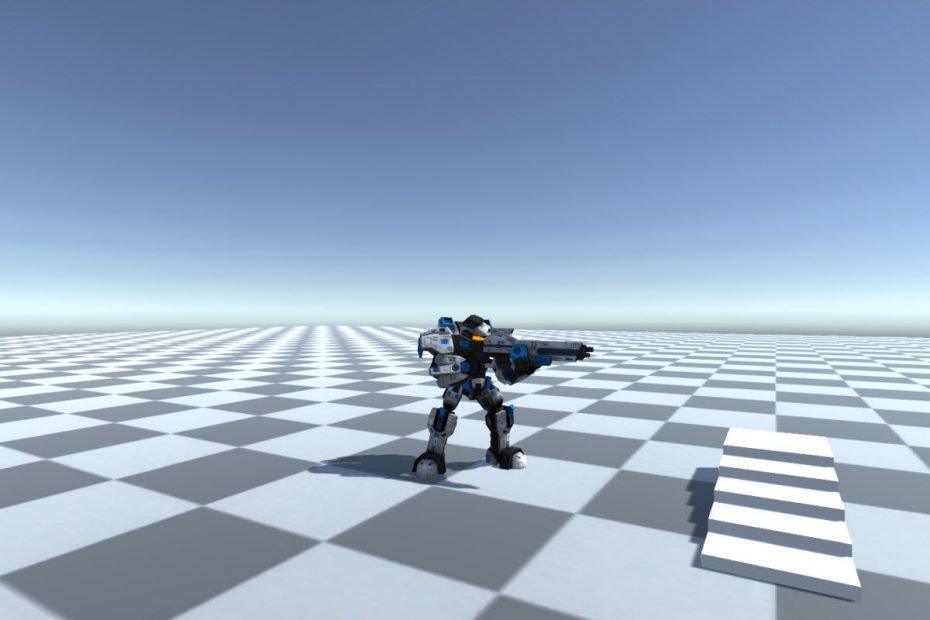
# Touch Input and Export to Android



Shamim Akhtar

## Introduction

In this worksheet, we will port the third-person camera types to Android, and along the way, we will learn about mobile touch implementation, Android build settings and the export process to Android mobile devices.

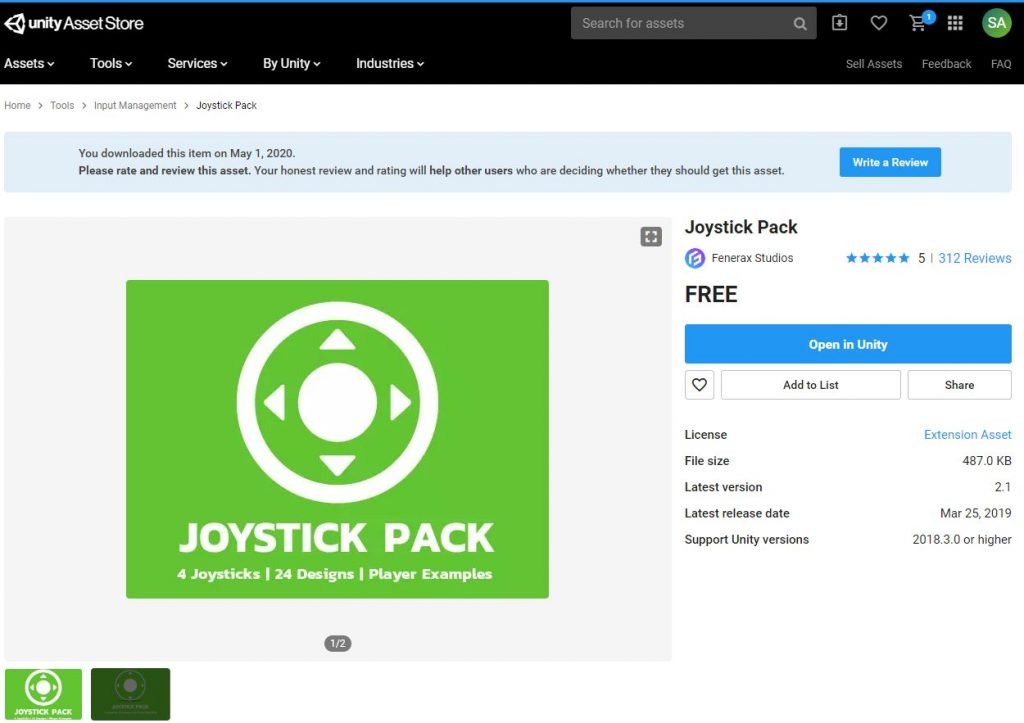
This worksheet is an extension of what we have implemented in our previous worksheet. Here we will port the five different types, viz.,

* Track
* Follow
* Follow and Track Rotation
* Follow and Independent Rotation
* Top-Down

of third-person camera controls to a touch-sensitive device (we will export to Android phone).

## Section 3 – Porting the Camera Controls to Android

To implement the movement and camera control, we will use a virtual joystick package. You can directly go to <https://assetstore.unity.com/packages/tools/input-management/joystick-pack-107631#content> or go to Unity Asset Store and search for "Joystick Pack".



