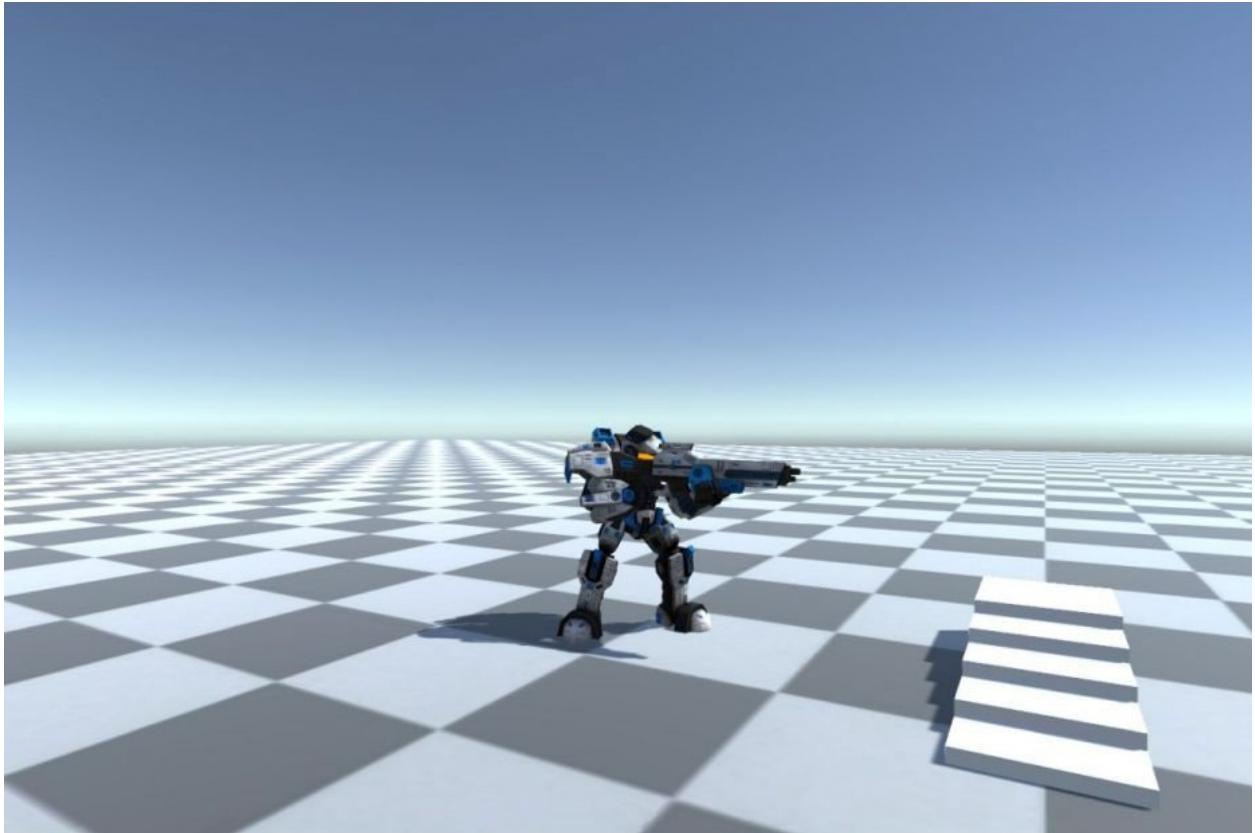


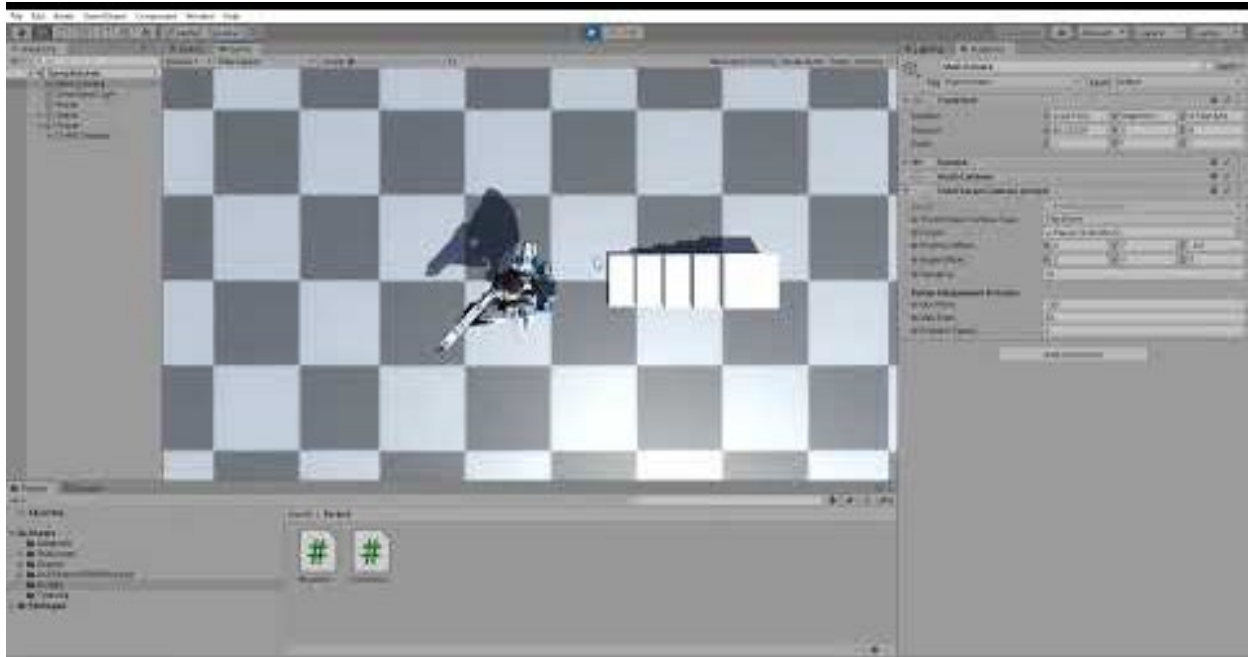
Third-Person Camera Control in Unity using C#



Shamim Akhtar

Introduction

This worksheet will apply the concept of inheritance and polymorphism to implement a configurable third-person camera control in Unity for an animated character. Look at the video below on what we will achieve by the end of our worksheet.

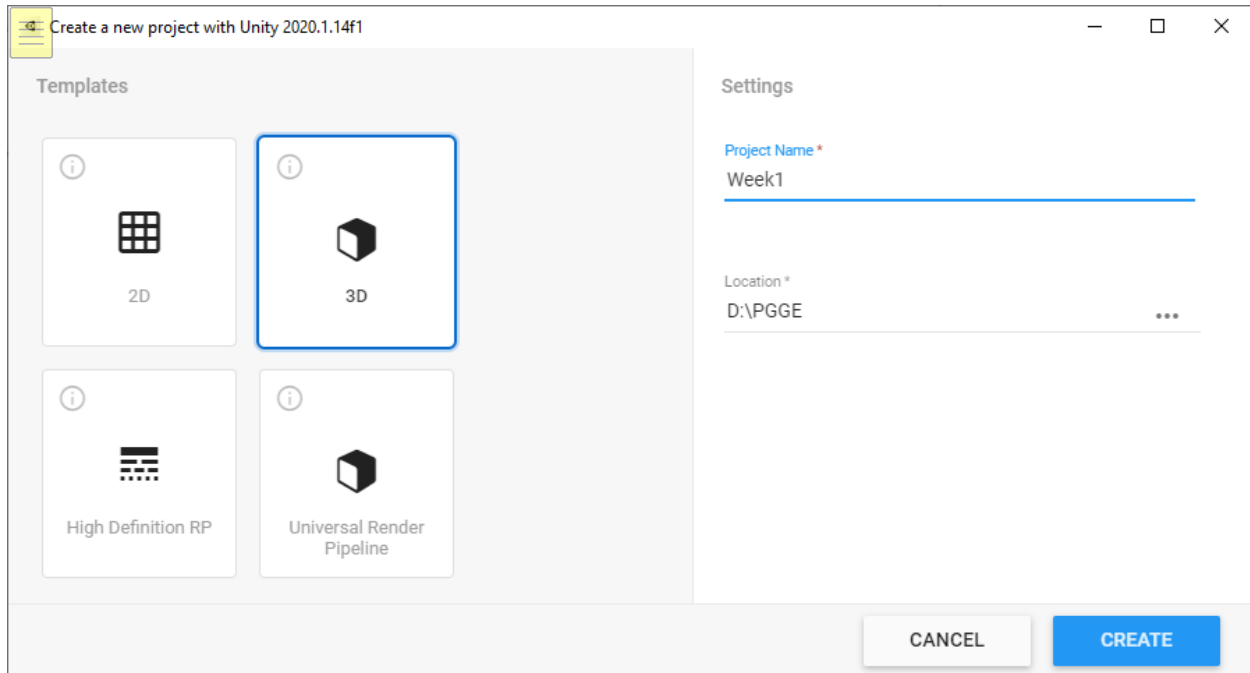


There are three sections in this worksheet.

- Section 1 will concentrate on setting up the scene and the animated character.
- Section 2 will implement the necessary third-person camera controls for PC, Linux, and Mac build settings.
- Section 3 will port the third-person camera controls for touch screen devices with a virtual joystick and deploy them to an Android phone.

