# Implementing Player Controls using C# in Unity



Shamim Akhtar

## Introduction

In this worksheet, we will proceed to an intermediate level of programming exercises by implementing the Player states. We will touch on the concepts of coroutines, interfaces, delegates and some design patterns.

We will start from where we left last week. We will do some refactoring in our code. Then, we will implement the aiming, shooting and the physical bullet game object.

## Code Refactoring

In this section, we will learn about code refactoring. We will do a hands-on session on code refactoring based on what we did in Worksheet 1 and Worksheet 2. We will start from where we left in Week 1. Download the Unity project from your Teams/LMS and load it in Unity.

### What is code refactoring?

In computer programming and software design, code refactoring is the process of restructuring existing computer code—changing the factoring—without changing its external behaviour. The intention for refactoring is to improve the design, structure, and implementation of the software while preserving its functionality.

### Refactor GameConstants

The objective for this refactoring is to bring out **GameConstants** class into a new file. We will call this file **GameConsants.cs**. We will rename the class to **CameraConstants** as we know that the values in this class only affects the Camera. We will also put our classes in namespace **PGGE** for clarity of use. See below the implementation for **GameConstants.cs.**

using System;

using System.Collections.Generic;

using UnityEngine;

namespace PGGE

{

public static class CameraConstants

{

public static Vector3 CameraAngleOffset { get; set; }

public static Vector3 CameraPositionOffset { get; set; }

public static float Damping { get; set; }

public static float RotationSpeed { get; set; }

public static float MinPitch { get; set; }

public static float MaxPitch { get; set; }

}

}

**[Do a quiz to confirm your understanding.](https://forms.office.com/Pages/ResponsePage.aspx?id=8JupJXKOKkeuUK373w328SOl3R0X2XFDvrpvcEtyjmpURDVGRlBUREpYOTRONTdMTEpXVVNHVFFGUi4u)**

### Refactor ThirdPersonCamera

In this section, you will refactor the various classes for the third-person Camera into separate files. Bring each of these classes into PGGE namespace and individual files. Keeping each class in a separate file makes it organized.

1. Create a file called TPCBase.cs. Cut the TPCBase class from ThirdPersonCamera.cs and bring it over to this newly created file. Use the namespace PGGE.
2. Similarly create separate files each for TPCTrack, TPCFollow, TPCFollowTrackPosition, TPCFollowTrackPositionAndRotation, TPCTopDown and TPCFollowIndependentRotation.

Make sure that you click Play and check if your application is still behaving correctly. Ideally, it would help if you created test cases to test whether your application behaves correctly after you make changes to it.

**[Do a quiz to confirm your understanding.](https://forms.office.com/Pages/ResponsePage.aspx?id=8JupJXKOKkeuUK373w328SOl3R0X2XFDvrpvcEtyjmpUOUQxQTdFNUxWSTNQWjJOVENKM0QyMlE2VC4u)**

### Refactor PlayerMovement

In our previous implementation, we have put most of our code within the Update method. We will now refactor it so that we organize our code well.

**Discuss in class on how you can do that.**

We will try to distribute the functionality of our code in separate functions. You can see that in the **Update** method, we have two distinct sections. The first is to get the inputs, and the second is to apply the movement based on these inputs.

Based on the above justification, we shall now create two separate functions. The first function we shall call **HandleInputs** and the second function we shall call **Move**.

Go ahead and add these two functions in **PlayerMovement** script.

private void HandleInputs()

{

// We shall handle our inputs here.

}

private void Move()

{

// We shall apply movement to the game object here.

}

To save our values across these two functions, we shall now move the local variables hInput, vInput and speed to class variables.

private float hInput;

private float vInput;

private float speed;

Then we move the section of code that handles inputs from **Update** to **HandeInputs**

private void HandleInputs()

{

// We shall handle our inputs here.

#if UNITY\_STANDALONE

hInput = Input.GetAxis("Horizontal");

vInput = Input.GetAxis("Vertical");

#endif

#if UNITY\_ANDROID

hInput = 2.0f \* mJoystick.Horizontal;

vInput = 2.0f \* mJoystick.Vertical;

#endif

speed = mWalkSpeed;

if (Input.GetKey(KeyCode.LeftShift))

{

speed = mWalkSpeed \* 2.0f;

}

}

After that, we move the section of code that handles the actual movement of the game object from **Update** to **Move** method.

private void Move()

{

// We shall apply movement to the game object here.

if (mAnimator == null) return;

if (mFollowCameraForward)

{

// rotate Player towards the camera forward.

Vector3 eu = Camera.main.transform.rotation.eulerAngles;

transform.rotation = Quaternion.RotateTowards(

transform.rotation,

Quaternion.Euler(0.0f, eu.y, 0.0f),

mTurnRate \* Time.deltaTime);

}

else

{

transform.Rotate(0.0f, hInput \* mRotationSpeed \* Time.deltaTime, 0.0f);

}

Vector3 forward = transform.TransformDirection(Vector3.forward).normalized;

forward.y = 0.0f;

mCharacterController.Move(forward \* vInput \* speed \* Time.deltaTime);

mAnimator.SetFloat("PosX", 0);

mAnimator.SetFloat("PosZ", vInput \* speed / (2.0f \* mWalkSpeed));

}

Now from the **Update** method, we simply call the **HandleInputs** and **Move** methods.

void Update()

{

HandleInputs();

Move();

}

**Discuss in class how did this refactoring of code help us?**

You can do several other code refactoring. But, for now, we will leave it till here. You get the idea. Of course, as we move on, we will be doing refactoring on the go.

Make sure that you click **Play** and check if your application is still behaving correctly. Ideally, it would help if you created test cases to test whether your application behaves correctly after you make changes to it.

**[Do a quiz to confirm your understanding.](https://forms.office.com/Pages/ResponsePage.aspx?id=8JupJXKOKkeuUK373w328SOl3R0X2XFDvrpvcEtyjmpURUhPTlRURUI0VlJINFFUNFNGNE9VMzFLUC4u)**

## Configure the Player

In our previous week’s worksheet, we have added the basic movement animations for the Player. In this week’s worksheet, we will add some more animations so that we can create our Player controls.

### Attack Animations

We will add three attacks (in this case shooting) animations for this character, viz., **Attack1, Attack2 and Attack3**. We will use the following animations for our attacks.

\* Shoot\_AutoShot\_AR.fbx,

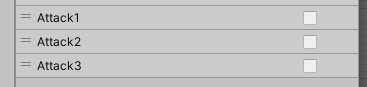
\* Shoot\_BurstShot\_AR.fbx, and

\* Shoot\_SingleShot\_AR.fbx

We will use these three animations and tie them to Fire1 (left mouse button or the left ctrl key), Fire2 (the left alt key) and Fire3 (the left shift key). Do note that in this tutorial we will target PC, MAC & Linux Standalone Build. We will not target this demo for Android build for now.

Go ahead and open the **PlayerAnim**animator in Unity.

Add three Boolean parameters called **Attack1**, **Attack2**and **Attack3**.



Add the three animations of shooting into the Animator. We will create the Transitions later.

### Reload and Die Animations

Add two new motions called **Reload**and **Die**into the Animator. The asset comes with **Reload.fbx** and **Die.fbx** motion files. After that, add two new Trigger parameters called **Reload** and **Die**.



### Jump and Crouch Animations

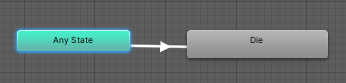
Finally, we add two new animations in the Animator called **Jump**and **Crouch**with the **Jump.fbx** and **Idle\_Ducking\_ar.fbx** respectively. We will also need to add a new boolean parameter called **Crouch**.



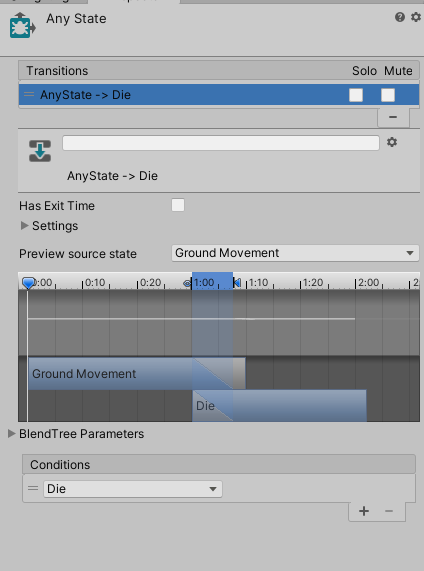
### Animation Transitions

#### Die Animation Transition

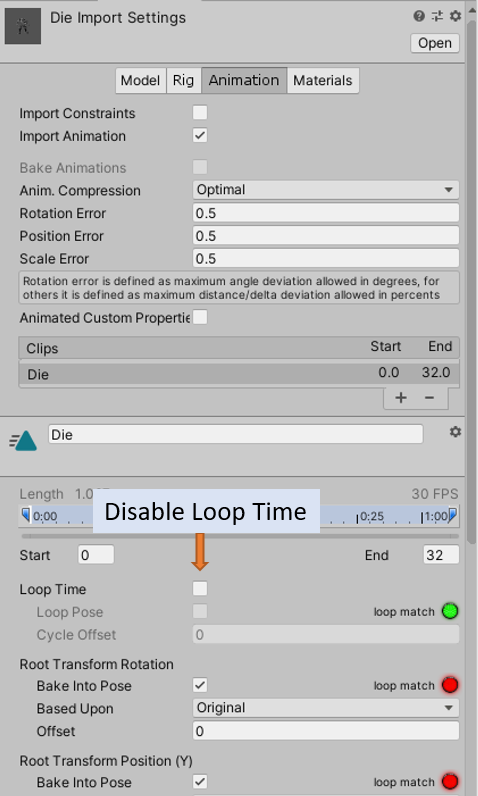
Now we are ready to create the Animation Transition. **Die**animation can transition from **Any State** by calling the Trigger **Die**.