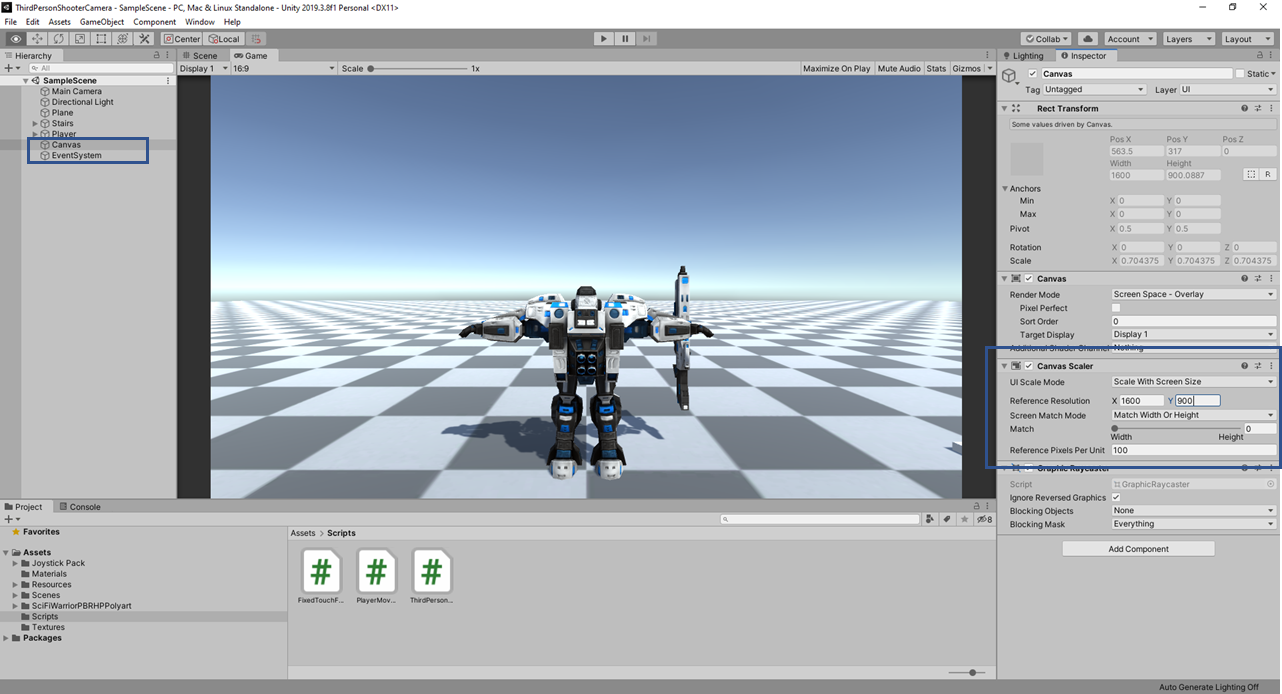
Download and import the package. You will also need the [FixedTouchField.cs](https://faramira.com/downloads/Tutorials/FixedTouchField.cs" \t "_blank) script. Download and put in your scripts folder. This script allows you to create a touch field. We will see the usage later in our worksheet.

In our first week, we created the third-person camera controls. These controls are primarily following the player position and orientation. For standard PC, Mac or Linux build, we used the W, A, S and D keys (and in combination with the left-shift key) to move (and run) and navigate the Player. For mobile touch, we cannot use the W, A, S and D keys. Instead, we use the joystick type control to navigate.

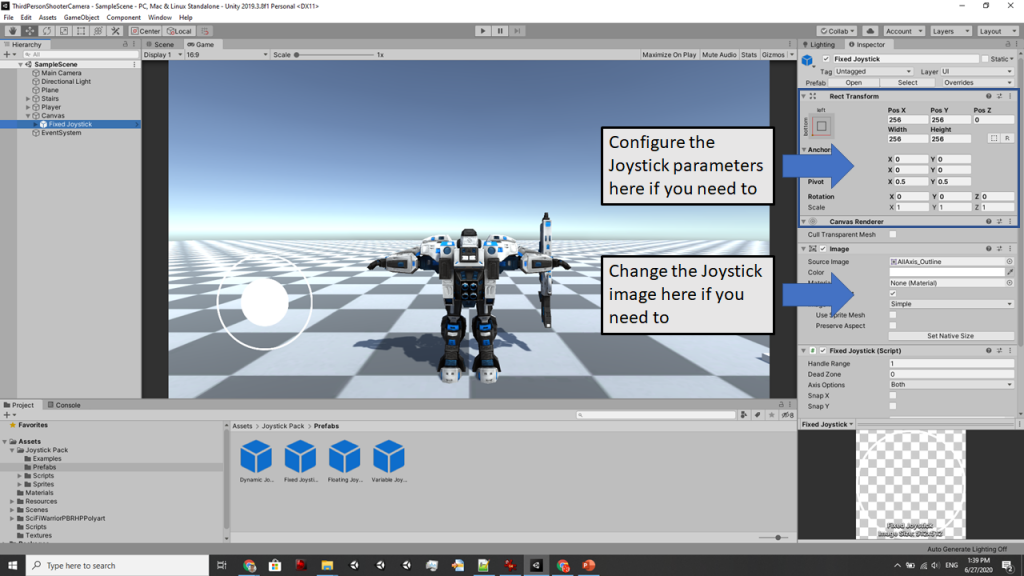
We will use the above virtual joystick for navigating the Player.

