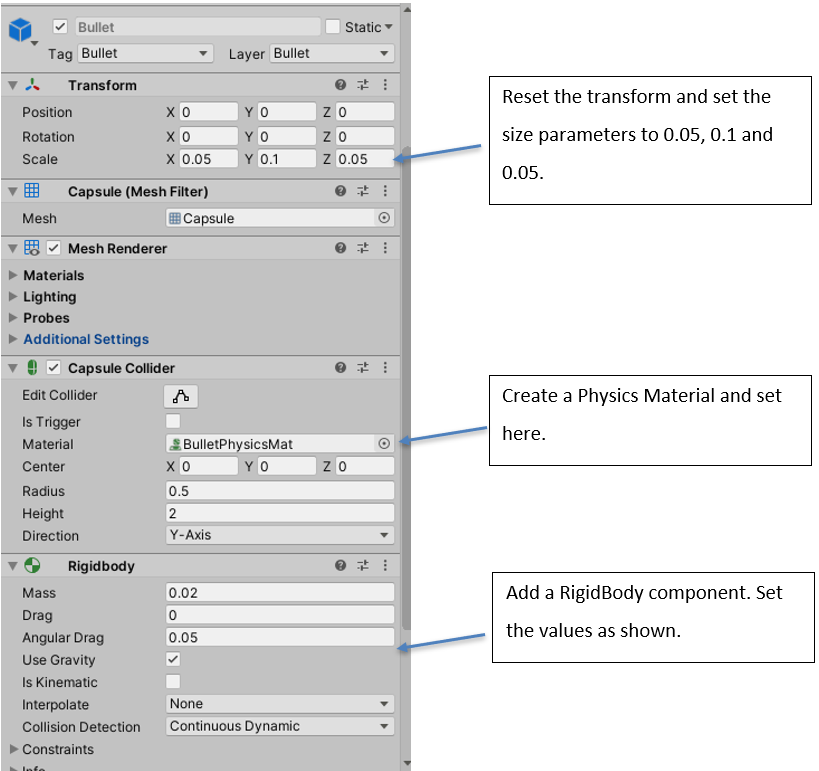
In this section, we will create the actual shooting of bullets. In the previous sections, we have made the structure of attack and aiming of the gun. Here will implement the real bullet game object and shooting of bullets.

### The Bullet Game Object

Right on the scene and add a capsule game object. Rename it to **Bullet**. We will use capsule shape for the bullet. You can create your own bullet 3D model using Maya or blender. However, for this demonstration, a capsule object will do just fine.



By default, the capsule is big. We will need to resize it to make the bullet look real. Select the **Bullet** game object and go to Inspector.



Carry out the steps shown in the figure above. Note that the Collision Detection mechanism for the bullet rigid body component is set to **Continuous Dynamic**. This is because bullets are fast-moving objects. If you set to the default value of **Discrete,** then it will miss some collision.

### The Bullet Script

Select the **Bullet** game object from the scene hierarchy. Go to the Inspector and add a New Script component called **Bullet**. The script file **Bullet.cs** will by default appear in **Assets** folder. Drag and move to it **Scripts** folder for convenience.

Double click and open the file in Visual Studio.

Add the **OnCollisionEnter** method.

private void OnCollisionEnter(Collision collision)

{

}

Unity calls **OnCollisionEnter** when this collider or rigid body has begun touching another collider or rigid body. We will use this method when the bullet hits an object. The parameter **collision** of type **Collision** class contains information, for example, about contact points, the game object and impact velocity. For more information, look at [Unity’s documentation](https://docs.unity3d.com/ScriptReference/Collider.OnCollisionEnter.html).

We will come back to this function later.

You can now drag and drop the Bullet game object into your **Assets->Resources->Prefabs** folder and make it a prefab. Once you have made it as a prefab, you can delete the Bullet game object from the scene.

### The IDamageable Interface

We will use an interface to implement the damage created by bullets when they hit any game object. Different game objects might have different damage implementation. And, when programming in Unity, it's easy to overcomplicate your code, which in turn can become harder to maintain the more you add to it. In our case, we can keep adding codes in **OnCollisionEnter** method of our Bullet script for each type of object that it hits. However, that will be very difficult to manage and maintain as you proceed with your game. Soon you will see that many different types of game objects exist that require damage handling. To our rescue, we can simplify this by implementing a C# Interface.