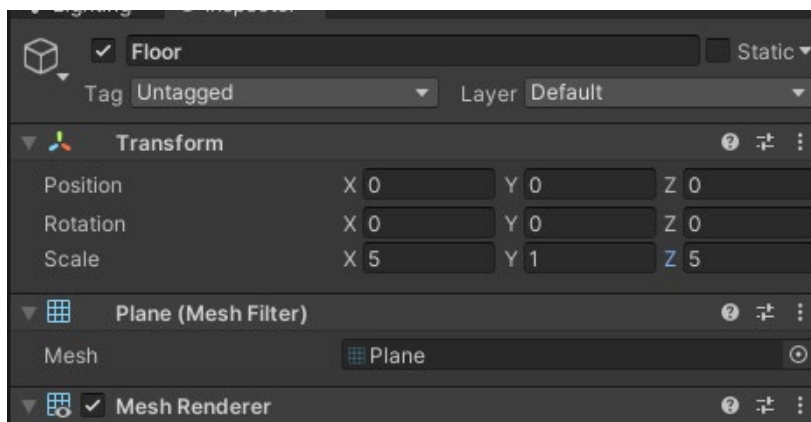
Section 1 – Scene and Character Setup

Create a Unity 3D project

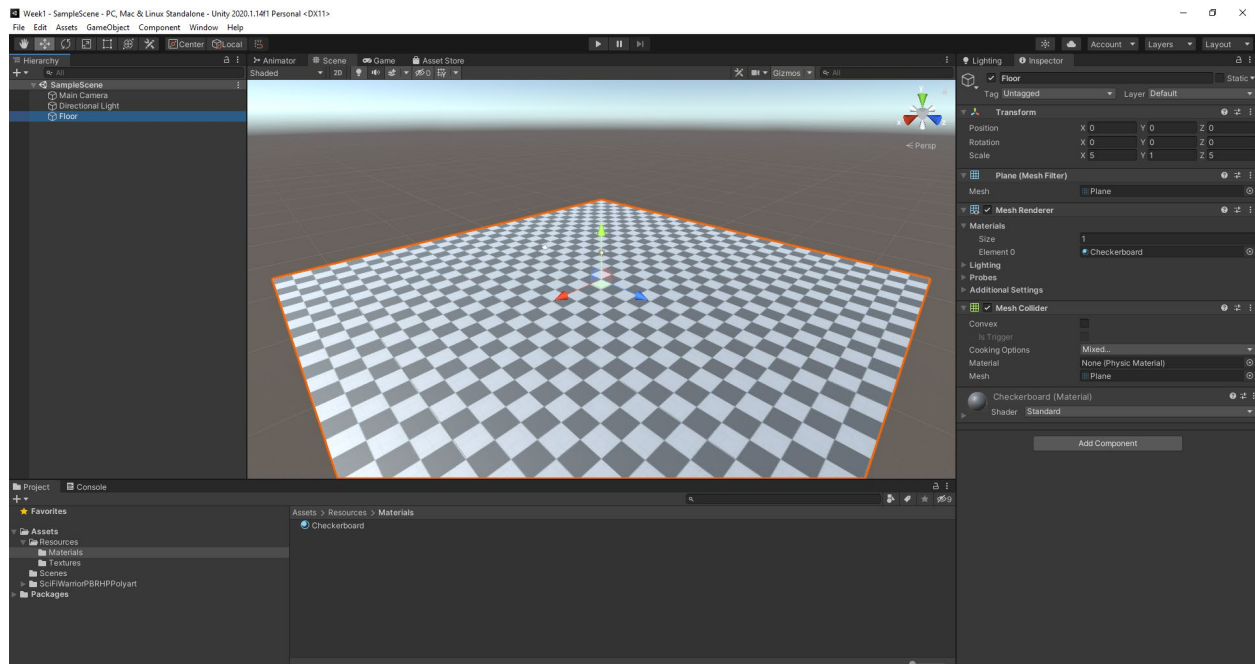
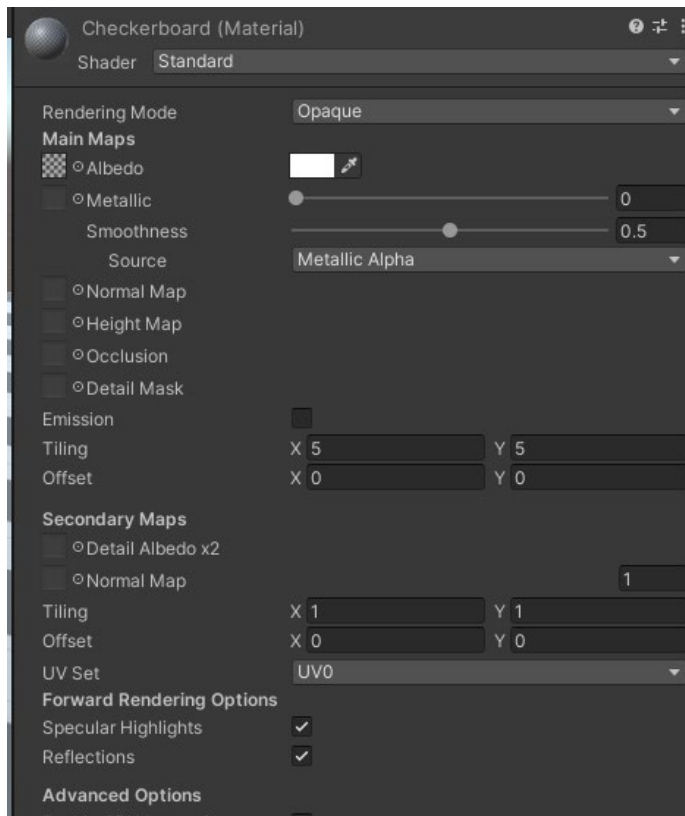


Create a Scene

1. Create a floor by dragging a plane onto the scene and name it as **Floor**. Resize it to 5, 1, 5.



2. Create a folder named **Resources** in **Assets**. Create another folder in the **Resources** and call it **Textures**.
3. Download the **Checkerboard.jpg** from your LMS and put it in the **Textures** folder. Drag and drop this image as a texture to the **Floor**. Change the tiling factor to 5, 5.



Download the following free Assets from the Unity Asset Store

[Sci Fi Warrior PBR HP Polyart](#)

The screenshot shows the Unity Package Manager interface. On the left, a list of packages includes 'Sci-Fi Warrior' (version 1.0) and 'Sci Fi Warrior PBR HP Polyart' (version 1.2). The right pane displays the details for the selected package. It includes the publisher 'Dungeon Mason', the version '1.2.0 - January 30, 2020', and links to the asset store, publisher website, and support. The features section lists that it includes a full character pack with modular parts and animations, and is optimized for mobile games (low poly) with a 2K texture atlas. Below this are three images of the character model. Further down, it shows the package size (23.82 MB), supported Unity versions (2018.4.15 or higher), purchase date (April 30, 2020), and release details (1.2 released on January 30, 2020; Original released on January 05, 2018). At the bottom, there are 'Import' and 'Download' buttons.

Package Manager

⊕ Packages: My Assets Sort: Name ↓ Filters Clear Filters

Sci-Fi Warrior 1.0

Sci Fi Warrior PBR HP Polyart 1.2

Sci Fi Warrior PBR HP Polyart

[Dungeon Mason](#)

Version 1.2.0 - January 30, 2020 [asset store](#)

[View in the Asset Store](#) · [Publisher Website](#) · [Publisher Support](#)

FEATURES

- If you need the full character pack with modular parts and animations, you can buy it from here.
- Optimized for mobile games(low poly) and one 2K texture atlas for all parts(For Polyart, 512x512)

[More...](#)

Images & Videos

[View images & videos on Asset Store](#)

Package Size
Size: 23.82 MB (Number of files: 39)

Supported Unity Versions
2018.4.15 or higher

Purchased Date
April 30, 2020

Release Details
1.2 (Current) - released on January 30, 2020 [More...](#)
Original - released on January 05, 2018

All 2 packages shown

Last update Dec 8, 15:43

[Import](#) [Download](#)

The screenshot shows the Unity Asset Store page for the 'Sci Fi Warrior PBR HP Polyart' package. The top banner features the 'Fantastic Fantasy Mega Bundle' promotion. The navigation bar includes links to Assets, Tools, Services, By Unity, and Industries, along with a search bar and sign-in options. Below the navigation bar, statistics show over 11,000 star assets, a rating of 85,000+ customers, and support by over 100,000 forum members. The main content area displays a large image of the character model. To the right, the package details are listed: 'Sci Fi Warrior PBR HP Polyart' by 'Dungeon Mason', rated 5 stars with 20 reviews, and marked as 'FREE'. There are buttons for 'Add to My Assets', 'Add to List', and 'Share'. A table of specifications includes the license (Extension Asset), file size (23.8 MB), latest version (1.2), latest release date (Jan 30, 2020), supported Unity versions (2018.4.15 or higher), and a link to support. At the bottom, there is a 'You might also like' section with a 'See more' link.