### Create a Canvas and Add the Joystick

Right-click on the **Hierarchy**click on **UI -> Canvas. Make sure you change the Canvas Scaler, as shown in the diagram below.**



Now browse to**Joystick Pack->Prefabs** and drag and drop the **Fixed Joystick.prefab**into the **Canvas**.



You can configure the Joystick parameters, including the images. But we will leave it as it is now.

### Port the Player Movement Script

We will now port our player movement so that the movement now uses this virtual joystick. Open the **PlayerMovement.cs** script file. In our original worksheet, we took inputs from the **GetAxis**Unity functions. Now we will take inputs from our virtual joystick controls. We will add a public variable called **mJoystick.** Then we will associate the Fixed Joystick in the canvass with this variable.

We will use platform-dependent conditional compilation.

#if UNITY\_ANDROID

public FixedJoystick mJoystick;

#endif

Then in the Update method, we make changes to get inputs from the joystick rather than from the mouse for Android build.

#if UNITY\_STANDALONE

float hInput = Input.GetAxis("Horizontal");

float vInput = Input.GetAxis("Vertical");

#endif

#if UNITY\_ANDROID

float hInput = mJoystick.Horizontal;

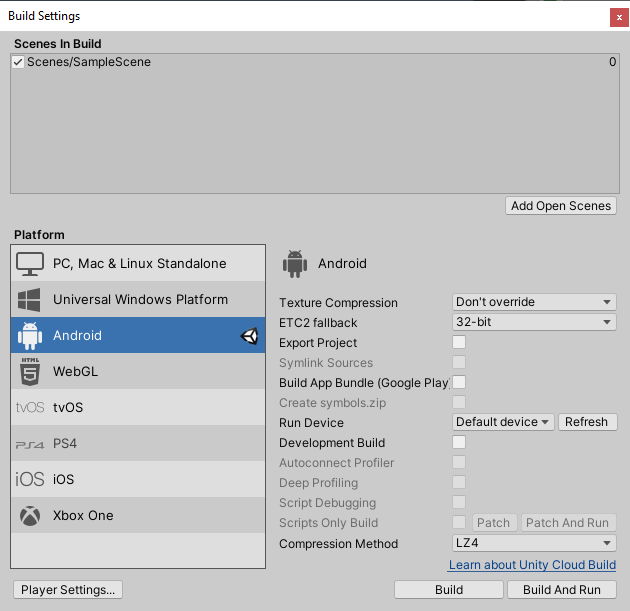
float vInput = mJoystick.Vertical;

#endif

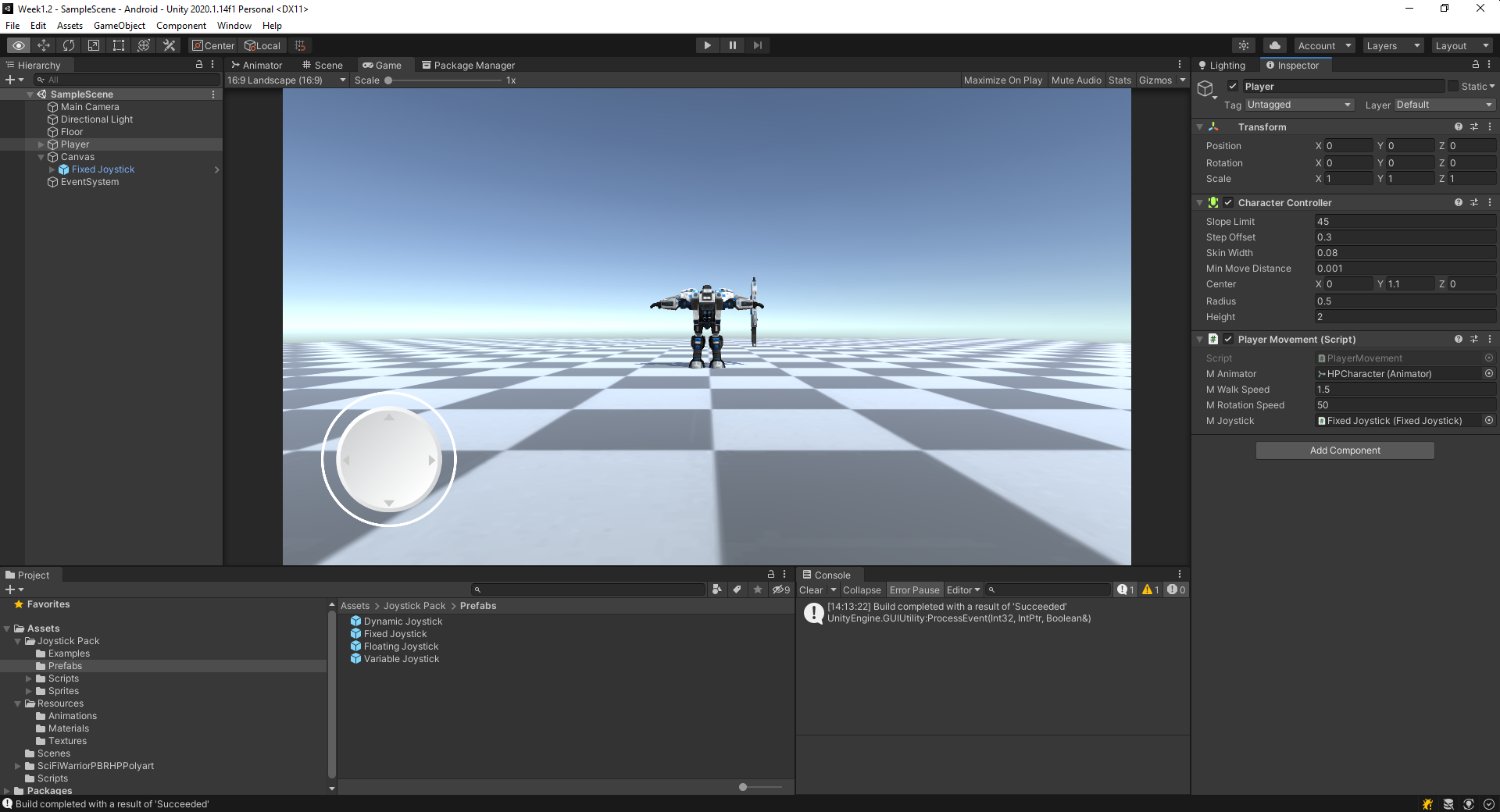
The above changes mean that for Android build the **hInput** and **vInput** values are coming from the virtual joystick. Let's test this and see if it works. We will test our application with any of the existing third-person camera types that we built.