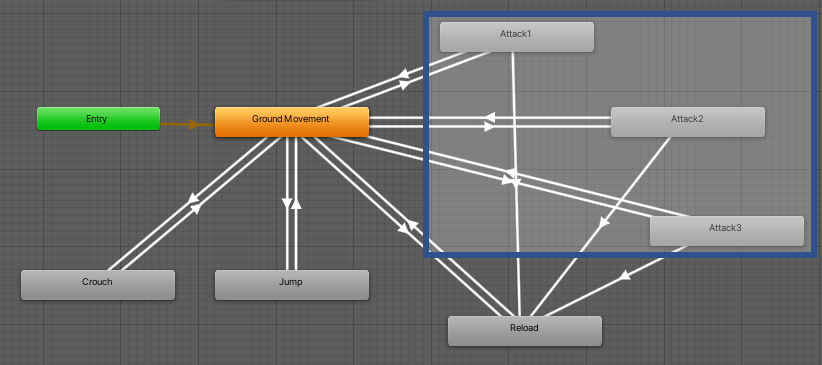
Disable Has Exit Time.



Ensure that **Die**animation is not looping. If it is looping then select the **Die**animation and deselect the **Loop Time**shown below. Click on Apply to apply the changes.

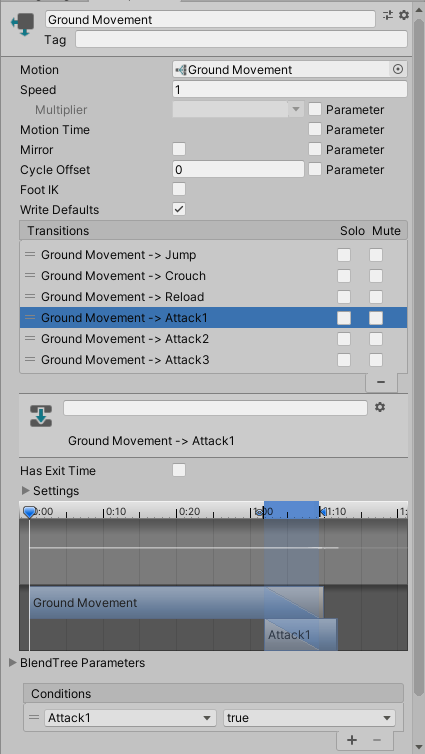


#### Attack Animation Transitions

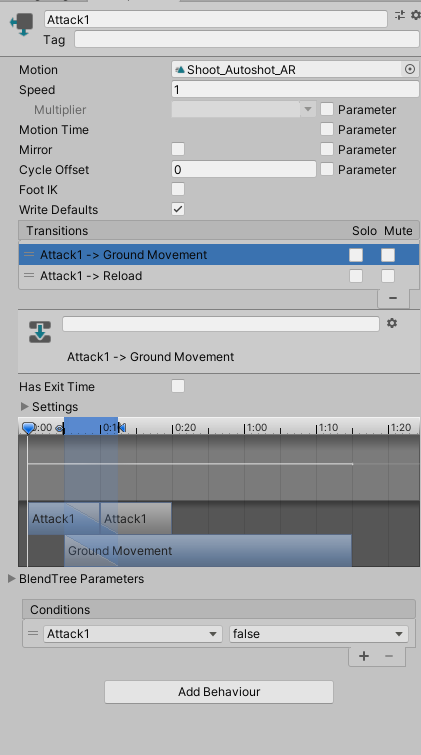


All attack animations can transition from the **Ground Movement** blend state.

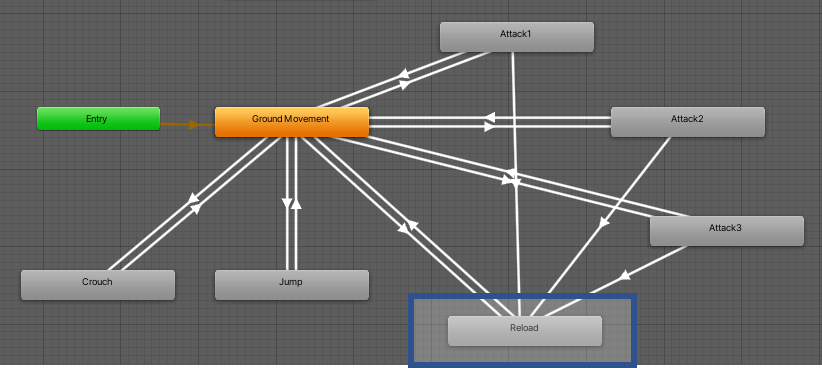
**Attack1, Attack2**or**Attack3**animations are transitioned based on setting the Boolean parameters **Attack1**, **Attack2**or **Attack3**as **true**.



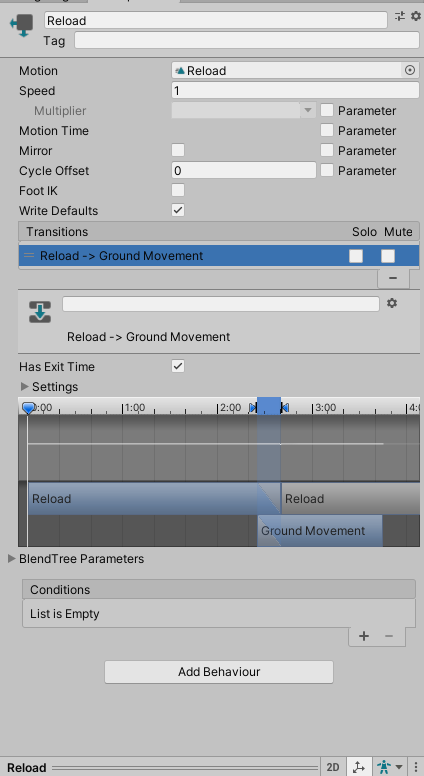
When these parameters are set to **false**the animation transitions back to **Ground Movement** blend state. Do ensure that **Has Exit Time**is disabled for both transitions.



#### Reload Animation Transition



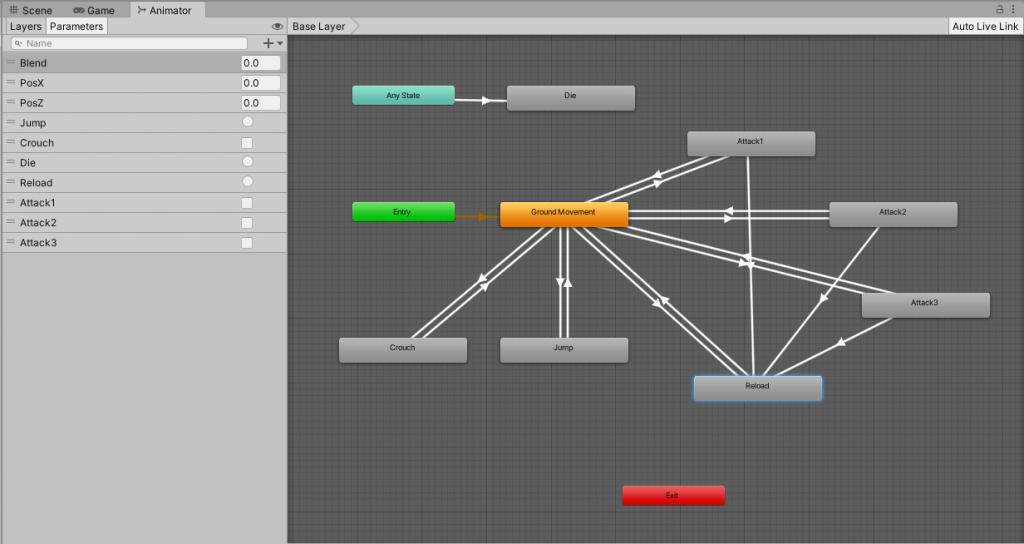
The **Reload**animation transition happens from all Attack animations when we execute **Reload**trigger. The Reload animation is played and then transits to **Ground Movement** blend state animation.



Ensure to check the **Has Exit Time** checkbox as you want this animation to exit after one complete animation run. Also, disable looping for this motion sequence.

#### Crouch and Jump Animations

Finally, add the **Crouch**using the **Idle\_Ducking\_ar.fbx** and **Jump**using the **Jump.fbx** animations. Make the transition to and from **Ground Movement** blend state. For crouch, you will add a Boolean parameter called Crouch which when true will show the **Crouch**animation and when false show the **Ground Movement** animations. Look below for the final Animator.



## Add Gravity in PlayerMovement

We are using the CharacterController. CharacterController doesn’t have a RigitBody so we will have to create a simple gravity-based movement. Without gravity, our player will not fall if the floor is below its feet. For example, if you are walking along a pathway that is going down then without gravity, our Player won’t go down along with the floor. It would remain at the same height.

To implement gravity, we will add two variables in **PlayerMovement**. These are

public float mGravity = -30.0f;

private Vector3 mVelocity = new Vector3(0.0f, 0.0f, 0.0f);

Then we create a function called **ApplyGravity**. In this function, we will implement how the game object is affected by gravity.

void ApplyGravity()

{

// apply gravity.

mVelocity.y += mGravity \* Time.deltaTime;

mCharacterController.Move(mVelocity \* Time.deltaTime);

if (mCharacterController.isGrounded && mVelocity.y < 0)

mVelocity.y = 0f;

}

We will call the **ApplyGravity** function from the **FixedUpdate** method.

private void FixedUpdate()

{

ApplyGravity();

}

**Discuss why we used FixedUpdate to calculate this rather than generic Update method?**

## Add Jump and Crouch in PlayerMovement

We have added our Jump and Crouch animations. We will now use our script to trigger and enable disable the jump and crouch, respectively.