Fantastic Fantasy Mega Bundle. Tools and art to build your next epic fantasy adventure.

unity Asset Store

Search for assets

Sign In

Assets Tools Services By Unity Industries

Sell Assets Feedback FAQ

Over 11,000 5 star assets

Rated By: 85,000+ customers

Supported by over 100,000 forum members

Home > 3D > Characters > Robots > Sci Fi Warrior PBR HP Polyart

Sci Fi Warrior PBR HP Polyart

[Dungeon Mason](#) ★★★★★ 5 | 20 Reviews

FREE

[Add to My Assets](#)

[Add to List](#) [Share](#)

License [Extension Asset](#)

File size 23.8 MB

Latest version 1.2

Latest release date Jan 30, 2020

Support Unity versions 2018.4.15 or higher

Support [Visit site](#)

You might also like [See more](#)

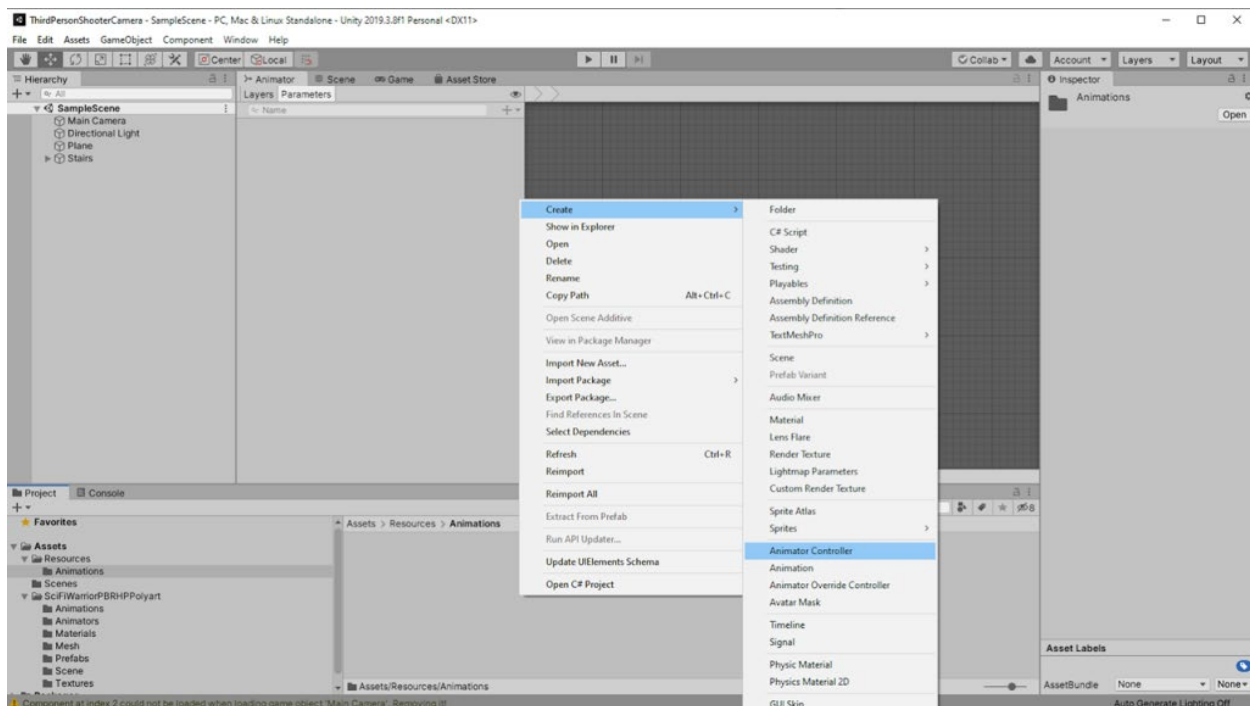
Import these assets. We will use this as our Player and centre our third-person camera control around this Player.

The imported assets comprise three prefabs and one animation controller. We will use one of these prefabs and create a new animation controller that will suit our needs.

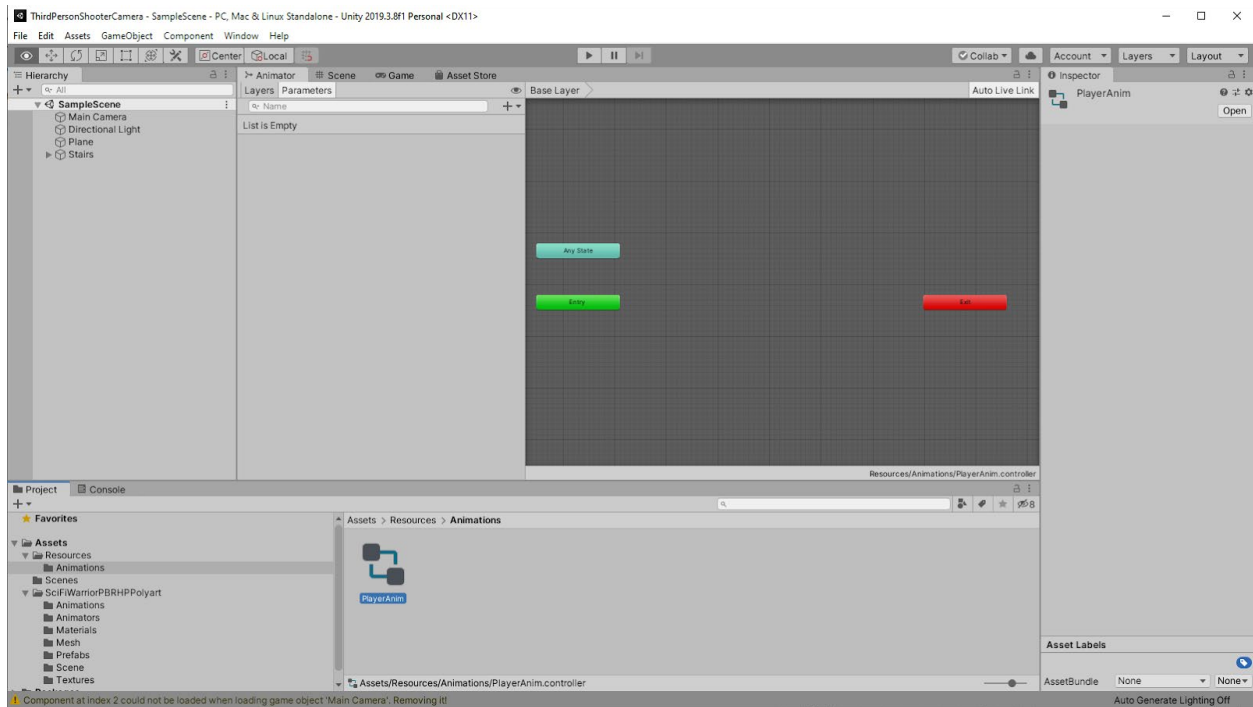
Create the Character's Animation Controller

Create a new folder called **Animations** inside the **Resources** folder.

Right-click on the **Project** window → **Create** → **Animation Controller**.

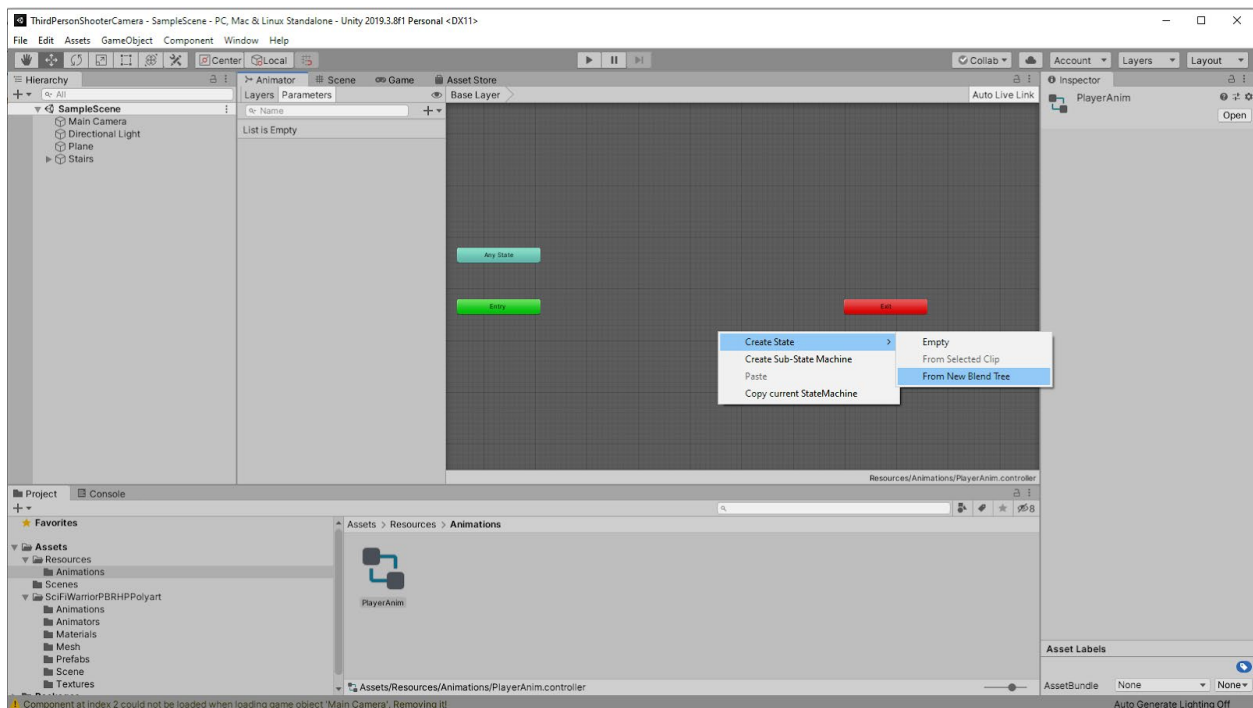


Name it **PlayerAnim**.