### Change the Build Settings to Android

Go to **File->Build Settings.** Select **Android**. It will take a while to change the build settings. Note: You should have the Android SDK already installed by now. If not, then go ahead and install the Android SDK. It will take quite a while to install the SDK.



Once you have changed the build settings to Android, you can now drag and drop the **Fixed Joystick** from the **Canvas** to the **PlayerMovement** script’s **M Joystick** field.



Click **Play** and drag the virtual joystick for the Player to move. It should work. If it doesn't then debug and see why it is not working. You can also build and run on your Android mobile device.

Test with all the four different third-person camera types that we created in Worksheet 1.

You must have noticed that even if we move the Handle of the virtual joystick to the fullest extent, we still cannot make the character run. While using the virtual joystick, we cannot press the left shift key. The solution for this is simple. We multiply the horizontal and the vertical inputs by a factor of 2.0f;

#if UNITY\_ANDROID

float hInput = 2.0f \* mJoystick.Horizontal;

float vInput = 2.0f \* mJoystick.Vertical;

#endif

Now click Play and see the behaviour.

### Apply Editor Changes at Runtime

Let's take a little detour before we proceed with the rest of the worksheet.

Have you noticed that anytime we change the values of the Camera Offset, Angle Offset, Damping or any other values from the Unity Editor we will need to stop playing and then click **Play** again? Why is this so? **Discuss in class**.

You also noticed that whenever we want to change our third-person camera type, we will need to uncomment and comment on specific code. For example, let's say we want to enable our TPCFollowTrackPositionAndRotation then we will need to go to our **ThirdPersonCamera.cs** and make the following changes.

//mThirdPersonCamera = new TPCTrack(transform, mPlayer);

//mThirdPersonCamera = new TPCFollowTrackPosition(transform, mPlayer);

mThirdPersonCamera = new TPCFollowTrackPositionAndRotation(transform, mPlayer);

//mThirdPersonCamera = new TPCTopDown(transform, mPlayer);

We do this so that we can have the one and only third-person camera that we want.

The above procedure of doing things is very inconvenient. What if we can make changes at game runtime and see the effect immediately? That means we will not stop the game, make changes and then click on Play every time we make changes. Instead, we click **Play** one time, then make changes while the game is in Play mode and then see the effect. For this to happen, we will create two significant changes.

The first change is easy. We reset the values in every **Update** to **GameConstants**. That way, any changes made to the editor values will immediately take effect. So, besides in **Start** (as below)

void Start()

{

// Set the game constant parameters to the GameConstants class.

GameConstants.Damping = mDamping;

GameConstants.CameraPositionOffset = mPositionOffset;

GameConstants.CameraAngleOffset = mAngleOffset;

\*\*\*\*

\*\*\*\*

}

we will also do the same in Update

void Update()

{

// Update the game constant parameters every frame

// so that changes applied on the editor can be reflected

GameConstants.Damping = mDamping;

GameConstants.CameraPositionOffset = mPositionOffset;

GameConstants.CameraAngleOffset = mAngleOffset;

}

Click **Play** and try out. Change the various values in your editor and see the effect immediately on your third-person camera behaviour.

For the second change, we want our third-person camera type to change at runtime so that we can see the behaviour of all four third-person camera types when we are in play mode. For this to happen, first of all, we will define an enumeration type for all the different kind of third-person camera controls.

public enum CameraType

{

Track,

Follow\_Track\_Pos,

Follow\_Track\_Pos\_Rot,

Topdown,

Follow\_Independent, //we have not implemented this mode yet

}

Then we create a public variable called mCameraType that by default, is set to one of the possible values.

public CameraType mCameraType = CameraType.Follow\_Track\_Pos;

We then create a Dictionary of CameraType and the TPCBase as following:

Dictionary<CameraType, TPCBase> mThirdPersonCameraDict = new Dictionary<CameraType, TPCBase>();

Then in the Start method, we fill the Dictionary with all possible third-person camera types.