An interface contains a definition of a method(s) or variable(s) that the class which uses it must implement. For our game, we will create an **IDamageable** interface. It will have just one method:

void TakeDamage();

Go ahead and create a new C# file in your Scripts folder and name it **IDamageable.cs.** The file contains the following:

public interface IDamageable

{

void TakeDamage();

}

Any object that takes damage will implement from this interface. Now, go back to your **OnCollisionEnter** method of **Bullet.cs** and implement the functionality of calling **TakeDamage** when a bullet hits a game object.

private void OnCollisionEnter(Collision collision)

{

IDamageable obj = collision.gameObject.GetComponent<IDamageable>();

if(obj != null)

{

obj.TakeDamage();

}

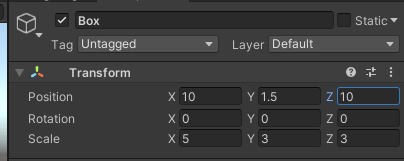
}

In the implementation above, you can see that we get the **IDamageable** component of the hit game object. Only if the game object has this component, then we handle the **TakeDamage**. For other game objects, we simply ignore.

Now, let’s go and create some objects in the scene which can take damage from the bullets.

#### Box

Right-click on the project hierarchy and create a 3D cube. Name it Box. Select the Box game object, go to the Inspector and reset the transform. Now set the values as shown below.



Add a new script called **Box.cs** to this game object.

Double click and open the **Box.cs** in Visual Studio.

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Box : MonoBehaviour, IDamageable

{

// Start is called before the first frame update

void Start()

{

}

// Update is called once per frame

void Update()

{

}

public void TakeDamage()

{

Debug.Log("Box: I am hit by a bullet!")

}

}

Modify the script to make it implement the IDamageable interface. Then implement the **TakeDamage** method. For now, we will simply write to Debug.Log.

However, for actual implementation, depending on whether the box is a metal box, wooden box or other types of box, you might want to play the bullet hit sound, you might want to add a decal, switch to a damage model, show special effect etc.

Again, different objects in the scene can have other implementations of the **TakeDamage** function.

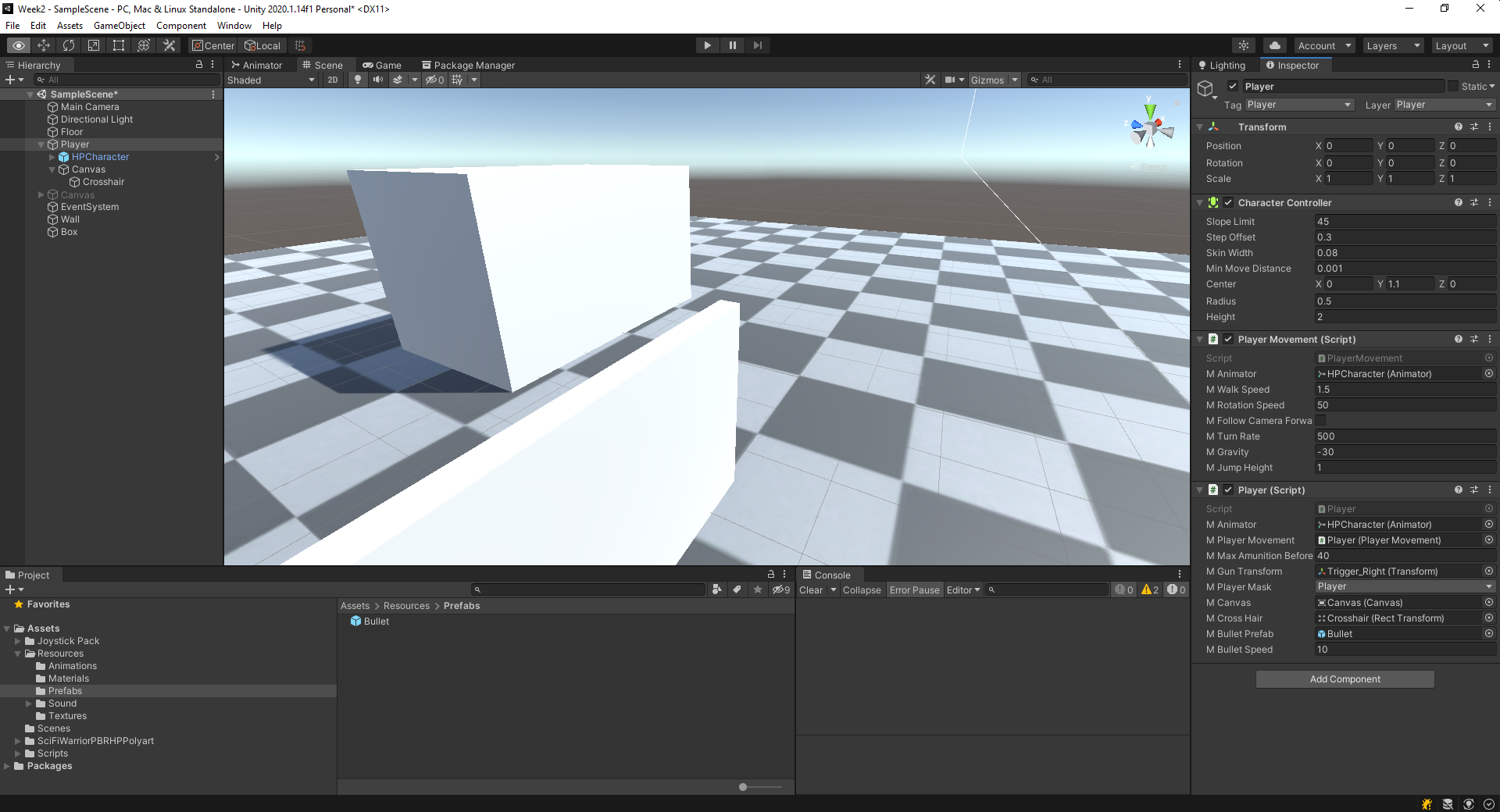
We shall now implement the shooting of the bullets. Double click and open the Player script.