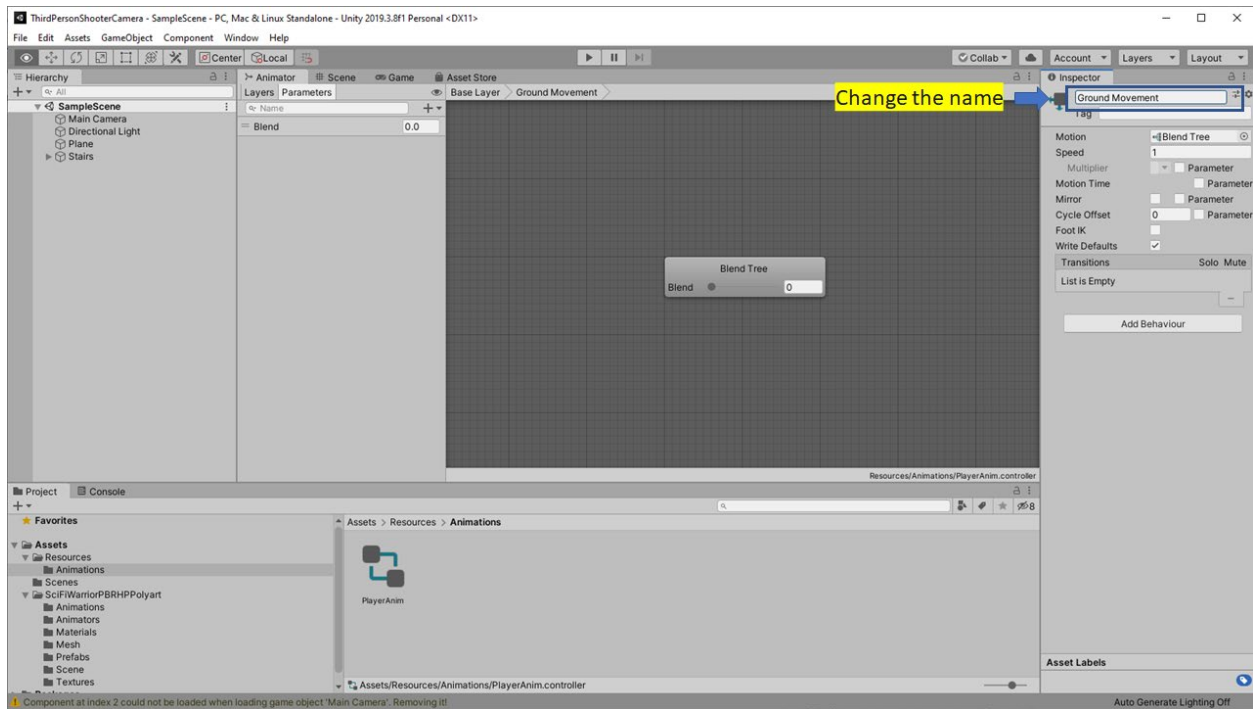


We will use **Blend Tree** to create movement control for the Player.

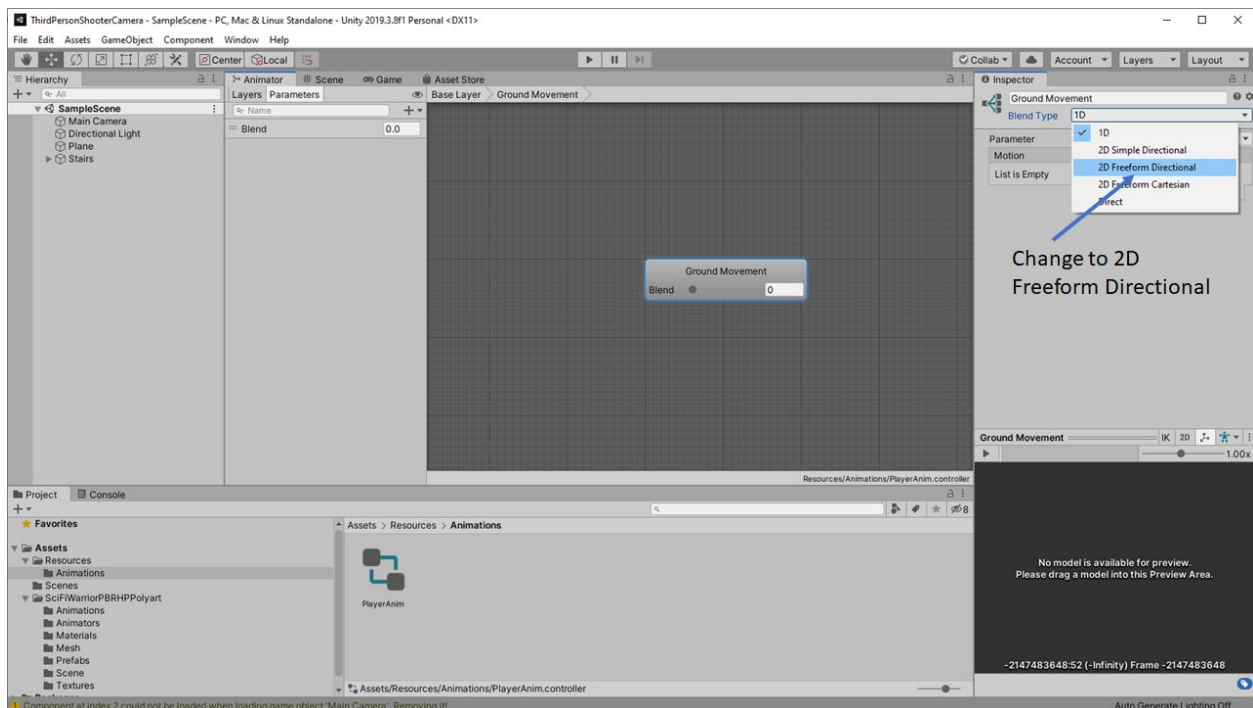
Right-click anywhere in the Base Layer window of the **PlayerAnim** animation controller and click on **Create State → From New Blend Tree**



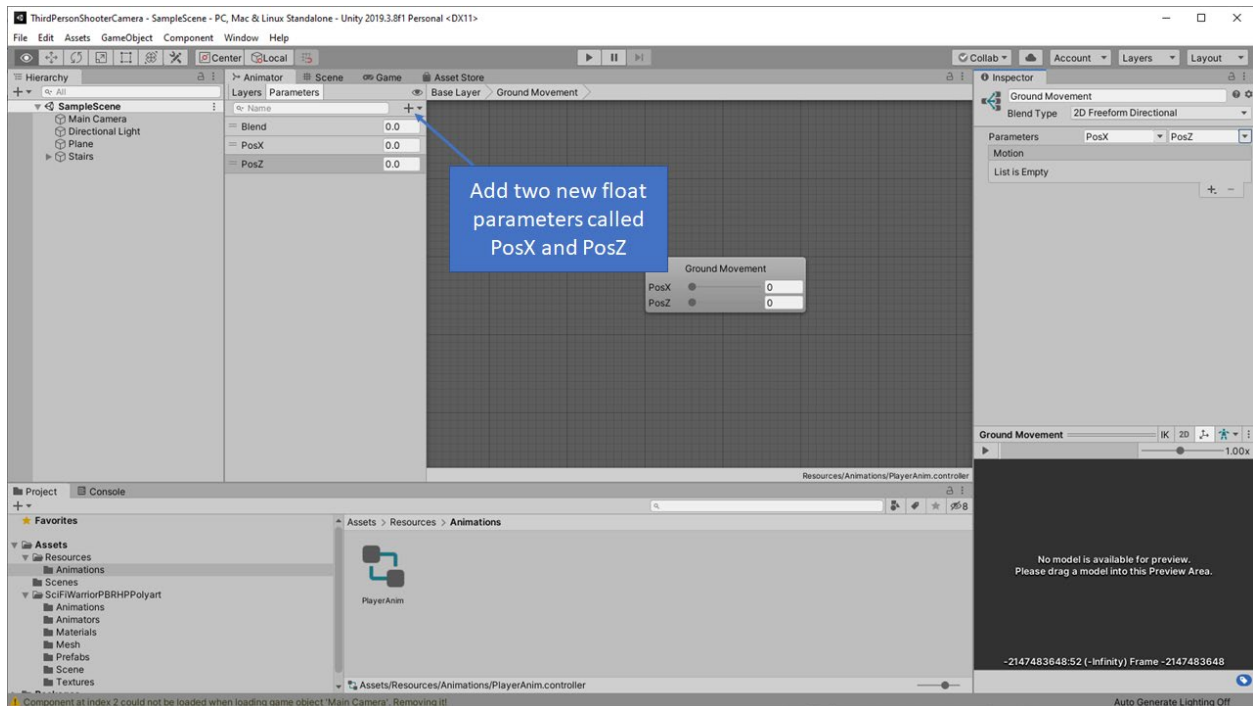
Double click and select the newly created Blend Tree.