//mThirdPersonCamera = new TPCTrack(transform, mPlayer);

//mThirdPersonCamera = new TPCFollowTrackPosition(transform, mPlayer);

//mThirdPersonCamera = new TPCFollowTrackPositionAndRotation(transform, mPlayer);

//mThirdPersonCamera = new TPCTopDown(transform, mPlayer);

mThirdPersonCameraDict.Add(CameraType.Track, new TPCTrack(transform, mPlayer));

mThirdPersonCameraDict.Add(CameraType.Follow\_Track\_Pos, new TPCFollowTrackPosition(transform, mPlayer));

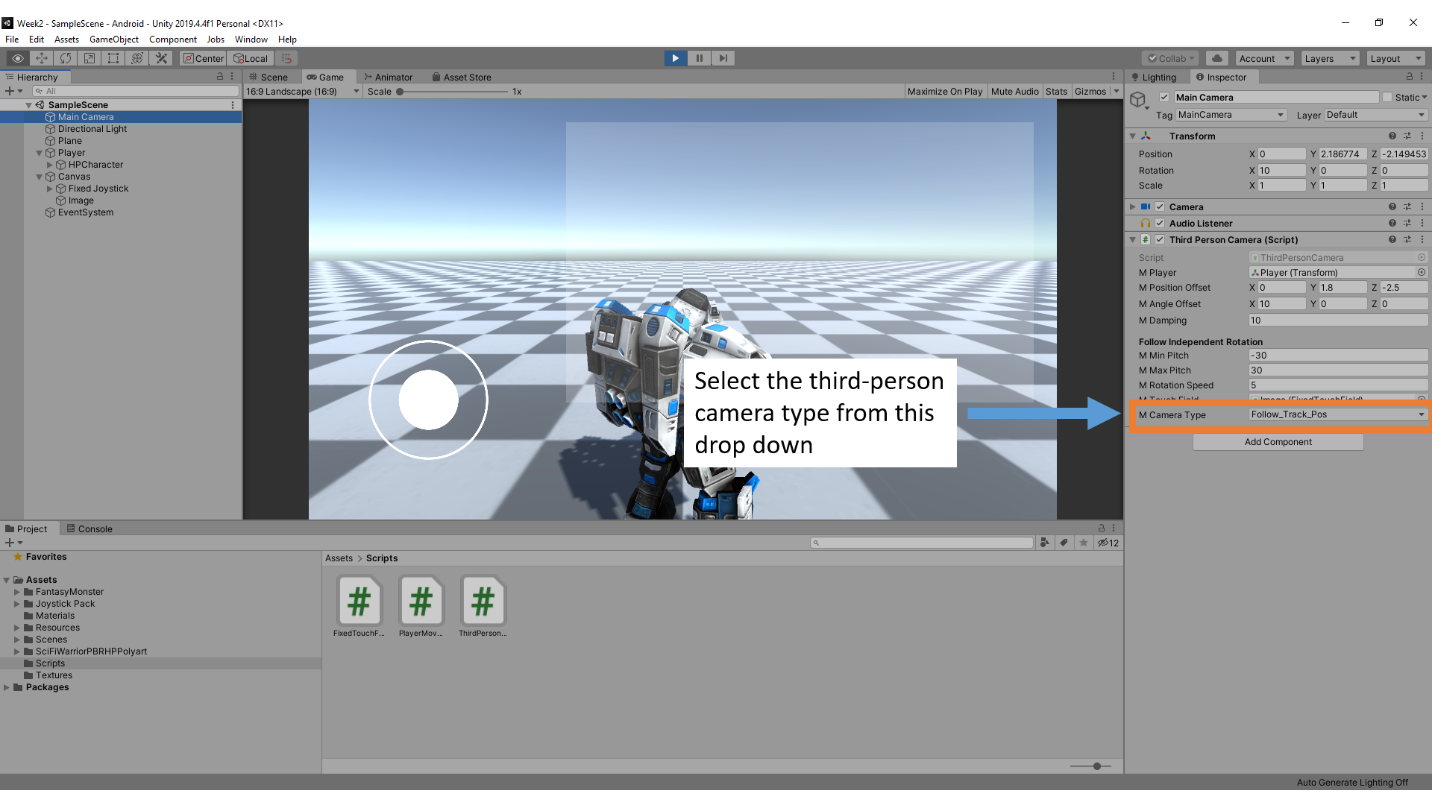
mThirdPersonCameraDict.Add(CameraType.Follow\_Track\_Pos\_Rot, new TPCFollowTrackPositionAndRotation(transform, mPlayer));

mThirdPersonCameraDict.Add(CameraType.Topdown, new TPCTopDown(transform, mPlayer));

mThirdPersonCamera = mThirdPersonCameraDict[mCameraType];