Add the following variables.

public GameObject mBulletPrefab;

public float mBulletSpeed = 10.0f;

Go to Editor and associate the bullet prefab to M Bullet Prefab field.



Create a new function called FireBullet, as shown below.

public void FireBullet()

{

if (mBulletPrefab == null) return;

Vector3 dir = -mGunTransform.right.normalized;

Vector3 firePoint = mGunTransform.transform.position + dir \*

1.2f - mGunTransform.forward \* 0.1f;

GameObject bullet = Instantiate(mBulletPrefab, firePoint,

Quaternion.LookRotation(dir) \* Quaternion.AngleAxis(90.0f, Vector3.right));

bullet.GetComponent<Rigidbody>().AddForce(dir \* mBulletSpeed, ForceMode.Impulse);

}

Call this function from the **Fire** function.

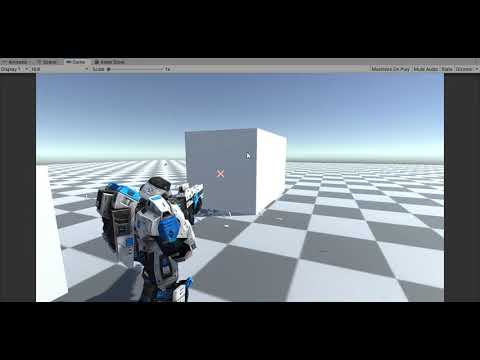
public void Fire(int id)

{

FireBullet();

}

Click **Play** and see the behaviour.

[](https://www.youtube.com/embed/8fSQ8W-iKtU?feature=oembed)

The video of our shooting bullet implementation. However, there are a few problems with this implementation.

Discuss in class some of these problems and how to solve these problems.

### Synchronizing Shooting with Animation

We have identified that one of the problems with our above implementation of shooting is that the firing does not synchronize with the animation. You can also see that the number of bullets coming out of the gun is quite large in quantity.

In this section, we will try to solve this problem of synchronization.

We will introduce two new variables, called **RoundsPerSecond** and **mFiring**. Both are arrays of type **float** and **boolean**, respectively. In **RoundsPerSecond** array, we will hold the number of rounds the gun fires bullets per second. The number will vary for the three different firing types. In the **mFiring** array, we will set the value of individual firing to true if that specific firing is currently happening.

public int[] RoundsPerSecond = new int[3];

bool[] mFiring = new bool[3];

We will then use Coroutine to handle correct timing of firing based on the RoundPerSecond. Now, let’s implement the Coroutine.

IEnumerator Coroutine\_Firing(int id)

{

mFiring[id] = true;

FireBullet();

yield return new WaitForSeconds(1.0f / RoundsPerSecond[id]);

mFiring[id] = false;

mBulletsInMagazine -= 1;

}

Now, we change the Fire method of the Player by calling the above Coroutine if the firing of this id is not already in place, meaning if the **mFiring[id] == false**.

public void Fire(int id)

{

if(mFiring[id] == false)

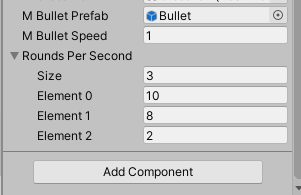
{

StartCoroutine(Coroutine\_Firing(id));

}

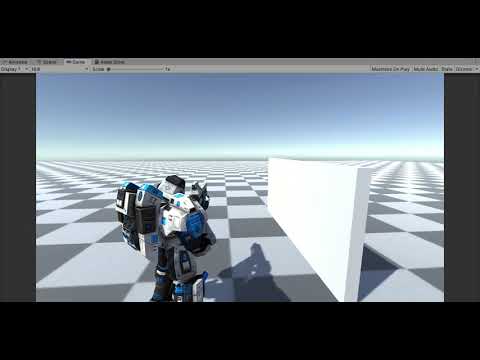
}

Go to Unity editor and set the values for RoundsPerSecond as shown in the picture below.