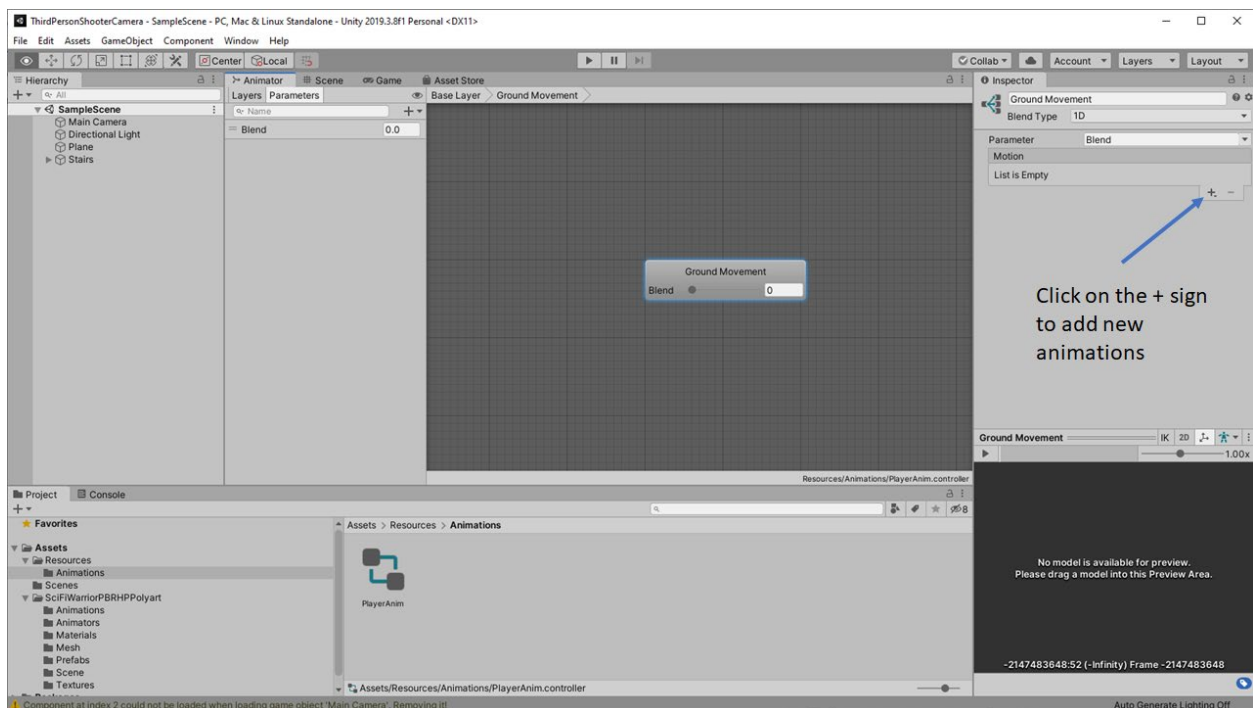
Rename it to **Ground Movement** or any other name that you fancy. Select Ground Movement and change the **Blend Type** to **2D Freeform Directional**.



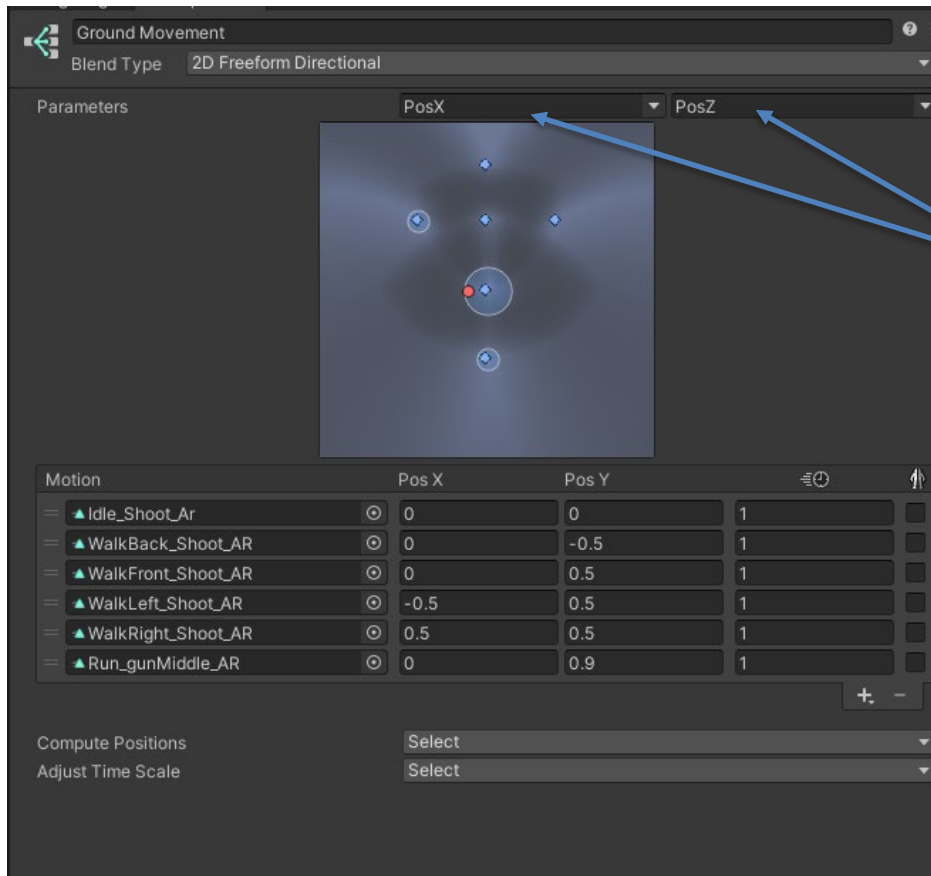
Now add two new float parameters called **PosX** and **PosZ**. These are the parameters that we will use to manipulate the animations of the Player based on inputs.



We will now start adding animations to the **Ground Movement** Blend Tree. We will do this by clicking on the + sign in **Motions** field.



Click on the + and select Add Motion Field. Add the animations, as shown below.

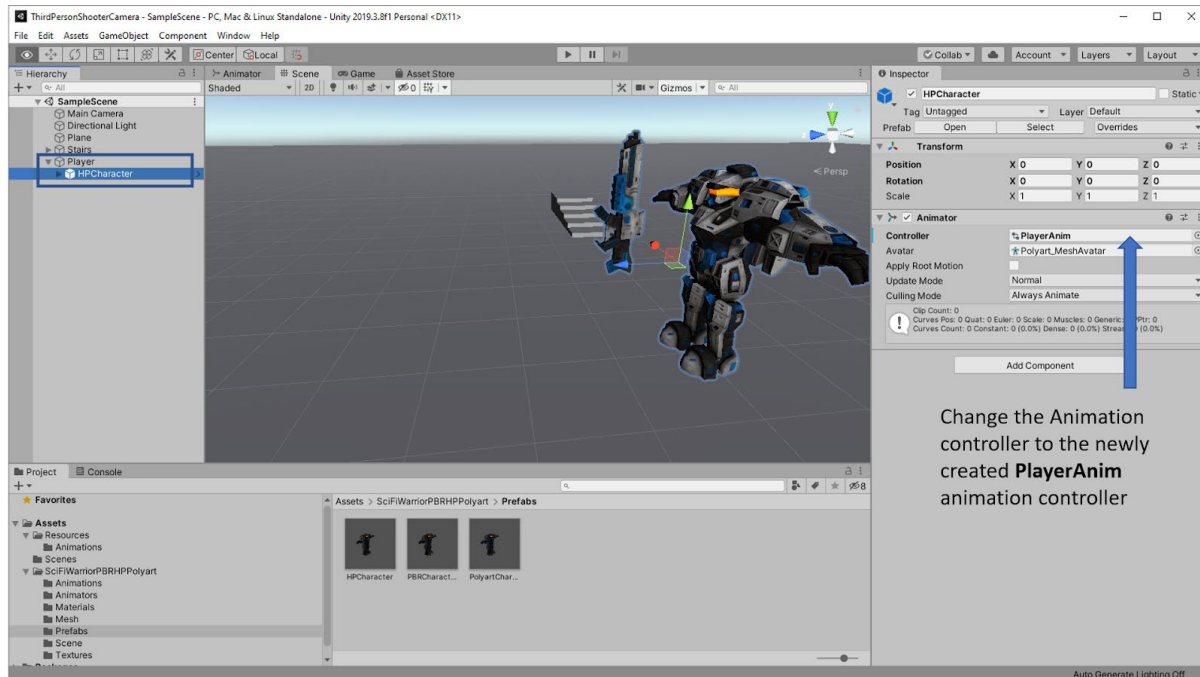


Select the parameters field to be PosX and PosZ.