We will add the following variables in PlayerMovement class.

private bool jump = false;

private bool crouch = false;

public float mJumpHeight = 1.0f;

Then in HandleInputs we will check if we press the correct keys to trigger jump or to enable/disable crouch animations.

private void HandleInputs()

{

\*\*\*\*

if (Input.GetKeyDown(KeyCode.Space))

{

jump = true;

}

if (Input.GetKeyUp(KeyCode.Space))

{

jump = false;

}

if (Input.GetKeyDown(KeyCode.Tab))

{

crouch = !crouch;

Crouch();

}

}

**Go ahead and implement the Jump and Crouch functions. Call the Jump function from within the Move method. I will not include this section of code here in this worksheet. Try it out. Discuss in class if you face any issues.**

## Understanding The Player States

I have provided a finite state machine in the Unity Project. You can find the state machine in **FSM.cs**. The source codes of the state machine are in the namespace **PGGE.Patterns**. You will learn details about implementing a state machine in your **GamAI subject next semester**. That’s the reason why we are not going to implement a state machine here in this subject but only make use of it. However, it will be a good idea to know at least what a state machine does and how we can use a state machine to represent the various Player states.

For more information about the state pattern and the finite-state machine, refer to the comments. I have provided several references.

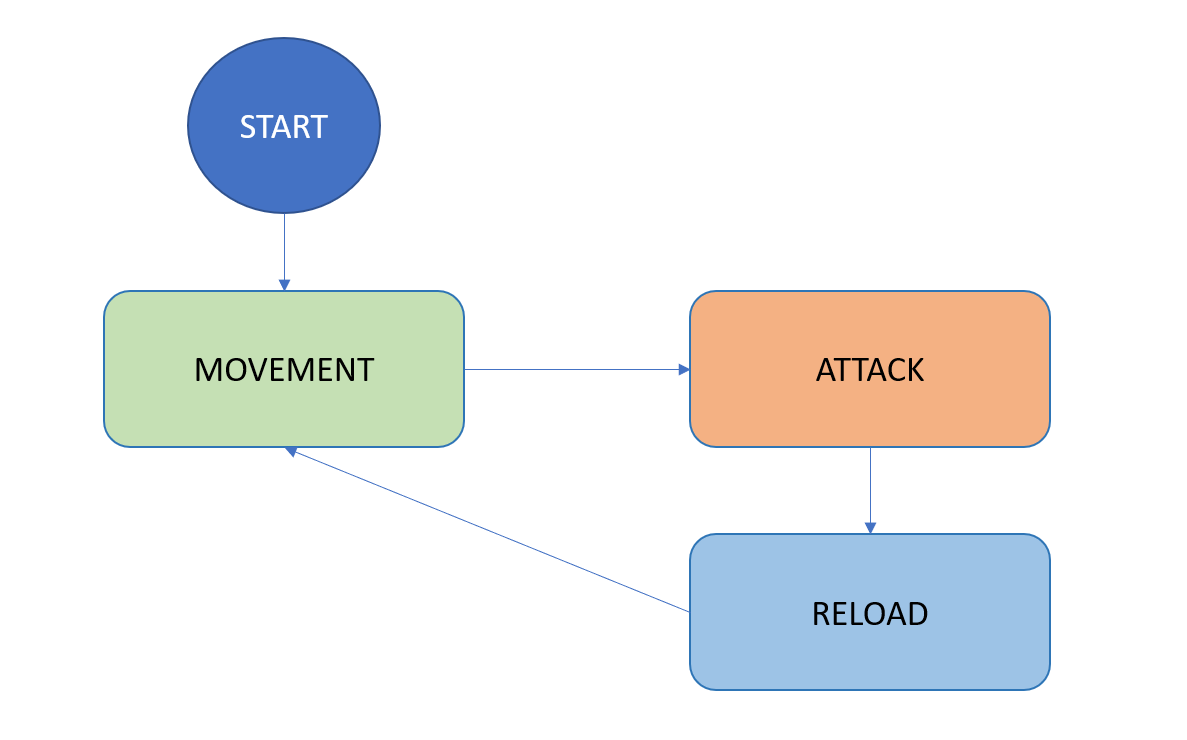
The state pattern is a behavioural software design pattern that allows an object to alter its behaviour when its internal state changes.

The state pattern is used in computer programming to encapsulate varying behaviour for the same object, based on its internal state. Using state pattern can be a cleaner way for an object to change its behaviour at runtime without resorting to conditional statements and thus improve maintainability.

I have implemented the Player states in the **PlayerState.cs**. It comprises the enumeration type called **PlayerStateType**, the **PlayerState** class and the three concrete **Player** states viz., the **PlayerState\_MOVEMENT**, **PlayerState\_ATTACK** and the **PlayerState\_RELOAD**. Before I explain these classes, let me show you the state diagram for the different player states.

### Player State Diagram

State machine diagram typically is used to describe state-dependent behaviour for an object. The below diagram below represents the states that our Player can have. Of course, there can be many other states like DIE, DAMAGE, etc., but for our implementation, for now, will only include the **MOVEMENT**, **ATTACK** and **RELOAD** states.



### MOVEMENT State

The **MOVEMENT** state is the default state of the Player. It comprises the idle, the walk and the run animation states.

**Note** that there is a difference between animation states and the player states. Animation states are purely different animations that a character can have. You can configure the animations and transitions in the Animator for Unity. Player state, on the other hand, is more like the behaviour of the Player. One player state sometimes can have more than one animation states.

In the MOVEMENT state, represented by the class PlayerState\_MOVEMENT, the Player either stay idle or walks and runs around the scene.

The Player can transit to the **ATTACK** state from this **MOVEMENT** state. Can you think how this transition from **MOVEMENT** to **ATTACK** can take place?

### ATTACK State

The **ATTACK** state is the state where the Player does the shooting/firing of the gun. The transition from **MOVEMENT** to **ATTACK** takes place either by clicking the mouse buttons (the **Fire1**, **Fire2** and **Fire3** Unity buttons) or by clicking on the UI buttons for mobile touch screens. Of course, we will need to follow some logic on the shooting of the bullets. For example, if there are enough bullets in the gun’s magazine. If not then the Player transits to our next state which is **RELOAD** state. If there is any bullet left in the magazine, then the Player shoots bullets at a specific rate. The rate of firing is dependent on which button is clicked.

### RELOAD State

In the RELOAD state, the Player does a reload of bullets into the magazine. The **RELOAD** state activates when there are no bullets left in the magazine during the **ATTACK** state. Also, note that a RELOAD cannot happen if the total ammunition that the Player has reaches to zero. In that case, the Player can neither **ATTACK** nor **RELOAD**.

We will understand more about the logic when we implement the shooting logic in the next section.

## Implementing The Logic of Shooting Bullets

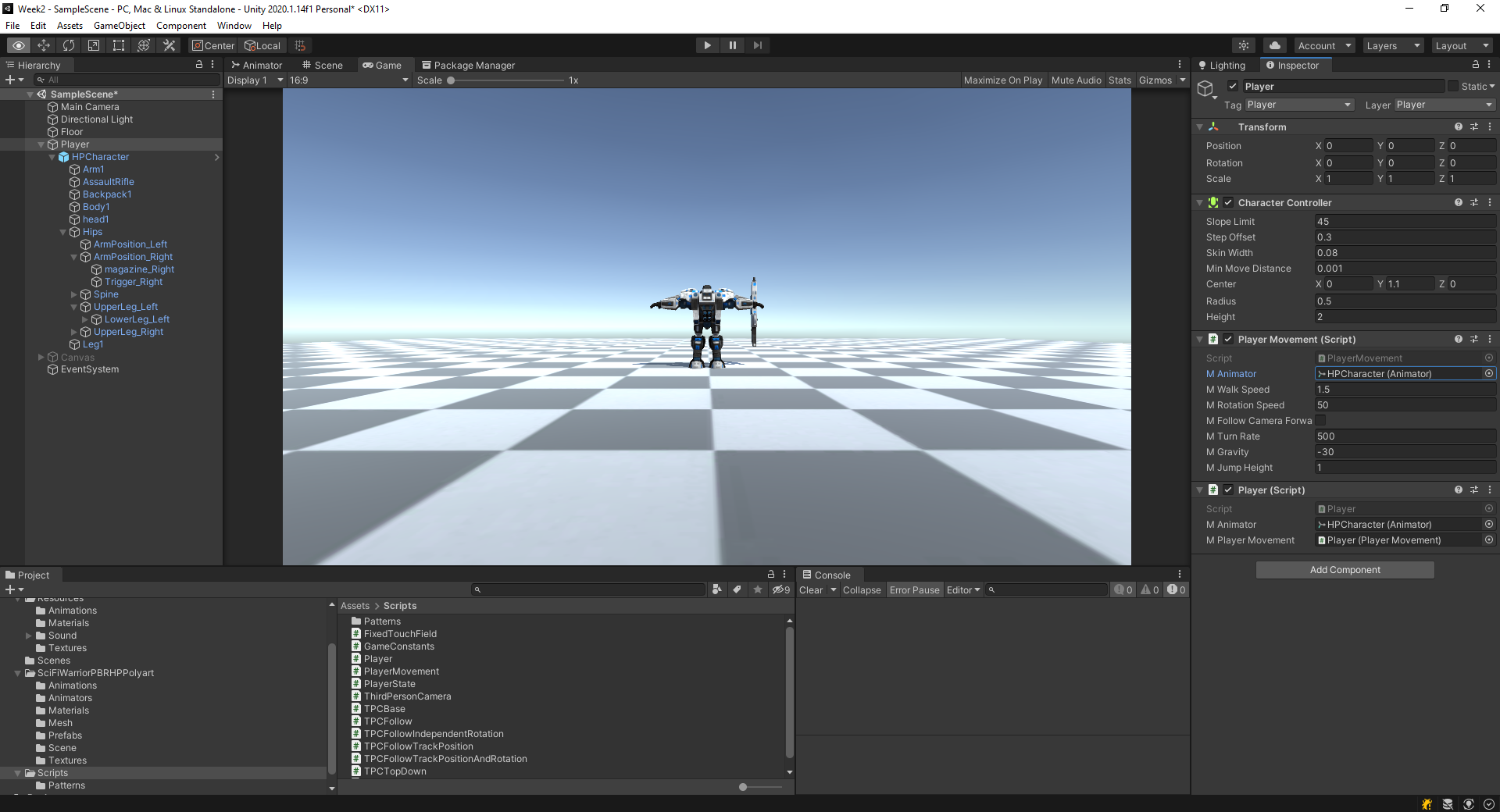
In this section, we will implement the logic of shooting bullets. We are not implementing any real bullets yet. We are going to cover creating and shooting bullets in a different section. In this section, the implementation will emphasize the logic of state transition and bullet counting. The implementation encompasses all three players states. Before we do that we will need to make some changes to the script structure. Right now, we have only one script called **PlayerMovement.cs** that is attached to the Player. This script, as the name suggests, is responsible for taking inputs and moving the Player.