The **Size** field is 3 as there are three types of attacks (or firing modes). The other values you can try to change and see the effect in **Play** mode.

Click **Play** and see the behaviour. Doesn’t it look slightly more realistic now?

[](https://www.youtube.com/embed/CiHaY3YL5gE?feature=oembed)

### Removing a Bullet After it Hits

The other problem that we have identified is bullets lying around after it hits. We should be removing a bullet game object from the scene after some time if the bullet did not hit any object. We should also be removing the bullet after it hits an object; maybe not immediately but after a specific duration of time.

We can achieve this by writing a Coroutine.

IEnumerator Coroutine\_Destroy(float duration)

{

yield return new WaitForSeconds(duration);

Destroy(gameObject);

}

The above Coroutine will call the **Destroy** method after a specific duration of time. You could create this Coroutine at the **Start** method with a longer time, let’s say 10 seconds. Then, again in OnCollisionEnter, you can create another Coroutine with a shorter time, let’s say 0.1 seconds.

Go ahead and amend **Bullet.cs** script with the above changes.

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Bullet : MonoBehaviour

{

void Start()

{

// Destroy the bullet after 10 seconds if it does not hit any object.

StartCoroutine(Coroutine\_Destroy(10.0f));

}

void Update()

{

}

IEnumerator Coroutine\_Destroy(float duration)

{

yield return new WaitForSeconds(duration);

Destroy(gameObject);

}

private void OnCollisionEnter(Collision collision)

{

IDamageable obj = collision.gameObject.GetComponent<IDamageable>();

if(obj != null)

{

obj.TakeDamage();

}

StartCoroutine(Coroutine\_Destroy(0.1f));

}

}

Now, click **Play** and see the behaviour.

### Implement Reload State

Double click and open **PlayerState** in Visual Studio. Go to class PlayerState\_RELOAD.

Add the following code in the **Enter** method.

public override void Enter()

{

mPlayer.mAnimator.SetTrigger("Reload");

mPlayer.Reload();

dt = 0.0f;

}

In the **Update** method, we check if the time has exceeded the **ReloadTime**. If so, we go back to the **MOVEMENT** state.

public override void Update()

{

dt += Time.deltaTime;

if (dt >= ReloadTime)

{

mPlayer.mFsm.SetCurrentState((int)PlayerStateType.MOVEMENT);

}

}

Finally, we will have to ensure proper counting of bullets after a Reload. We will do that in the **Exit** method. Note that we invoke the **Exit** method when the state completes and transits to another state.

public override void Exit()

{

if (mPlayer.mAmunitionCount > mPlayer.mMaxAmunitionBeforeReload)

{

mPlayer.mBulletsInMagazine += mPlayer.mMaxAmunitionBeforeReload;

mPlayer.mAmunitionCount -= mPlayer.mBulletsInMagazine;

}

else if (mPlayer.mAmunitionCount > 0 && mPlayer.mAmunitionCount < mPlayer.mMaxAmunitionBeforeReload)

{

mPlayer.mBulletsInMagazine += mPlayer.mAmunitionCount;

mPlayer.mAmunitionCount = 0;

}

}

Now, click **Play** and see the behaviour by shooting bullets till the Player reloads.

### The Bullet Script and Adding the Collision Event

Select the **Bullet** prefab from the Prefabs folder. Next, go to the Inspector and add a New Script component called **Bullet**. The script file **Bullet.cs** will, by default, appear in the **Assets** folder. Drag and move to the **Scripts** folder for convenience.

Double-click and open the file in Visual Studio.

Add the **OnCollisionEnter** method.

private void OnCollisionEnter(Collision collision)

{

}

Unity calls **OnCollisionEnter** when this collider or rigid body has begun touching another collider or rigid body. For example, we will use this method when the bullet hits an object. The parameter **collision** of type **Collision** class contains information, for instance, about contact points, the game object and impact velocity. For more information, look at [Unity’s documentation](https://docs.unity3d.com/ScriptReference/Collider.OnCollisionEnter.html).

We will come back to this function later.

You can now drag and drop the Bullet game object into your **Assets->Resources->Prefabs** folder and make it a prefab. Once you have made it as a prefab, you can delete the Bullet game object from the scene.

### The IDamageable Interface

We will use an interface to implement the damage created by bullets when they hit any game object. Of course, different game objects might have other damage implementations. And, when programming in Unity, it's easy to overcomplicate your code, which can become harder to maintain the more you add to it. In our case, we can keep adding codes in **the OnCollisionEnter method of our Bullet script for each type of objec**t it hits. However, that won't be easy to manage and maintain as you proceed with your game. Soon you will see that many different types of game objects exist that require damage handling. To our rescue, we can simplify this by implementing a C# Interface.