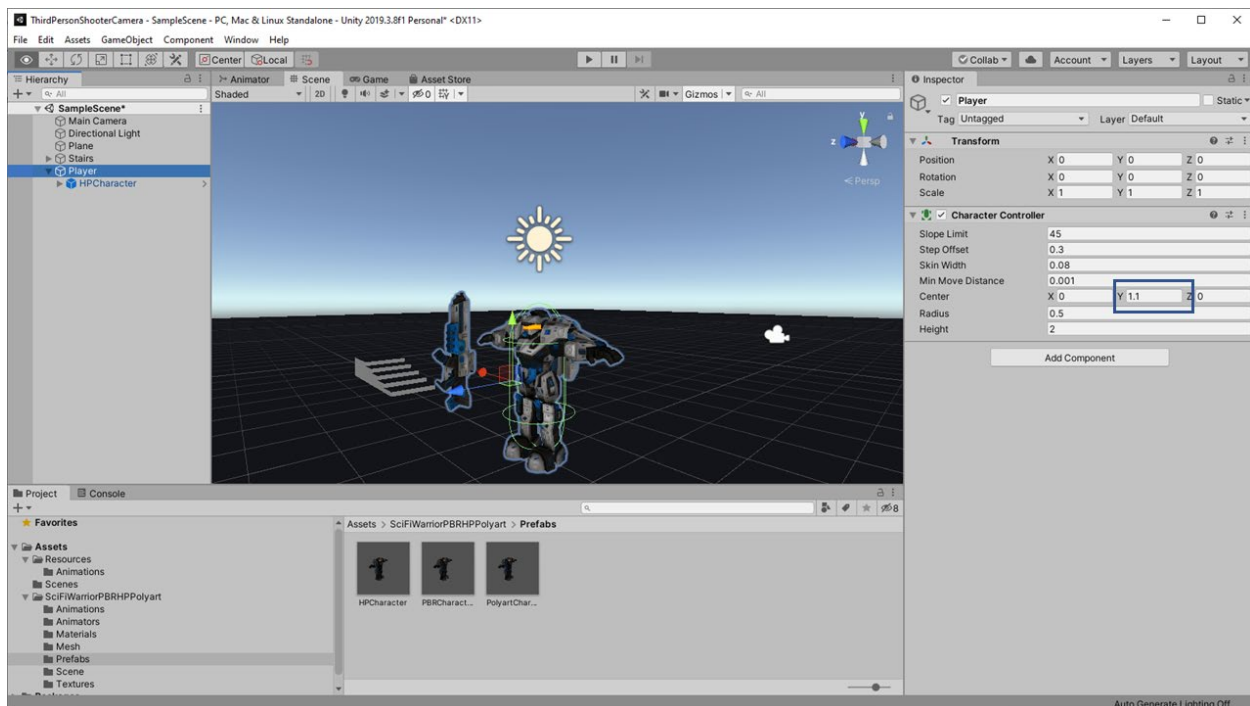
The list of animations with the appropriate values

Create the Player

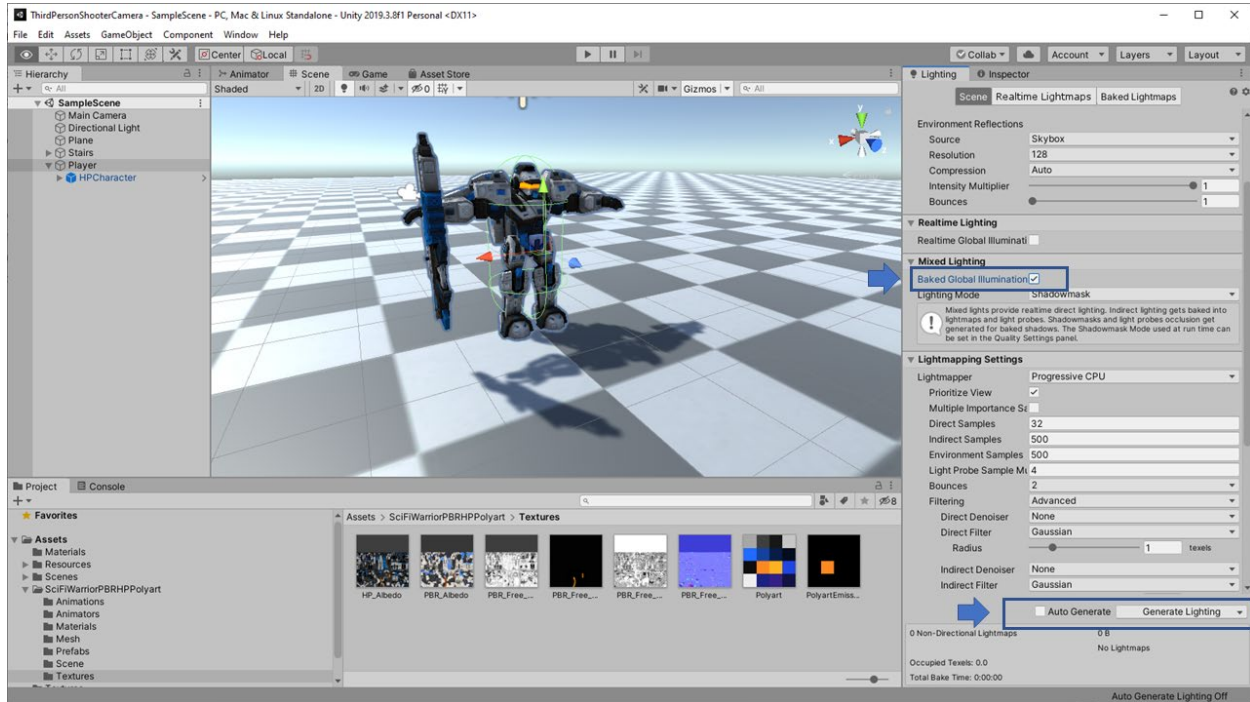
We will now create the actual Player. For this, we create an empty game object and rename it to **Player**. Drag and drop the **HPCharacter** prefab to the **Player** game object. Change the Animation Controller to the newly created **PlayerAnim**.



Add the **Character Controller** component to **Player**. Change the Center Y value to 1.1.



Go to **Window** → **Rendering** → **Lighting Settings** and open the **Lighting** window. Now check the **Baked Global Illumination** and click **Generate Lighting**.



Create a Player movement script

We will now create a movement controller for the character. This controller will allow us to control the character's movement by using keyboard and mouse inputs for PCs and a virtual joystick for touch screen devices. For now, we will only concentrate on build for PC, Linux and Mac. We will handle touch screen devices in the later part of the tutorial.

Create a folder in **Assets** and call it **Scripts**. Create a new script file called **PlayerMovement.cs** and attach the script to the **Player** game object. Move the file **PlayerMovement.cs** from the **Assets** folder to the **Scripts** folder. Remember that Player is an empty game object that holds the actual character with the **PlayerAnim** Animator.

Programming Assignment 1 - Implement Player Movement

In this programming assignment, you will implement the necessary scripts in the C# script file **PlayerMovement.cs** to allow the Player to make a basic movement. These movements will include **Walking and Running**. For walking, you should use the **W, A, S, D** keys, and when combined with the **left Shift** key, the same keypresses should make the Player run.

Once implemented, click **Play** and test the application.

We will come back to **PlayerMovement** later again to enhance its functionality. We move on to implement the third-person camera control now, one step at a time. By the time we finish, we hope to create a robust third-person camera control system.

Section 2 – Implement Third-person Camera Controls

In this section, we will implement four different types of third-person camera controls. We will use inheritance and polymorphism that we learnt in class to build our classes for the third-person camera control.

The four types of third-person camera controls are:

1. Track

In this type of third-person camera control, we simply track the object. The camera does not move but only looks at the Player.

2. Follow

In this type of third-person camera control, we make the camera follow the Player. The distance of the camera from the Player is configurable. The camera does not track the Player's rotation but moves with the Player.

3. Follow and Track Rotation

In this type of third-person camera control, we make the camera follow the Player and keep track of the payer's rotation. That means when the Player rotates, the camera also turns.

4. Follow with Independent Rotation

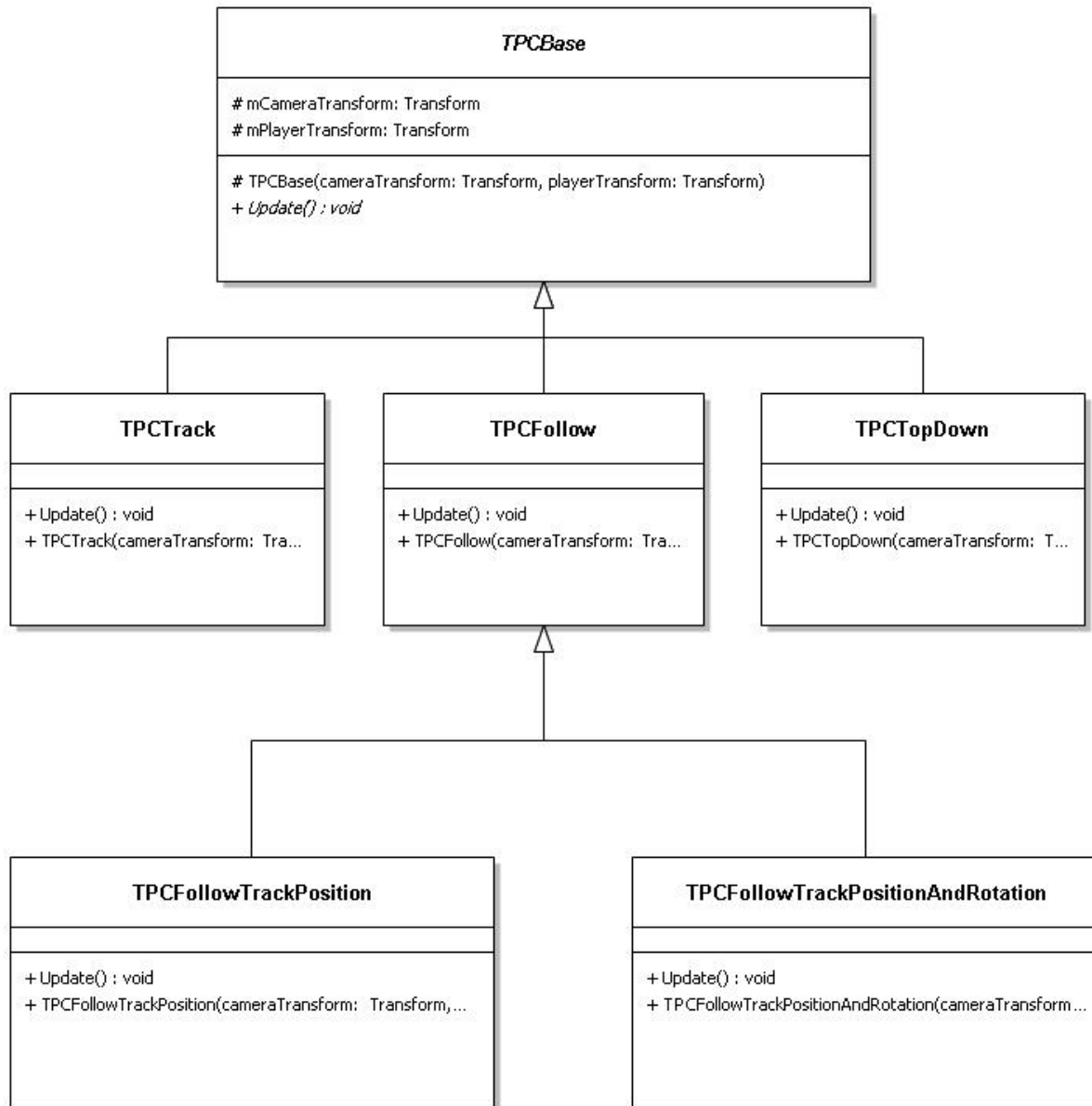
In this type of third-person camera control, we make the camera follow the Player, keep track of the payer's rotation and use independent yaw and pitch. The Player can then use the camera's look at direction to be the forward direction to move. This type of camera is beneficial for mobile and touch screen devices where the players forward direction is based on where the camera is looking. (This will be for the next workshop when we learn about mobile touch features)