At this juncture, I will introduce to you the **Player.cs** script. **Player.cs** script is now going to be the central script for the Player. This script will have references to the **PlayerMovement.cs** and any other scripts that we will create for the Player later. Doing this way is one of the critical factors of good software design, concentrating specific functionality in specific classes. This way, if later, we change the behaviour of player movement, then we will only have to change the implementation of **PlayerMovement.cs** and will not affect any other components of the Player.



Go ahead and drag and drop the **Player.cs** to the Player game object.

After that drag and drop the Player Movement component to the **M Player Movement** field of Player and drag and drop the HPCharacter to the **M Animator** field.



Now, double click and open the PlayerState script and open in Visual Studio. Look at the **Update** method of **PlayerState\_MOVEMENT** class.

### Programming Task 1 – Implement the Player Movement and transition from MOVEMENT to ATTACK

In this programming assignment, you will implement the **Update** method of the **PlayerState\_MOVEMENT** class.

public override void Update()

{

base.Update();

// For Student ---------------------------------------------------//

// Implement the logic of player movement.

//----------------------------------------------------------------//

// Hint:

//----------------------------------------------------------------//

// You should remember that the logic for movement

// has already been implemented in PlayerMovement.cs.

// So, how do we make use of that?

// We certainly do not want to copy and paste the movement

// code from PlayerMovement to here.

// Think of a way to call the Move method.

//

// You should also

// check if fire buttons are pressed so that

// you can transit to ATTACK state.

mPlayer.Move();

}

I have provided hints on what you should implement in the code above. Discuss with your tutor in class on how you would go about implementing the method. Your objectives for this assignment are:

(1) Associate the player movement from **PlayerState\_MOVEMENT** state, and

(2) Allow transition from **MOVEMENT** to **ATTACK** state based on inputs (**Fire1, Fire2 and Fire3**).

### Programming Task 2 – Implement the attack and transitions from ATTACK to RELOAD and ATTACK to MOVEMENT

After implementing your programming assignment 1, you will notice that once we transit from **MOVEMENT** to **ATTACK**, we cannot revert to **MOVEMENT** or to **RELOAD**. That is the reason you see the shooting animation keep repeating. This phenomenon happens because we did not implement the **PlayerState\_ATTACK** **Update** method. In this programming assignment, you will implement the **PlayerState\_ATTACK** **Update** method. Your objectives are:

1. Implement the logic to reload ammunition and transit to **RELOAD** state,

Notice that we have three variables in **Player.cs** script, viz., **mAmunitionCount,**  **mBulletsInMagazine** and **mMaxAmunitionBeforeReload**. You will need to make use of these variables while implementing the transition to **RELOAD**. Specifically, you will have to **RELOAD** when the number of bullets in your magazine goes to zero, i.e., **mBulletsInMagazine** count goes to 0. Once both **mBulletsInMagazine** and **mAmunitionCount** reaches to 0 your should transit to **MOVEMENT**

1. Handle three different types of attack mode and stay in **ATTACK** state as long as only of the Fire buttons are clicked, and

You should remain in **ATTACK** state as long as the Fire buttons are pressed. During **ATTACK** state, you should trigger the correct **ATTACK** animation based on which button is pressed. You can access this through the AttackID. Every bullet shot should reduce the count of **mAmunitionCount** and **mBulletsInMagazine**. Once **mBulletsInMagazine** reaches to 0 you should transit to RELOAD state.

1. To revert to **MOVEMENT** state when no attack buttons are clicked.

You should transit to **MOVEMENT** state if you do not press any of the **Fire** buttons.

Look at the function below. I have provided you with hints on how to implement it. Go ahead and give it a try.

Discuss in class if you face any difficulty.

public override void Update()

{

base.Update();

// For Student ---------------------------------------------------//

// Implement the logic of attack, reload and revert to movement.

//----------------------------------------------------------------//

// Hint:

//----------------------------------------------------------------//

// 1. Transition to RELOAD

// Notice that we have three variables, viz.,

// mAmunitionCount

// mBulletsInMagazine

// mMaxAmunitionBeforeReload

// You will need to make use of these variables while

// implementing the transition to RELOAD.

//

// 2. Staying in ATTACK state

// You should stay in ATTACK state as long as the

// Fire buttons are pressed. During ATTACK state

// you should trigger the correct ATTACK animation

// based on which button is pressed and shoot bullets.

// Every bullet shot should reduce the count of mAmunitionCount

// and mBulletsInMagazine.

// Once mBulletsInMagazine reaches to 0 you should

// transit to RELOAD state.

//

// 3. Transition to MOVEMENT state

// You should transit to MOVEMENT state when any of the

// following two situations happen.

// First you have exhausted all your bullets, that means your

// mAmunitionCount is 0 or if you do not press any of the

// Fire buttons.

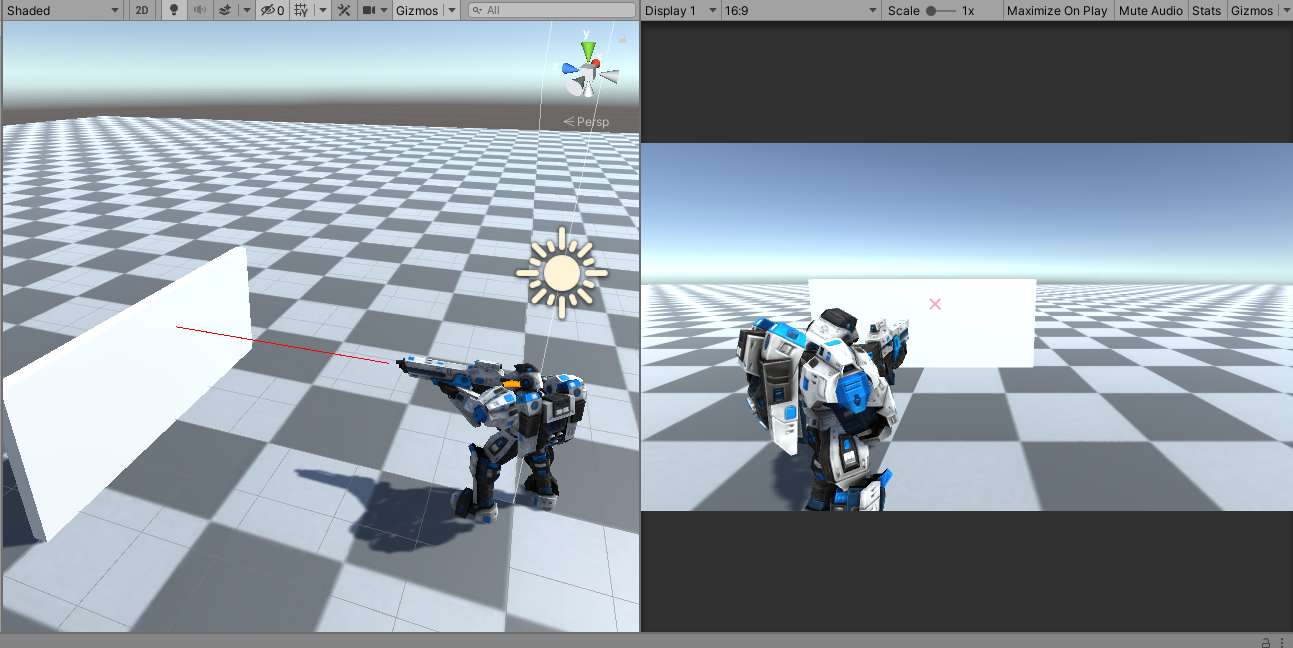
// Discuss with your tutor if you find any difficulties

// in implementing this section.

}

## Implementing The Cross Hair And Aiming

In this section, we will implement the aiming of the gun, and showing a crosshair at the point of impact.