An interface contains a definition of a method(s) or variable(s) that the class which uses it must implement. For our game, we will create an **IDamageable** interface. It will have just one method:

void TakeDamage();

Please create a new C# file in your Scripts folder and name it **IDamageable.cs.** The file contains the following:

public interface IDamageable

{

void TakeDamage();

}

Any object that takes damage will implement from this interface. Now, go back to your **OnCollisionEnter** method of **Bullet.cs** and implement the functionality of calling **TakeDamage** when a bullet hits a game object.

private void OnCollisionEnter(Collision collision)

{

IDamageable obj = collision.gameObject.GetComponent<IDamageable>();

if(obj != null)

{

obj.TakeDamage();

}

}

The implementation above shows that we get the **IDamageable** component of the hit game object. If the game object has this component, we handle the **TakeDamage**. For other game objects, we ignore them.

Now, let’s go and create some objects in the scene which can take damage from the bullets.

#### Box

Right-click on the project hierarchy and create a 3D cube. Name it, Box. Select the Box game object, go to the Inspector and reset the transform. Now set the values as shown below.

Graphical user interface

Description automatically generated

Add a new script called **Box.cs** to this game object.

Double-click and open the **Box.cs** in Visual Studio.

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Box : MonoBehaviour, IDamageable

{

// Start is called before the first frame update

void Start()

{

}

// Update is called once per frame

void Update()

{

}

public void TakeDamage()

{

Debug.Log("Box: I am hit by a bullet!")

}

}

Modify the script to make it implement the IDamageable interface. Then implement the **TakeDamage** method. For now, we will write to Debug.Log.

However, for your implementation, depending on whether the box is a metal box, wooden box or another type of box, you could play the bullet hit sound, show some special effects such as show a decal, switch to a damage model, or trigger a particle system.

Again, different objects in the scene can have other implementations of the **TakeDamage** function.

Game Audio

## Implementing The Sound Effects

No game can have a feeling of reality without sound effects. So, in this section, we will add sound effects to our game. But before we start with our sound effects, let’s import a map to make the entire game look more realistic. Double-click and open the scene Week8\_Industrial

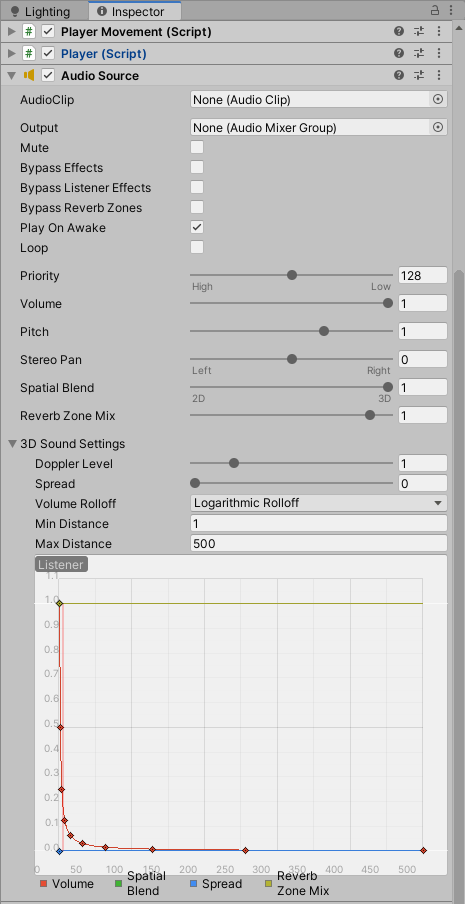
Click **Play** and see the behaviour.

[](https://www.youtube.com/embed/bS3_dyB43eY?feature=oembed)

### Firing, Reload and NoAmmo Sound

In this section, we will add the bullet fire sound. In addition, I have included some audio files that you can use for your sound effects.

Select the **Player** game object from the scene hierarchy. Next, go to Inspector and add an Audio Source component.



Set the Spatial Blend to 3D

Open the **Player.cs** script in Visual Studio. Add the following variables.

public AudioSource mAudioSourceFiring;

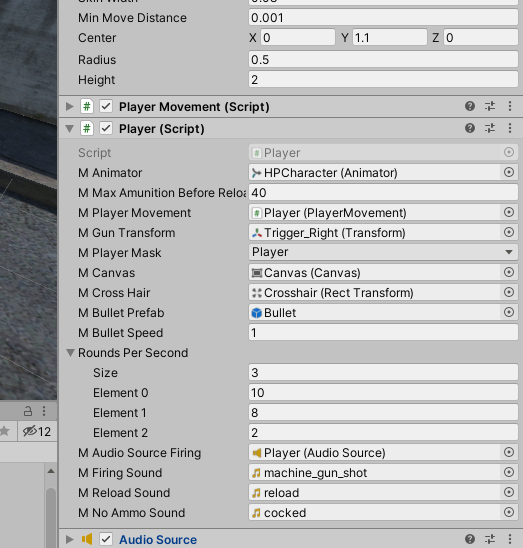
public AudioClip mFiringSound;

public AudioClip mReloadSound;

public AudioClip mNoAmmoSound;