5. Top-Down Camera

In this type of camera, we make the camera look from above to the Player's head. This type of camera is usually used in dungeon style games.

Discussion Question 1: How can we use inheritance and polymorphism to implement the different third-person camera controls? In this discussion, we do not discuss the actual mechanics of the various cameras but how to organize the classes from a software programming perspective.

The Class Hierarchy – Using Class Diagram



Third-Person Camera Base Class

Select the Main Camera from the Scene Hierarchy and add a new Script Component by creating a script file called **ThirdPersonCamera.cs**. We will add our codes to this file.

Create a new class and call it **TPCBase**. **TPCBase** is an abstract class that will provide the basis for the parent class. It will have an abstract method called **Update** that all third-person camera controllers must implement to achieve that specific third-person camera's functionality. Do note that this **Update** is not associated with Unity's Update method.

[Click here and do an in-class quiz now to confirm your understanding.](#)

Fields

We will have two member variables for this class. One will be the camera transform, and the other will be the player transform.

```
protected Transform mCameraTransform;  
protected Transform mPlayerTransform;
```

[Click here and do an in-class quiz now to confirm your understanding.](#)

Constructor

We will make the constructor take in two parameters (each for the specific transform) that will initialize the two member fields.

```
public TPCBase(Transform cameraTransform, Transform playerTransform)  
{  
    mCameraTransform = cameraTransform;  
    mPlayerTransform = playerTransform;  
}
```

Methods

For our implementation, we will have only one method named **Update**. This method will be abstract, and all derived classes will need to implement this method.

```
public abstract void Update();
```

[Click here and do an in-class quiz now to confirm your understanding.](#)

The complete code for TPCBase class is as below.

```
public abstract class TPCBase
{
    protected Transform mCameraTransform;
    protected Transform mPlayerTransform;

    public Transform CameraTransform
    {
        get
        {
            return mCameraTransform;
        }
    }
    public Transform PlayerTransform
    {
        get
        {
            return mPlayerTransform;
        }
    }

    public TPCBase(Transform cameraTransform, Transform playerTransform)
    {
        mCameraTransform = cameraTransform;
        mPlayerTransform = playerTransform;
    }

    public abstract void Update();
}
```

Third-Person Track Camera Control

We will name this class as **TPCTrack**, and we will derive this class from the **TPCBase** class. Remember, our **TPCBase** was an abstract class with an abstract method called **Update**. So, here for **TPCTrack** class, we will have to override the base class' **Update** method with the correct implementation of mechanics for our camera tracking our Player.

The track camera control will be a straightforward implementation, where we will let the camera track the Player by setting the **LookAt** value to be the designated game object's position. In this mode, the camera does not change its position. This implementation is by far the most straightforward third-person camera control. Let us get to the code.

```
public class TPCTrack : TPCBase
{
    public TPCTrack(Transform cameraTransform, Transform playerTransform)
        : base(cameraTransform, playerTransform)
    {
    }

    public override void Update()
    {
        Vector3 targetPos = mPlayerTransform.position;
        mCameraTransform.LookAt(targetPos);
    }
}
```

[Click and do an in-class quiz now to confirm your understanding.](#)

Testing Third-Person Track Camera Control

We have implemented our first third-person camera control. We will now test this with our Player. For this, we will implement it in the **ThirdPersonCamera** script that we have created.

```
public class ThirdPersonCamera : MonoBehaviour
{
    public Transform mPlayer;

    TPCBase mThirdPersonCamera;

    void Start()
    {
        mThirdPersonCamera = new TPCTrack(transform, mPlayer);
    }

    void LateUpdate()
    {
        mThirdPersonCamera.Update();
    }
}
```

[Click and do an in-class quiz now to confirm your understanding.](#)

From Unity: *LateUpdate is called every frame if the Behaviour is enabled. LateUpdate is called after all Update functions have been called. This is useful to order script execution. For example, a follow camera should always be implemented in LateUpdate because it tracks objects that might have moved inside Update.*

 **Click Play** and view the behaviour.

For our animated character, the position of the transform is at height 0. If you want to track the target's height, we will need to add the Player's height into the **LookAt** position.

So, we will have to change the **Update** function for the TPCTrack to cater for the Player's height.

Programming Assignment 2 – Track Third-Person With Player Height

Track third-person camera control with Player height. Add the Player height to the **Update** method.

```
public override void Update()
{
    Vector3 targetPos = mPlayerTransform.position;
    // Add your code to cater for the player height.
    // Add the player height to the targetPos variable
    // before setting to the LookAt
    // amend and implement your code here.
    // mCameraTransform.LookAt(targetPos);
}
```

After you have completed your implementation, click **Play** and view the behaviour.

You can see the difference between the two implementations now. You can also change the height of the Player depending on where you would want the camera to look.

The GameConstants Class

Before we proceed with other third-person camera implementations, let us do some **Code Refactoring**. We have seen above that we will need to store the player height variable for us to use in our third-person track camera implementation. As we move with more third-person camera implementations, we will require more such variables. These variables are game constants that do not change during the execution of the game. Let us add a new **static** class called **GameConstants**.

[Click and do an in-class quiz now to confirm your understanding.](#)

Instead of keeping just the player height, we can use a **Vector3** variable that stores the Camera offset. If we wish to use the player height, then this value will be (0.0f, playerHeight, 0.0f), where playerHeight is the variable that stores the player height.

```
public static class GameConstants
{
    public static Vector3 CameraPositionOffset { get; set; }
}
```

We can set the values of the game constants from the Unity Editor for now. In Week 6, we will learn about FILE I/O using C# and write code to set the values of these game constants from a comma-separated variable (CSV) file.

We modify the ThirdPersonCamera script to set these values to the GameConstants class.

```
public class ThirdPersonCamera : MonoBehaviour
{
    *****

    // Get from Unity Editor.
    public Vector3 mPositionOffset = new Vector3(0.0f, 2.0f, -2.5f);

    void Start()
    {
        // Set to GameConstants class so that other objects can use.
        GameConstants.CameraPositionOffset = mPositionOffset;
        mThirdPersonCamera = new TPCTrack(transform, mPlayer);
    }

    *****
}
```

Then we refactor our implementation of **TPCTrack Update** method as:

```
public override void Update()
{
    Vector3 targetPos = mPlayerTransform.position;

    // We add the camera offset on the Y-axis.
    targetPos.y += GameConstants.CameraPositionOffset.y;
    mCameraTransform.LookAt(targetPos);
}
```

Third-Person Follow Camera Control

Our following implementation of a third-person camera control will be called **TPCFollow**, where the camera follows the target player. The Player moves, and the camera follows. There are two possible implementations. In the first implementation, the camera only follows and tracks the position of the Player. In the second implementation, the camera follows and tracks both the position and the rotation of the Player.

To achieve the above, we will introduce two new variables.

- The first variable to store the initial rotation offset of the camera, and
- The second variable is to hold the damping factor to smoothen the changes to the position and rotation of the camera. We will use this with **Lerp**.