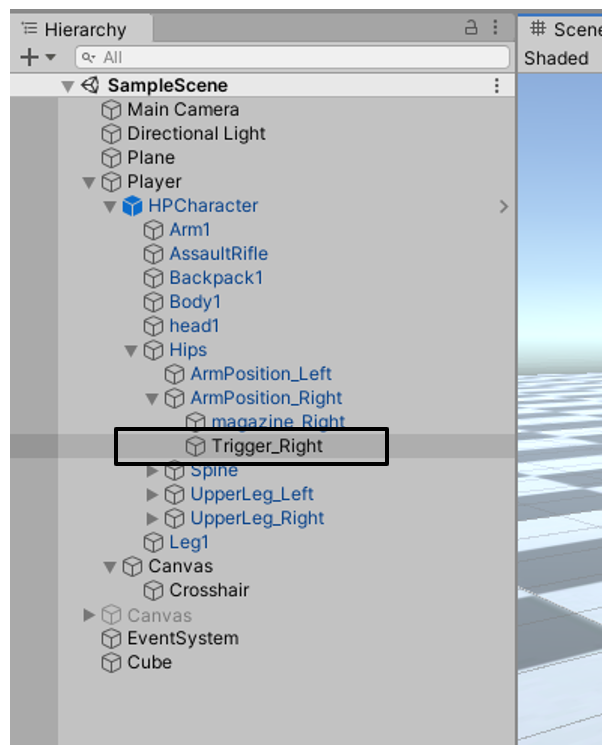
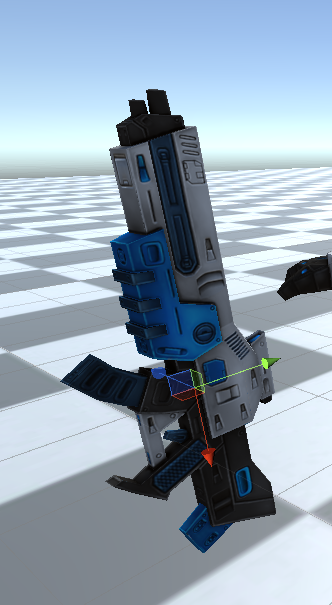
To implement aiming, you will require three pieces of information.

* The first is the gunpoint (the point from where the bullet comes out),
* The second is the forward direction of the gun, and
* The third is the layer mask for all objects that the weapon can hit.

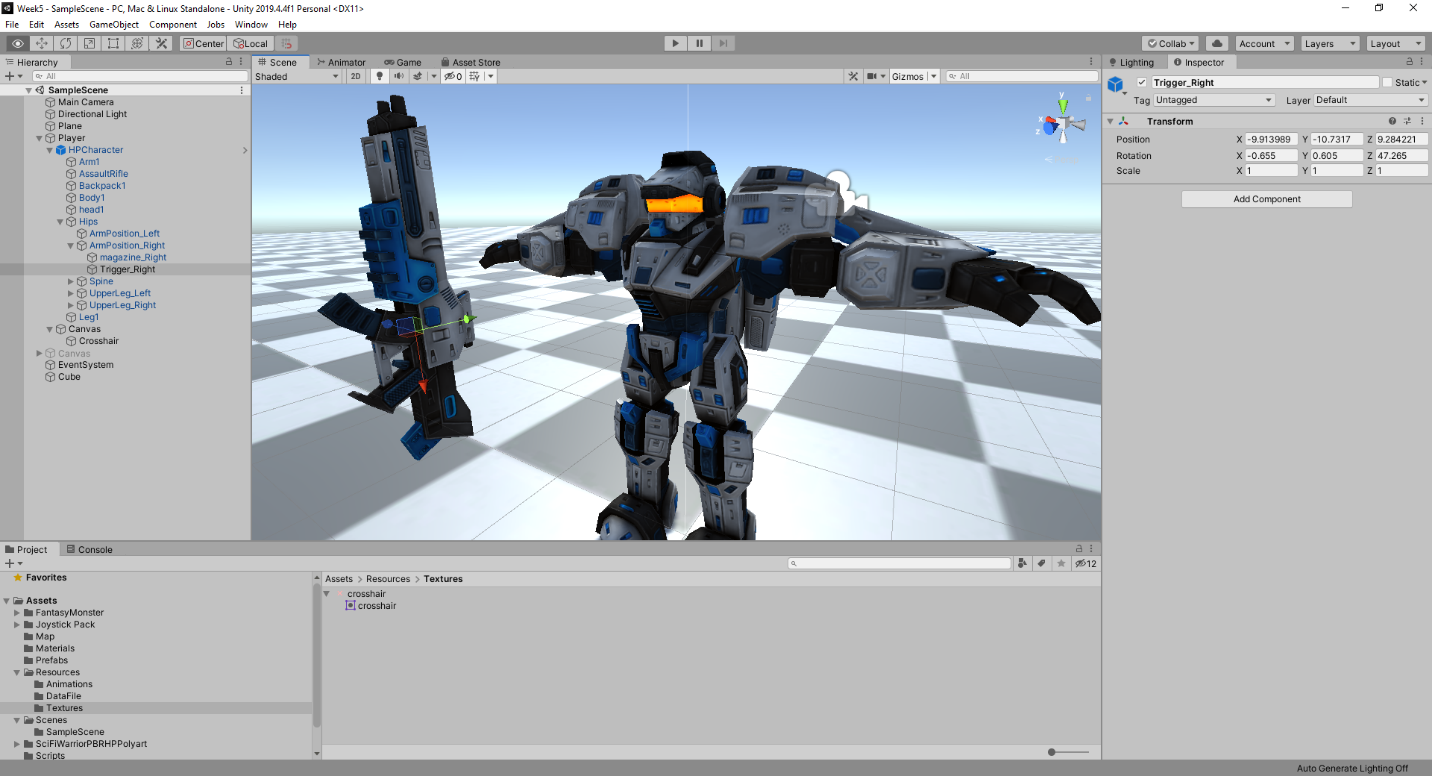
Once we have these three pieces of information, we can create a ray from the gunpoint towards the forward direction and do a **Physics.Raycast** to find if there is an intersection.

### Configure the Gunpoint

Select the Player game object from the scene hierarchy and find the **Trigger\_Right** transform.

Trigger\_Right is the transform node that represents the gun. See the picture below.



Selecting the **Trigger\_Right** node from the Player’s hierarchy shows the three axes of the gun’s transform. Red->x-axis (right), green->y-axis (up) and blue->z-axis (forward). You can see that the gun’s muzzle is pointing towards the negative x-axis. Hence, our direction for the Raycast will be the negative x-axis (or right) of this transform.

To find the exact gunpoint will be slightly cumbersome for us, as we did not create the 3D model of this character.

Now drag and drop **Trigger\_Right** to **M Gun Transform** field in the Inspector.

**Discuss how you can find the approximate location of the muzzle point or gunpoint.**

It turns out (after some trial and error) that the gunpoint is approximately

mGunTransform.transform.position + dir \* 1.2f - mGunTransform.forward \* 0.1f;

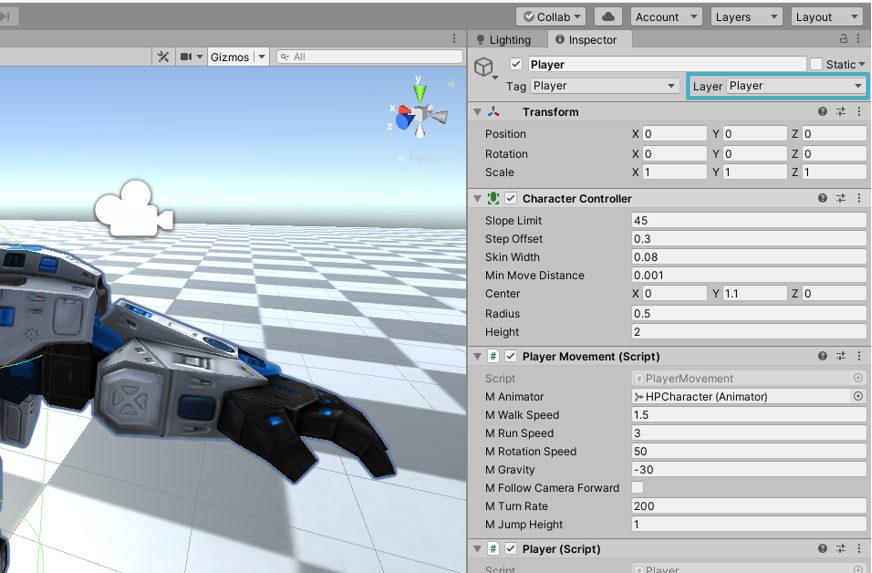
We will use the above value as the gunpoint. We now have the gunpoint and the direction of the firing. As for the layer mask, we want to intersect any object but the Player itself. We do not wish an intersection test to be done against self (which in this case is the Player). So our layer mask will be a bitwise complement of Player’s layer mask.

### Create a Player Mask

Select the **Player** game object in the scene hierarchy. Click on **Layer** dropdown on the Inspector. Select on Add Layer and add a new **User Layer** called **Player**.



Now set this new Player layer to the **Player** as shown below.