Dictionary<AudioClip, bool> mSoundStatus = new Dictionary<AudioClip, bool>();

Go to Unity editor, select **Player** game object and then drag and drop the Audio Source component to the **M Audio Source Firing** field. Next, go to **Assets->Sounds->gunshot** and drag and drop **machine\_gun\_shot.wav**, **reload.wav** and **cocked.wav** to the **M Firing Sound**, **M Reload Sound** and **M No Ammo Sound** fields, respectively, as shown below.



Dictionary<AudioClip, bool> mSoundStatus = new Dictionary<AudioClip, bool>();

The variable **mSoundStatus** is a dictionary to set the status of whether a specific AudioClip is playing. For example, not calling PlayOneShot again on the same audio clip is necessary when it is already playing.

Now, add the following line of codes to the **Player** **Start** method.

mSoundStatus.Add(mFiringSound, false);

mSoundStatus.Add(mReloadSound, false);

mSoundStatus.Add(mNoAmmoSound, false);

The above lines of code set the default status for the three audio clips to be false, which means they are currently not in Play.

### Coroutine to Call PlayOneShot

We will use a coroutine to call **AudioSource.PlayOneShot**.

IEnumerator Coroutine\_PlayOneShot(AudioClip audioClip)

{

mSoundStatus[audioClip] = true;

mAudioSourceFiring.volume = 0.8f;

mAudioSourceFiring.pitch = 0.8f;

mAudioSourceFiring.PlayOneShot(audioClip);

yield return new WaitForSeconds(audioClip.length);

mSoundStatus[audioClip] = false;

}

As shown in the code above, we set the status for the AudioClip to true while it is playing. After it has finished playing, we set the status to false.

Amend the function **FireBullet** by adding the below code.

public void FileBullet()

{

\*\*\*\*\*

\*\*\*\*\*

if (mSoundStatus[mFiringSound] == false)

{

StartCoroutine(Coroutine\_PlayOneShot(mFiringSound));

}

}

Similarly, amend the **Reload** and **NoAmmo** methods.

public void Reload()

{

if (mSoundStatus[mReloadSound] == false)

{

StartCoroutine(Coroutine\_PlayOneShot(mReloadSound));

}

}

public void NoAmmo()

{

if (mSoundStatus[mNoAmmoSound] == false)

{

StartCoroutine(Coroutine\_PlayOneShot(mNoAmmoSound));

}

}

### Programming Task 1 – Implement bullet hit sound for metallic wall game objects in the scene.

Now go ahead and implement a bullet hit sound for metallic objects.

## Implementing Ambient Sound

Ambient sound is critical in all media related to moving images, video games, and live performances. It defines its place and time, temporalizes it towards a future goal and is vital in creating audience immersion and belief in what we see.

Ambient sound is an often undervalued area of video game sound. It's the canvas upon which all the spot effects, dialogue, and music sit. Still, it also has a much more powerful interactive potential. Unfortunately, ambience typically tends to be loops of one or two minutes in duration, which is generally inactive and doesn't draw any attention to itself.

Ambient sound requires audio across multiple scenes. For example, while the game shows the menu, you will play background music. Then as the scene loads and transits to another scene, you will play another set of music. Implementing such a mechanism would mean you will have to persist a game object across multiple scenes. You should also ensure that there is only one such game object instance across numerous scenes.

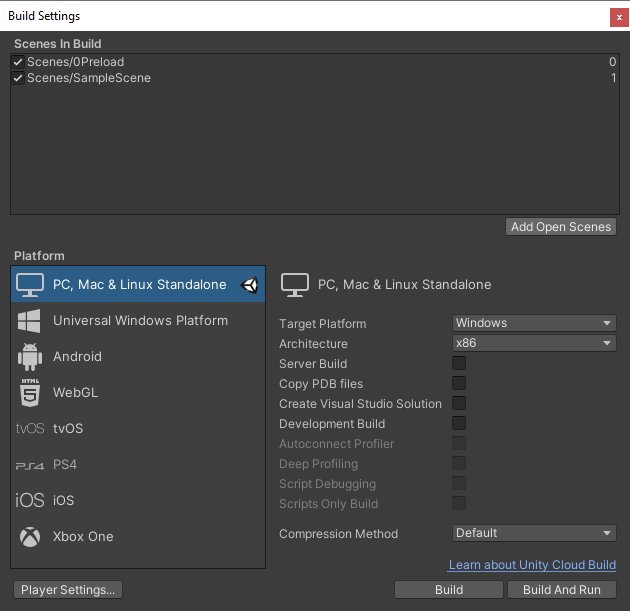
Remember any pattern that we can make use of here? **The Singleton Pattern**!