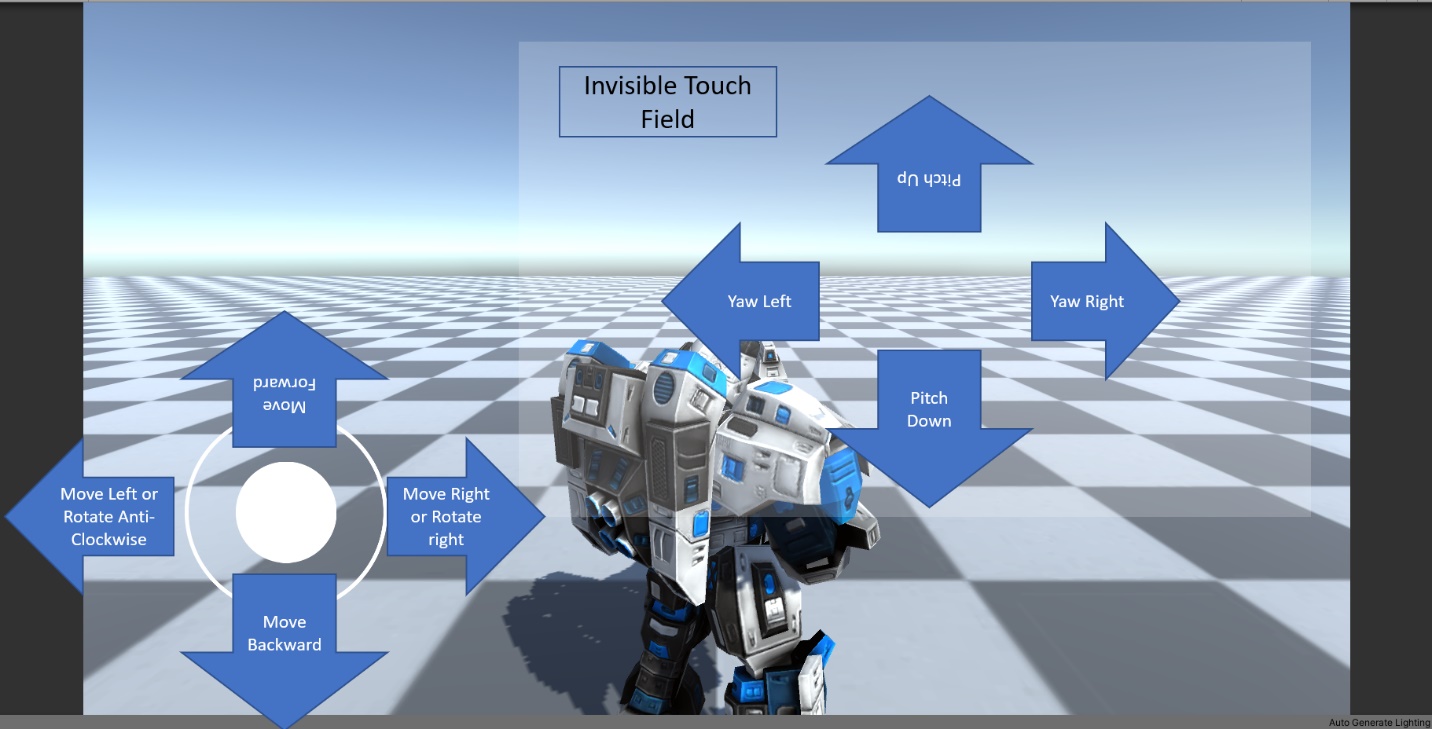
We will also set the actual third-person camera by setting the mThirdPersonCamera to the camera with the key set to mCameraType (as shown above in the last line).

Now, to be able to select the third-person camera type at runtime, duplicate the same line in the **Update** method, click Play and use the DropDown in Unity Editor to choose the third-person camera that you want to apply.



### Follow with Independent Camera Rotation

In this section, we will implement the final, and the fifth type of third-person camera called the TPCFollowIndependentRotation. In this third-person camera implementation, the camera will not only follow the Player but also will have independent rotation using the touch-field. This type of camera implementation is quite common for the mobile third-person shooter and RPG type of games.

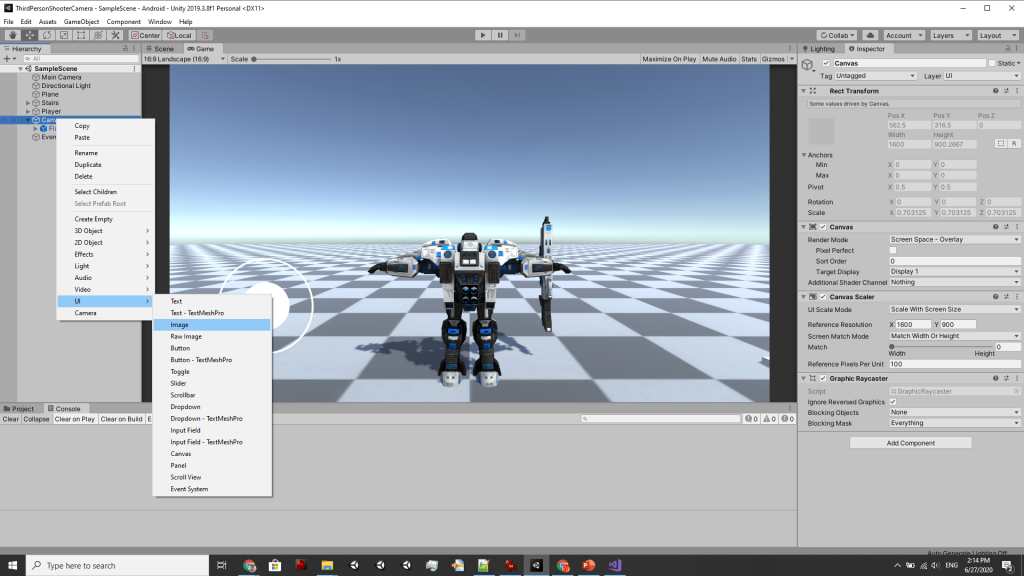


The independent rotation for the camera is limited between a min and max values for pitch. You can pitch the camera by moving your finger up or down on the invisible touch field. By moving your finger left and right on the invisible touch-field, you can yaw the camera left and right. Finally, when you move the Player using the virtual joystick, **the Player moves forward in the camera's look at direction**.