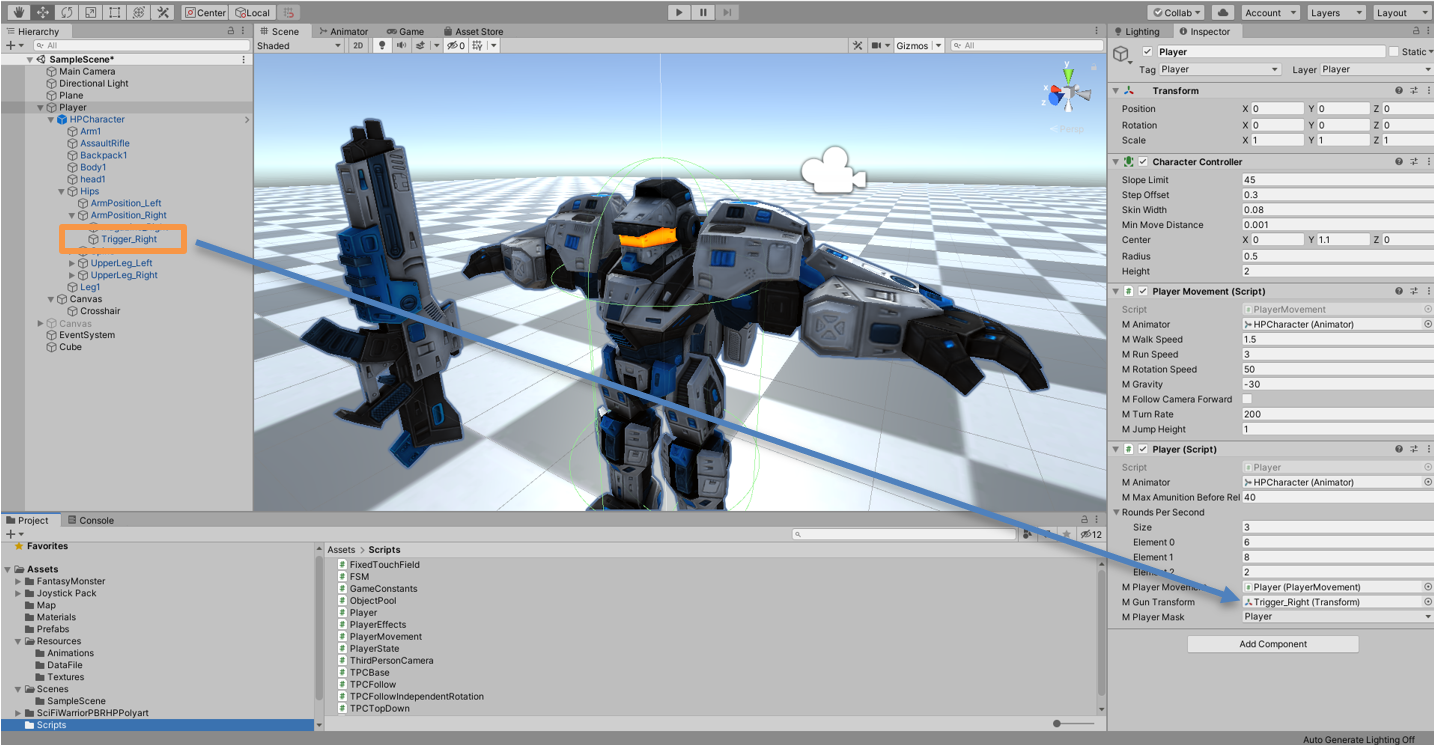
Now, we have all the necessary information to do the Raycast.

Add two public variables in **Player.cs**.

public Transform mGunTransform;

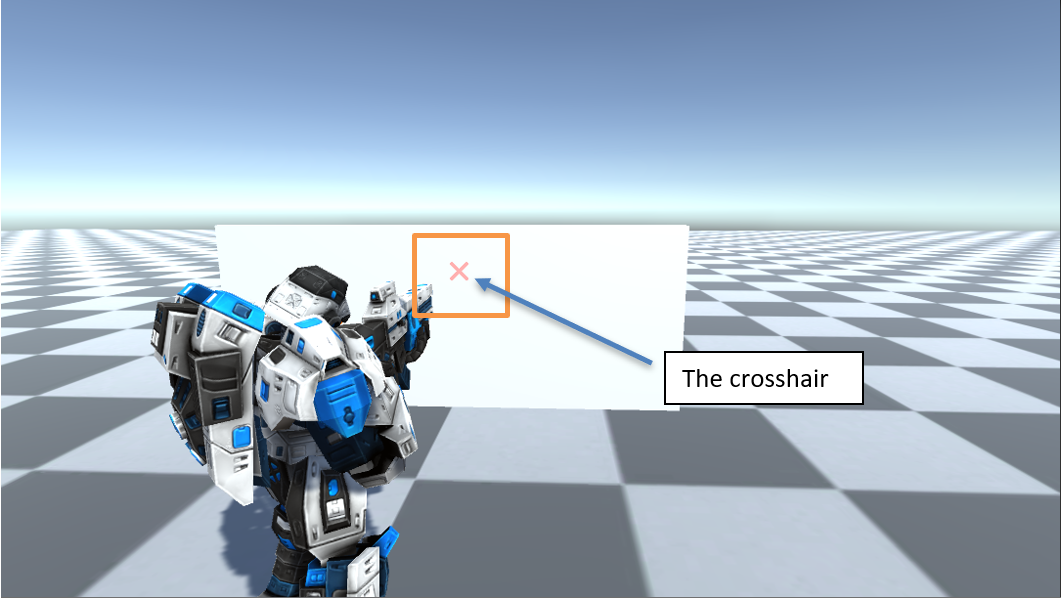
public LayerMask mPlayerMask;

Go to Unity editor and drag and drop the **Trigger\_Right** transform **M Gun Transform** field.



Select the **Player** mask for **M Player Mask** dropdown field.

### Create the Crosshair



The crosshair

The crosshair is a visual indication of the bullet hit point. For our implementation, the crosshair will be the hit point of a **Physics.Raycast** from the gunpoint to the object towards the gun’s forward direction in world coordinates.

**Discuss how we can implement this in the class with your tutor.**

There are a few requirements that we want to meet for our crosshair. The first requirement is to show the crosshair whenever a gun points towards an object, no matter the shape of that object. The second requirement is that it should not be behind the object and should not have a z-fighting situation.

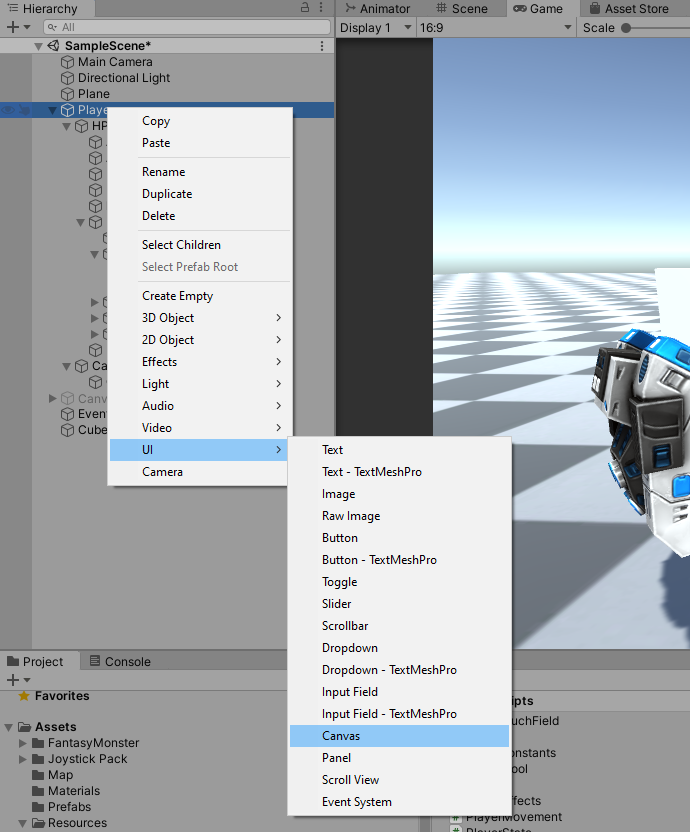
These requirements lead us to implement the crosshair as a UI element, in 2D. A UI element will always be rendered last and will not suffer from z-fighting with other objects in the scene. The idea is to create a 2D image of a crosshair on a canvas and then hide it. Then at runtime, the script will do the **Physics.Raycast** and only if there is an intersection, transform the crosshair from its original 2D location (with reference to the canvas) to the intersected 3D location projected to screen space and show it.

Don’t worry if it sounds too tricky for now. We will do the implementation together, and after we do the implementation, it will become more apparent.

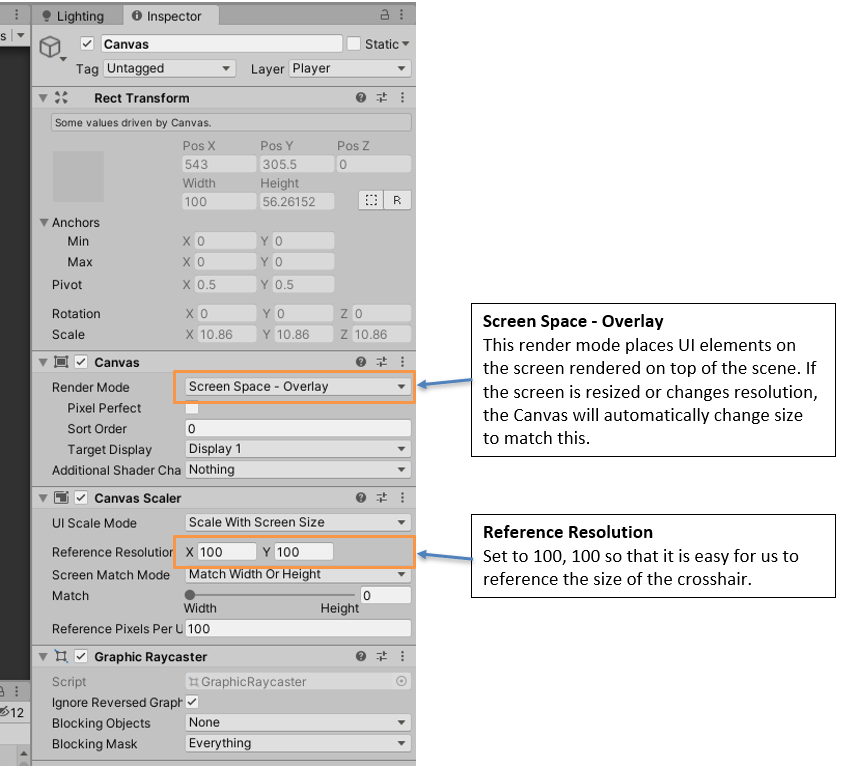
#### Create the Canvas

Navigate to **Assets->Resources->Textures**. You will find the image that we will use for our crosshair. You can also create your own image using Photoshop or other tools.