The singleton pattern is one of the best-known patterns in software engineering. A singleton is a class that only allows a single instance of itself and usually gives simple access to that instance.

Usually, in a game, we should maintain one singleton object and then add different scripts such as AmbientSound, GameManager and so on to this singleton object.

### Add Singleton Object Called GameApp

A singleton object must persist across scenes. So, we will create a new scene called **0Preload**. **0Preload** should be the first scene that a game should load. First, open the 0Preload scene by double-clicking on the **0Preload** scene. Then, go to Build Settings and add this scene by clicking Add Open Scene.



Right-click on the scene hierarchy and create an empty game object. Name it, **GameApp**. Add a new script to this object and name it **GameApp**. For convenience, drag and drop the script from the **Assets** folder to the **Scripts** folder. Then, double-click and open the **GameApp** file in Visual Studio.

The first thing we want to do is to make it a singleton class. There are many ways to implement a singleton in C#. We will choose one of the most simple singleton implementations. I have created a singleton class for you in the namespace **Patterns**.

You need to derive your class from this **Singleton**<You class type> to create your singleton implementation.

public class GameApp : Singleton<GameApp>

Now, you can use **GameApp** as a singleton object. For now, we do not have any other use for this **0Preload** scene rather than to create the singleton object. So in the **Start** method, we will load the next scene in the build order.

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using PGGE.Patterns;

using UnityEngine.SceneManagement;

public class GameApp : Singleton<GameApp>

{

void Start()

{

SceneManager.LoadScene(1);

}

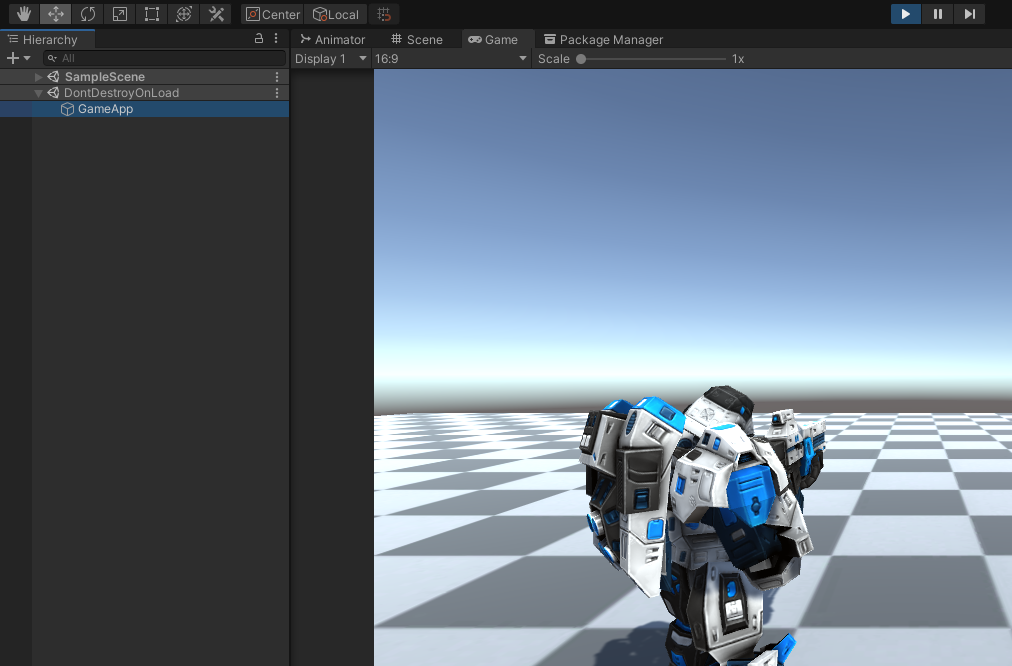
void Update()

{

}

}

Now when you click **Play,** you should see the **DontDestroyOnLoad** object.



A DontDestroyOnLoad object of name GameApp is created. This object persists across muliple scenes.

We can now add our AmbientSound into this singleton object.

### Add Audio Source Component

Select the **GameApp** game object and add a new Audio Source component from Inspector.

### Add AmbientSound Component

Select the GameApp game object, go to the Inspector and add another new script component. Name it **AmbientSound**. For convenience, drag and drop the script from the **Assets** folder to the **Scripts** folder.