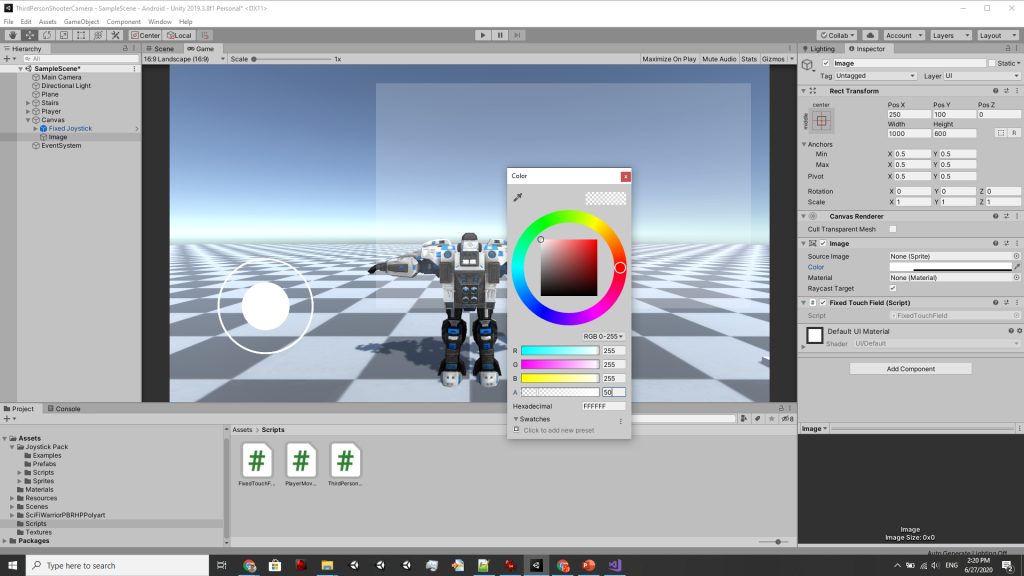
Let's implement this third-person camera behaviour.

#### Implement the Touch Field for Camera

Right-click on the **Canvas**and create a **UI->Image**.



Attach the **FixedTouchFiled.cs** script to this image. Resize and position the image to your liking (this is where the user will touch and drag to orient the camera. Typically in the top-right section of the scene).



Change the transparency to 50 for now so that it is slightly visible. Later we will make the image to be completely transparent to make it invisible.

#### TPCFollowIndependentRotation

Open the **ThirdPersonCamera.cs.** Add three new properties in **GameConstants**. These are:

public static float RotationSpeed { get; set; }

public static float MinPitch { get; set; }

public static float MaxPitch { get; set; }

The RotationSpeed is the speed with which the camera rotates. MinPitch and MaxPitch are the minimum and maximum angles that the camera can look up and down (or pitch), respectively. You can play around with these variables later in the Unity Editor to see what values fit correctly.

Now implement the **TPCFollowIndependentRotation** camera control.

public class TPCFollowIndependentRotation : TPCBase

{

FixedTouchField mTouchField;

private float angleX = 0.0f;

public TPCFollowIndependentRotation(Transform cameraTransform, Transform playerTransform)

: base(cameraTransform, playerTransform)

{

}

#if UNITY\_ANDROID

public TPCFollowIndependentRotation(Transform cameraTransform, Transform playerTransform, FixedTouchField fixedTouch)

: base(cameraTransform, playerTransform)

{

mTouchField = fixedTouch;

}

#endif

public override void Update()

{

//implement the Update for this camera controls

}

}

You can see above that the constructor for the class has a different signature than the base class. For this camera control, we will require access to the touch-field so that we can get inputs for our camera movement. It is the only third-person camera control among all five that requires either mouse input or the joystick input, the reason being rotating the camera independently.

public override void Update()

{

#if UNITY\_STANDALONE

float mx, my;

mx = Input.GetAxis("Mouse X");

my = Input.GetAxis("Mouse Y");

#endif

#if UNITY\_ANDROID

float mx, my;

mx = mTouchField.TouchDist.x \* Time.deltaTime;

my = mTouchField.TouchDist.y \* Time.deltaTime;

#endif

// We apply the initial rotation to the camera.