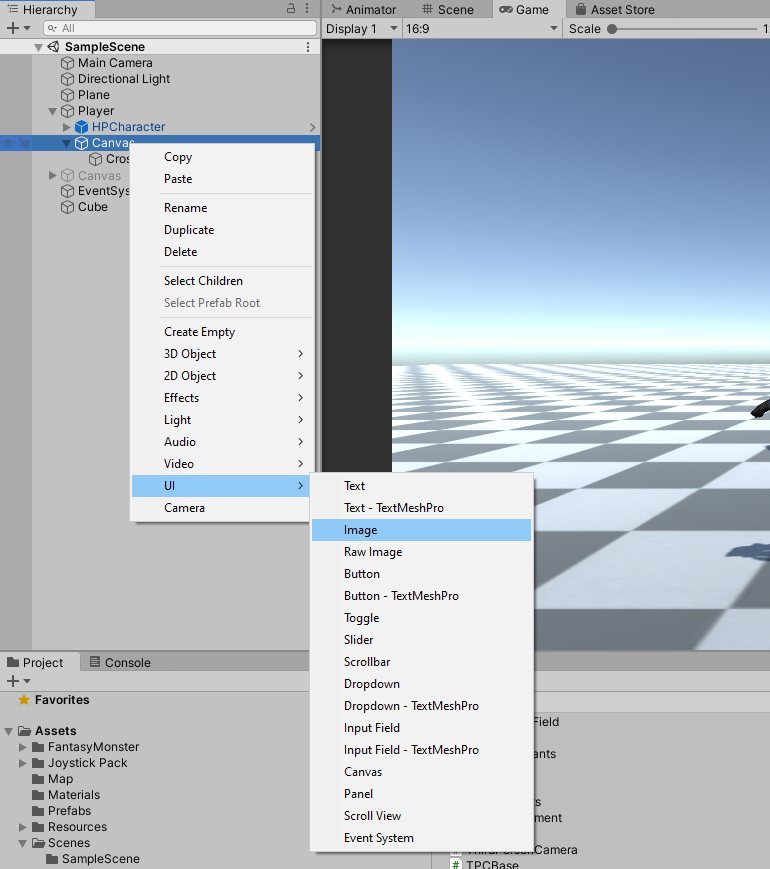
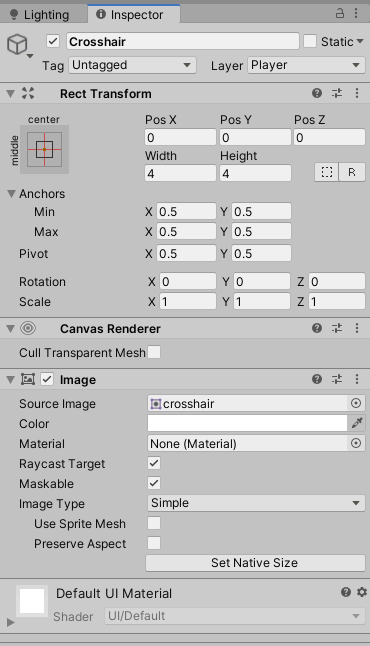
Select the **Player**, right-click and add a new **Canvas**.



Select the Canvas, go to Inspector and set the values as shown in the picture below.



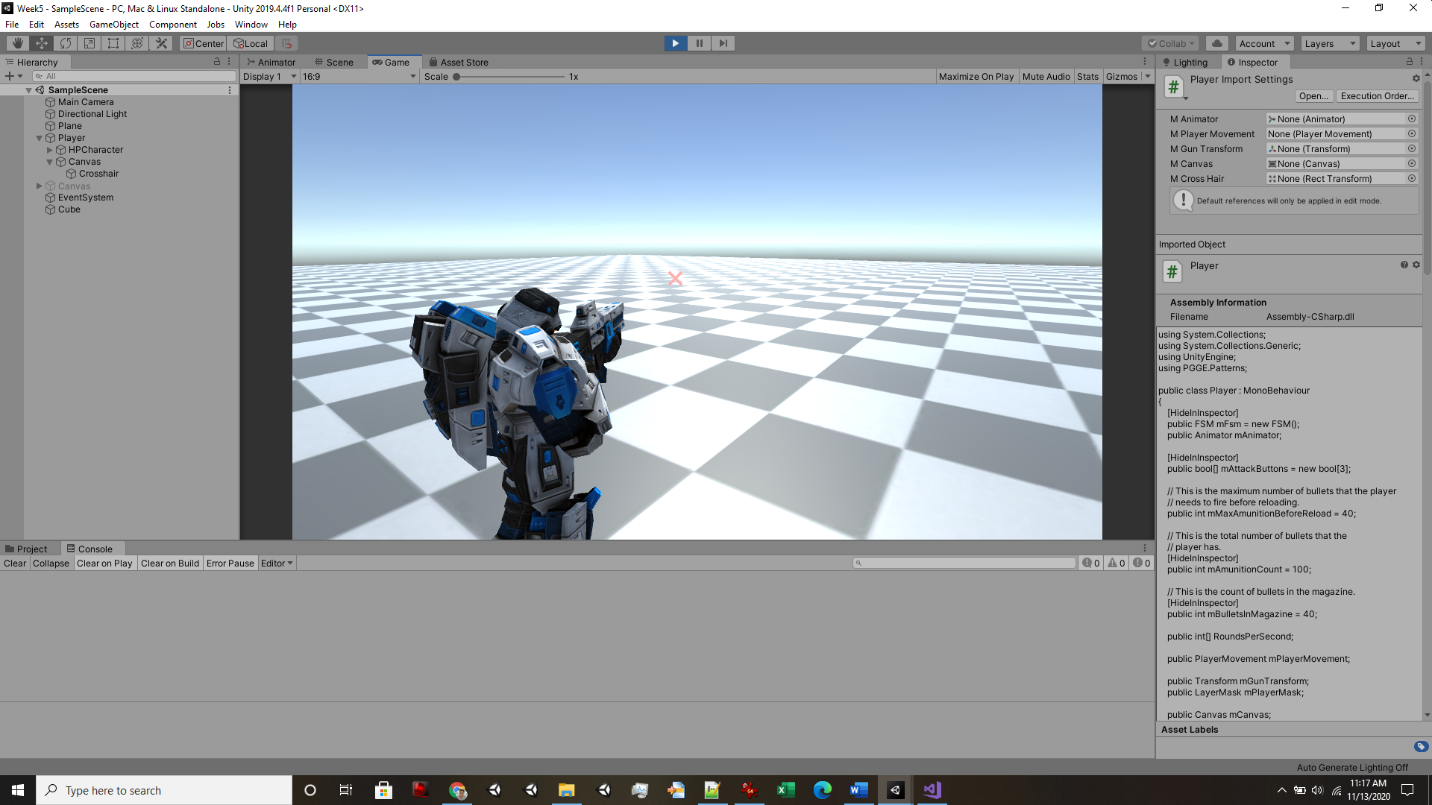
#### Create the Crosshair Image

Select the canvas, right-click, go to **UI** and select **Image**.

Rename the **Image** to **Crosshair**. Go to Inspector and

* Reset Rect Transform.
* Set the width and height to 4 and 4, respectively.
* Select the Source Image to be the crosshair image that is in the Assets->Resources->Texture folder.

Click **Play** and see whether the crosshair appears on the screen.



You should be able to see the crosshair on the screen when you play the game, as shown above. The crosshair always shows and it at the centre of the screen. So far, this behaviour is just like any other UI element.

We will now implement the necessary script to hide this crosshair and show only when the gun’s line of fire intersects an object. If there is an intersection, then we will also need to transform this crosshair to the position of intersection point projected to the screen space.

### Programming Task 3 – Implement the Aim function

We have completed all the necessary setup for aiming. We are now only left with implementing the code to do the actual aiming using **Physics.Raycast** and find out what objects get hit by our gun.

Create a new function called **Aim** in **Player.cs**.

public void Aim()

{

// For Student ----------------------------------------------------//

// Implement the logic of aiming and showing the crosshair

// if there is an intersection.

//

// Hints:

// Find the direction of fire.

// Find gunpoint as mentioned in the worksheet.

// Find the layer mask for objects that you want to intersect with.

//

// Do the Raycast

// if (intersected)

// {

// Draw a line as debug to show the aim of

// fire in the scene view.

//

// Find the transformed intersected point to screenspace

// and then transform the crosshair position to this

// new position.

//

// Enable or set active the crosshair game object.

// }

// else

// {

// // Hide or set inactive the crosshair game object.

// }

//-----------------------------------------------------------------//

}

I have given you hints as shown above.

**Discuss with your tutor and implement the necessary.**

Amend the **Player.Update()** method and add **Aim()** below **mFsm.Update()** as follows.

void Update()

{

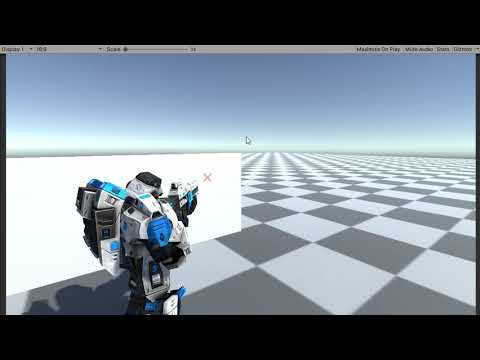
mFsm.Update();

Aim();