Now, double-click and open the **AmbientSound** file in Visual Studio.

Add a public variable called **mAudioSource**.

public class AmbientSound : MonoBehaviour

{

public AudioSource mAudioSource;

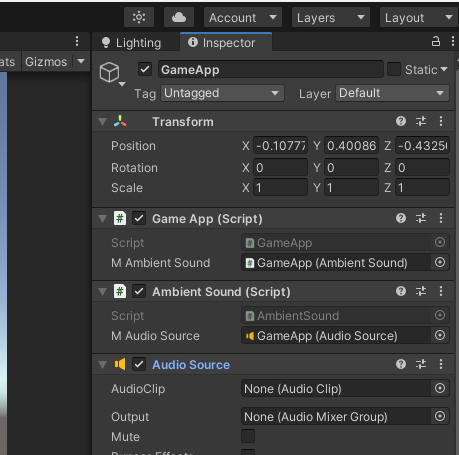
private void Start()

{

}

}

In Unity Editor, drag the Audio Source component from the **GameApp** game object to the **M Audio Source** field.



Drag and drop the Audio Source component to the M Audio Source field.

Add a method called **Play** that takes in an AudioClip, the pitch value and the volume value.

public void Play(AudioClip clip, float volume = 1.0f, float pitch = 1.0f, bool loop = true)

{

if (mAudioSource.isPlaying)

mAudioSource.Stop();

mAudioSource.clip = clip;

mAudioSource.pitch = pitch;

mAudioSource.volume = volume;

mAudioSource.loop = loop;

mAudioSource.Play();

}

Finally, expose the AmbientSound object using a public variable in GameApp to access the object from other scripts.

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using PGGE.Patterns;

using UnityEngine.SceneManagement;

public class GameApp : Singleton<GameApp>

{

public AmbientSound mAmbientSound;

void Start()

{

SceneManager.LoadScene(1);

}

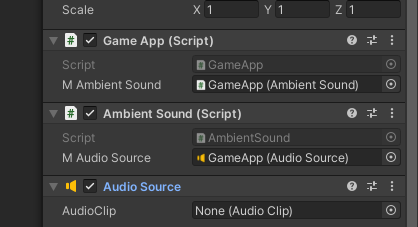
void Update()

{

}

}

Drag and drop the AmbientSound component from the Unity Editor to the **M Ambient Sound** field.



Drag and drop the Ambient Sound component to the M Ambient Sound field.

We have now implemented a basic ambient sound framework using the **AmbientSound** component. We shall now implement a scene-level sound by adding AudioClips specific to a scene.

### Implement a Random Ambient Sound for a Scene

Double-click and load the **SampleScene**. Next, add an empty game object and name it **SceneSound**. Next, select the **SceneSound** game object from the Hierarchy, go to the Inspector and add a new script component called **SceneSound**.

Double-click the **SceneSound** script file and open it in Visual Studio.

For our worksheet, we will implement a simple scene sound that randomizes an index from a list of audio clips and plays for a specific duration and continues with the same process endlessly.

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class SceneSound : MonoBehaviour

{

public AudioClip[] mAudioClips;

public float mMinSoundDuration = 10.0f;

public float mMaxSoundDuration = 20.0f;

private float mCurrentSoundDuration = 1.0f;

private AmbientSound mAmbientSound;

void Start()

{

mAmbientSound = GameApp.Instance.mAmbientSound;

StartCoroutine(WaitForDuration ());

}

void PlayAmbientSound()

{

if (mAudioClips.Length > 0)

{

int index = Random.Range(0, mAudioClips.Length - 1);

mCurrentSoundDuration = Random.Range(mMinSoundDuration, mMaxSoundDuration);

mAmbientSound.Play(mAudioClips[index]);

Debug.Log("Playing sound " + index.ToString() + " for " + mCurrentSoundDuration + " secs");

}

}

public IEnumerator WaitForDuration()

{

while (true)

{

yield return new WaitForSeconds(mCurrentSoundDuration);

PlayAmbientSound();

}

}

}

Go to Unity Editor and set some audio clips for the SceneSound. You can find the sound files in **Resources->Sound->ambient** folder.

Click **Play** and see the behaviour. Discuss in class.

### Programming Task 3 – Implement audio fade-in and fade-out.

You have seen in the above implementation that the sound stops abruptly and starts abruptly, which sounds very odd. How can we improve on it? A straightforward way to improve it is to implement a sound fade-in and fade-out by controlling the volume. In this assignment, you will implement one such mechanism. Discuss in class how you can implement this.

See “[**How to fade audio in Unity: I tested every method, this one’s the best**](https://gamedevbeginner.com/how-to-fade-audio-in-unity-i-tested-every-method-this-ones-the-best/#second_method)” article on implementing such behaviour.