We will add these to our **GameConstants** class and set the values through Unity Editor.

```
public static class GameConstants
{
    public static Vector3 CameraAngleOffset { get; set; }
    public static Vector3 CameraPositionOffset { get; set; }
    public static float Damping { get; set; }
}

public class ThirdPersonCamera : MonoBehaviour
{
    ***
    public Vector3 mAngleOffset = new Vector3(0.0f, 0.0f, 0.0f);
    [Tooltip("The damping factor to smooth the changes in position and rotation of the camera.")]
    public float mDamping = 1.0f;

    void Start()
    {
        // Set the game constant parameters to the GameConstants class.
        GameConstants.Damping = mDamping;
        GameConstants.CameraPositionOffset = mPositionOffset;
        GameConstants.CameraAngleOffset = mAngleOffset;
        ***
    }
    ***
}
```

We implement a **TPCFollow** class that will act as the base class for the two derived classes **TPCFollowTrackPosition** and **TPCFollowTrackPositionAndRotation**.

Programming Assignment 3 – Implement the Update Method for TPCFollow

In this programming task, you will implement some sections of the **Update** method for the **TPCFollow** class. I have provided hints at locations where you need to do your implementations. We will discuss this in class and implement it.

```
public abstract class TPCFollow : TPCBase
{
    public TPCFollow(Transform cameraTransform, Transform playerTransform)
        : base(cameraTransform, playerTransform)
    {
    }

    public override void Update()
    {
        // Now we calculate the camera transformed axes.
        // We do this because our camera's rotation might have changed
        // in the derived class Update implementations. Calculate the new
        // forward, up and right vectors for the camera.
        Vector3 forward = *** Your code ***;
        Vector3 right = *** Your code ***;
        Vector3 up = *** Your code ***;

        // We then calculate the offset in the camera's coordinate frame.
        // For this we first calculate the targetPos
        Vector3 targetPos = mPlayerTransform.position;



        // Add the camera offset to the target position.
        // Note that we cannot just add the offset.
        // You will need to take care of the direction as well.
        Vector3 desiredPosition = *** Your code ***;

        // Finally, we change the position of the camera,
        // not directly, but by applying Lerp.
        Vector3 position = Vector3.Lerp(mCameraTransform.position,
            desiredPosition, Time.deltaTime * GameConstants.Damping);
        mCameraTransform.position = position;
    }
}
```


Third-Person Follow Track Position Camera Control

We are going to derive this class from TPCFollow. This class will implement the **Update** method so that the camera follows the Player and tracks the position. Do note that the initial offset position and rotation for the camera still applies.

```
public class TPCFollowTrackPosition : TPCFollow
{
    public TPCFollowTrackPosition(Transform cameraTransform, Transform
playerTransform)
        : base(cameraTransform, playerTransform)
    {
    }

    public override void Update()
    {
         Create the initial rotation quaternion based on the
         camera angle offset.
        Quaternion initialRotation =
            Quaternion.Euler(GameConstants.CameraAngleOffset);

        // Now rotate the camera to the above initial rotation offset.
        // We do it using damping/Lerp
        // You can change the damping to see the effect.
        mCameraTransform.rotation =
            Quaternion.RotateTowards(mCameraTransform.rotation,
                initialRotation,
                Time.deltaTime * GameConstants.Damping);

        // We now call the base class Update method to take care of the
        // position tracking.
         base.Update();
    }
}
```

To use this new third-person camera control, we will initialize the mThirdPersonCamera to **TPCFollowTrackPosition**.

```
void Start()
{
    ***
    //mThirdPersonCamera = new TPCTrack(transform, mPlayer);
    mThirdPersonCamera = new TPCFollowTrackPosition(transform, mPlayer);
}
```

Click **Play** and view the behaviour.

Third-Person Follow Track Position and Rotation Camera Control

Like the above, we are going to derive this class too from **TPCFollow**. **TPCFollowTrackPositionAndRotation** class will implement the **Update** method such that the camera follows the Player and tracks both the position and the rotation of the Player. Do note that the initial offset position and rotation for the camera still applies.

```
public class TPCFollowTrackPositionAndRotation : TPCFollow
{
    public TPCFollowTrackPositionAndRotation(Transform cameraTransform,
    Transform playerTransform)
        : base(cameraTransform, playerTransform)
    {
    }

    public override void Update()
    {
        // We apply the initial rotation to the camera.
        Quaternion initialRotation =
            Quaternion.Euler(GameConstants.CameraAngleOffset);

        // Allow rotation tracking of the player
        // so that our camera rotates when the Player rotates and at the same
        // time maintain the initial rotation offset.
        mCameraTransform.rotation = Quaternion.Lerp(
            mCameraTransform.rotation,
            mPlayerTransform.rotation * initialRotation,
            Time.deltaTime * GameConstants.Damping);

        base.Update();
    }
}
```

To use this new third-person camera control, we will initialize the **mThirdPersonCamera** to

TPCFollowTrackPositionAndRotation.

```
void Start()
{
    ***
    //mThirdPersonCamera = new TPCTrack(transform, mPlayer);
    //mThirdPersonCamera = new TPCFollowTrackPosition(transform, mPlayer);
    mThirdPersonCamera = new TPCFollowTrackPositionAndRotation(transform,
    mPlayer);
}
```

Click **Play** and view the behaviour.

Programming Assignment 4 – Implement the TPCTopDown Camera Control

In this programming task, you will implement the **TPCTopDown** third-person camera control. **TPCTopDown** is a simple Top-Down camera mode where the camera looks down on the Player from an altitude. This mode will not use the **mRotationOffset** and the x and z values of the **mPositionOffset**. Give it a try.

Follow a similar line of thoughts as the previous few implementations.

Step 1: Derive a new **TPCTopDown** class from **TPCBase** class.

Step 2: Override the **Update** method

Step 3: In the Update method, get the player position in a temporary variable **targetPos**.

Step 4: Add the **CameraPositionOffset.y** value to the **targetPos**.

Step 5: Use **Lerp** to move the camera to this **targetPos**.

Step 6: Rotate the camera to look down.

That's all. Now initialize the **mThirdPersonCamera** with **TPCTopDown**

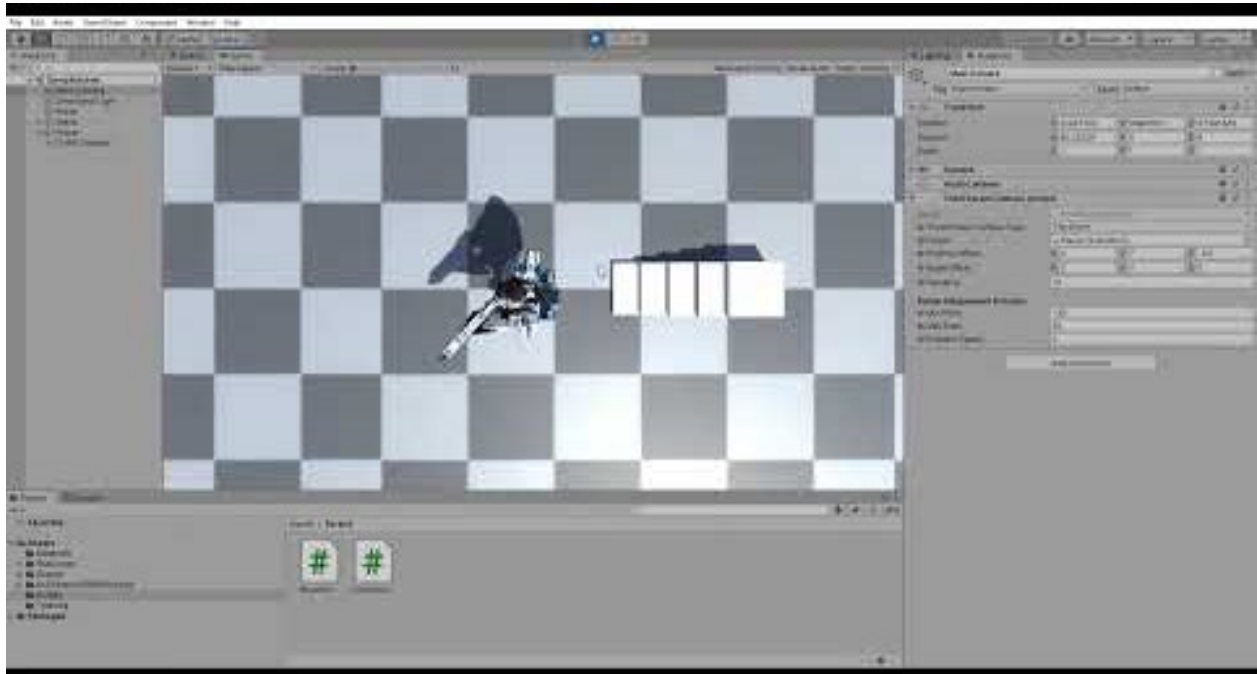
```
void Start()
{
    // Set the game constant parameters to the GameConstants class.
    GameConstants.Damping = mDamping;
    GameConstants.CameraPositionOffset = mPositionOffset;
    GameConstants.CameraAngleOffset = mAngleOffset;

    //mThirdPersonCamera = new TPCTrack(transform, mPlayer);
    //mThirdPersonCamera = new TPCFollowTrackPosition(transform, mPlayer);
    //mThirdPersonCamera = new TPCFollowTrackPositionAndRotation(transform,
mPlayer);
    mThirdPersonCamera = new TPCTopDown(transform, mPlayer);
}
```

Click **Play** and view the behaviour.

We have concluded the worksheet on how to create a custom third-person camera control in Unity using C# by applying the concepts of inheritance and polymorphism.

Below is the final video.



In our following worksheet, we will continue with this project to port our third-person camera to an Android touch control phone and implement another camera control that is useful for touch control devices.