\*\*\*\*

\*\*\*\*

Click **Play** and observe the behaviour. Below is

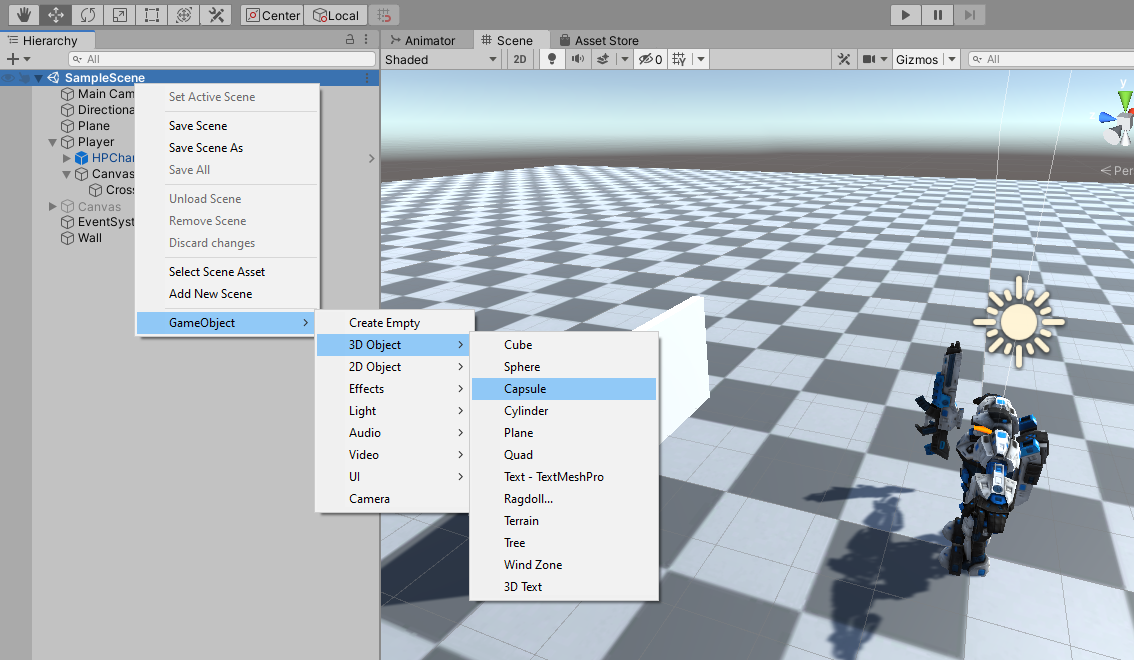
[](https://www.youtube.com/embed/VEdr2Qg29_I?feature=oembed)

## Implementing Shooting Bullets

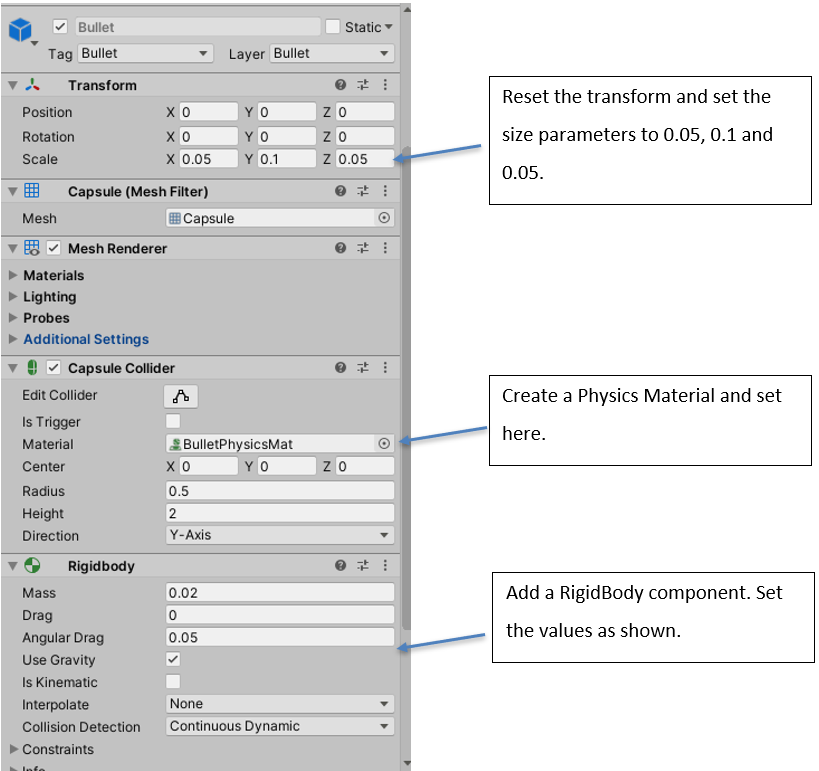
In this section, we will create the actual shooting of bullets. In the previous sections, we have made the structure of attack and aiming of the gun. Here will implement the real bullet game object and shooting of bullets.

### 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();

}

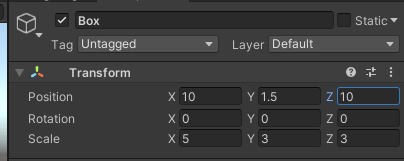
}

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.

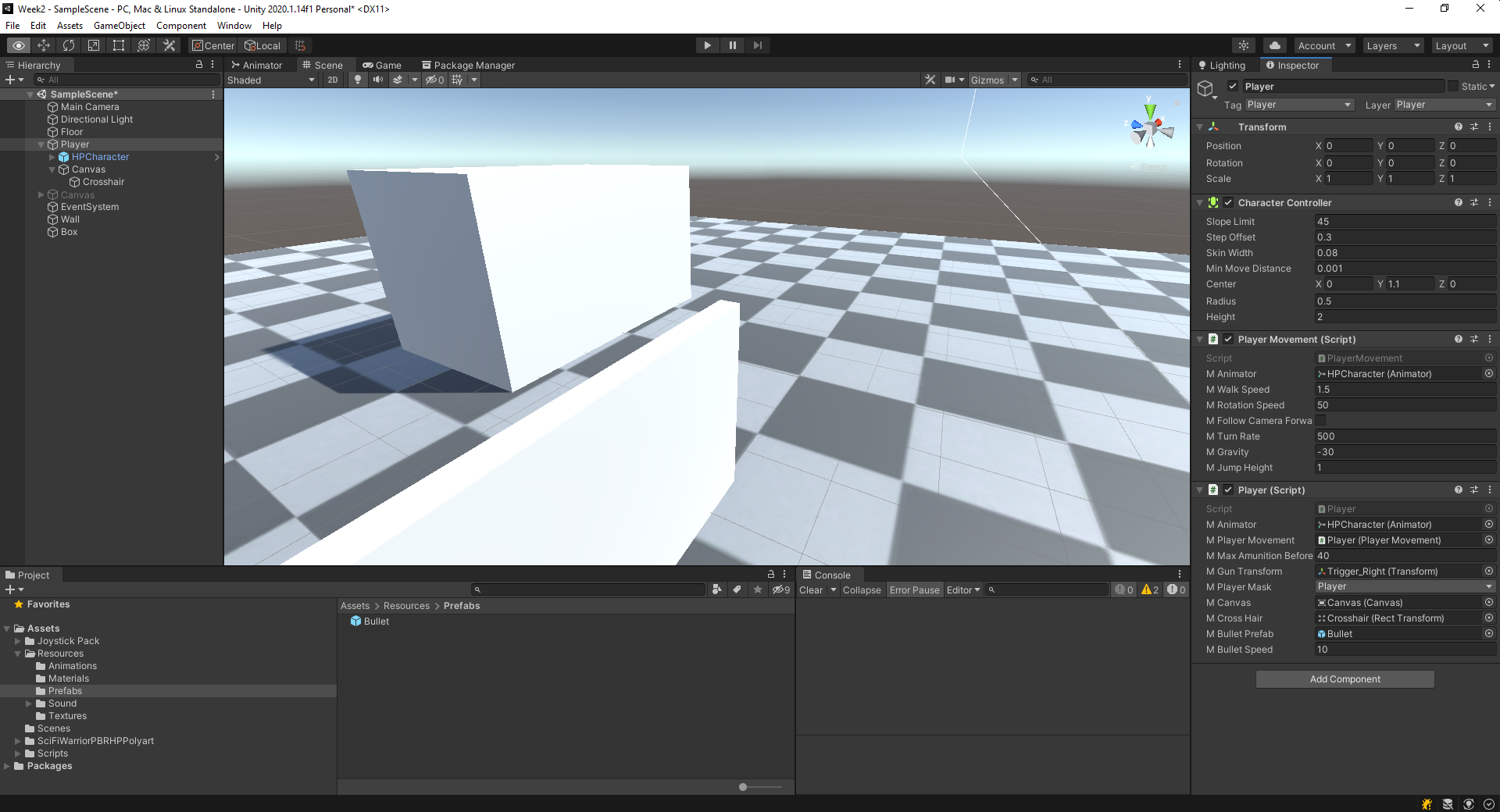
We shall now implement the shooting of the bullets. Double click and open the Player script.

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.

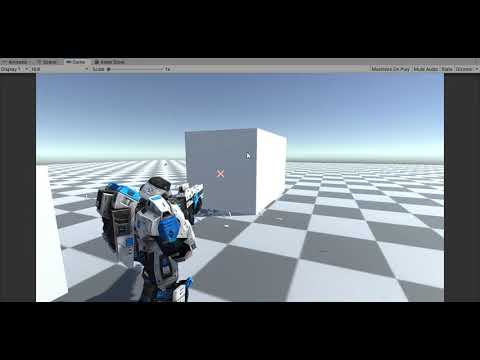
public void Fire(int id)

{

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)

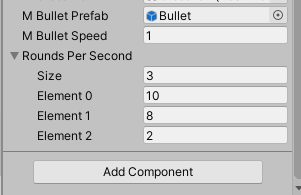
{

StartCoroutine(Coroutine\_Firing(id));

}

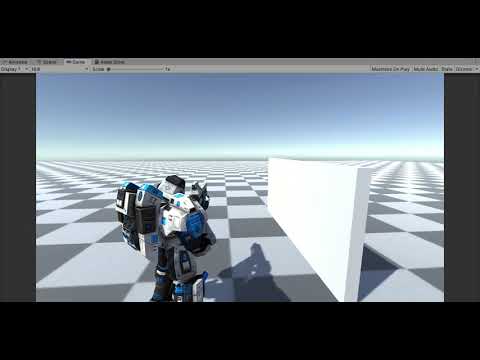
}

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.