Quaternion initialRotation = Quaternion.Euler(GameConstants.CameraAngleOffset);

Vector3 eu = mCameraTransform.rotation.eulerAngles;

angleX -= my \* GameConstants.RotationSpeed;

// We clamp the angle along the Xaxis to be between the min and max pitch.

angleX = Mathf.Clamp(angleX, GameConstants.MinPitch, GameConstants.MaxPitch);

eu.y += mx \* GameConstants.RotationSpeed;

Quaternion newRot = Quaternion.Euler(angleX, eu.y, 0.0f) \* initialRotation;

mCameraTransform.rotation = newRot;

Vector3 forward = mCameraTransform.rotation \* Vector3.forward;

Vector3 right = mCameraTransform.rotation \* Vector3.right;

Vector3 up = mCameraTransform.rotation \* Vector3.up;

Vector3 targetPos = mPlayerTransform.position;

Vector3 desiredPosition = targetPos

+ forward \* GameConstants.CameraPositionOffset.z

+ right \* GameConstants.CameraPositionOffset.x

+ up \* GameConstants.CameraPositionOffset.y;

Vector3 position = Vector3.Lerp(mCameraTransform.position,

desiredPosition,

Time.deltaTime \* GameConstants.Damping);

mCameraTransform.position = position;

}

The above is the Update method for this camera. You can see that for Android build, we take inputs from the touch-field, whereas, for the standalone build, we take the inputs from the mouse.

The rest of the code is similar to our previous Follow camera implementation. Except for the fact that we combine the rotation from the camera with the rotation of the Player.

We will now add this new third-person camera into our main ThirdPersonCamera script. First, add a new enum type for this new third-person camera.

public enum CameraType

{

Track,

Follow\_Track\_Pos,

Follow\_Track\_Pos\_Rot,

Topdown,

Follow\_Independent, // the new third-person camera

}

Second, we instantiate and add this new third-person camera into our Dictionary. For Android build, we will require the FixedTouchField component. Go ahead and add a public variable as below in the **ThirdPersonCamera** class.

public FixedTouchField mTouchField;

Now, instantiate the new camera type and add to the Dictionary.

mThirdPersonCameraDict.Add(CameraType.Track, new TPCTrack(transform, mPlayer));

mThirdPersonCameraDict.Add(CameraType.Follow\_Track\_Pos, new TPCFollowTrackPosition(transform, mPlayer));

mThirdPersonCameraDict.Add(CameraType.Follow\_Track\_Pos\_Rot, new TPCFollowTrackPositionAndRotation(transform, mPlayer));

mThirdPersonCameraDict.Add(CameraType.Topdown, new TPCTopDown(transform, mPlayer));

// We instantiate and add the new third-person camera to the dictionary

#if UNITY\_STANDALONE

mThirdPersonCameraDict.Add(CameraType.Follow\_Independent, new TPCFollowIndependentRotation(transform, mPlayer));

#endif

#if UNITY\_ANDROID

mThirdPersonCameraDict.Add(CameraType.Follow\_Independent, new TPCFollowIndependentRotation(transform, mPlayer, mTouchField));

#endif

Third, we set the newly added **GameConstants** variables in the **Start** and **Update** (in **Update** because we want to see the changes happen at runtime). To do this, add the following variables in the ThirdPersonCamera class.

public float mMinPitch = -30.0f;

public float mMaxPitch = 30.0f;

public float mRotationSpeed = 50.0f;

Now, amend the **Start** method:

GameConstants.Damping = mDamping;

GameConstants.CameraPositionOffset = mPositionOffset;

GameConstants.CameraAngleOffset = mAngleOffset;

GameConstants.MinPitch = mMinPitch;

GameConstants.MaxPitch = mMaxPitch;

GameConstants.RotationSpeed = mRotationSpeed;

And in the **Update** method.

void Update()

{

// Update the game constant parameters every frame

// so that changes applied on the editor can be reflected

GameConstants.Damping = mDamping;

GameConstants.CameraPositionOffset = mPositionOffset;

GameConstants.CameraAngleOffset = mAngleOffset;

GameConstants.MinPitch = mMinPitch;

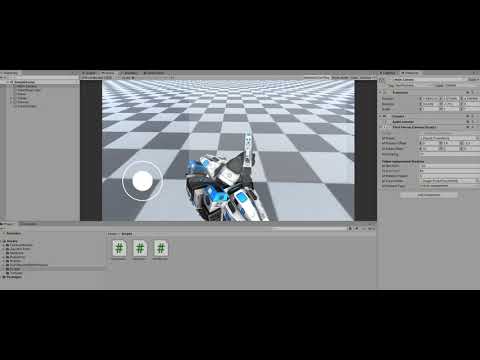
GameConstants.MaxPitch = mMaxPitch;

GameConstants.RotationSpeed = mRotationSpeed;

mThirdPersonCamera = mThirdPersonCameraDict[mCameraType];

}

Click Play and select the Follow\_Independent from the camera type drop-down menu.

[](https://www.youtube.com/embed/Mcz-2VCUWL0?feature=oembed)

You can rotate the camera by clicking and dragging the mouse on the touch-field image.

### Orient Player to Camera LookAt Direction

Observe the implementation. You will see that the Player doesn't orient to our camera's LookAt direction. It happens because we did not change the **PlayerMovement** script. We will now make changes to our **PlayerMovement** script so that the Player automatically orients itself to the LookAt direction of the camera.

Open the **PlayerMovement** file.

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

In the **Update** method, replace the above line of code with the below lines of codes. Before doing so, add a new public boolean variable mFollowCameraForward.

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

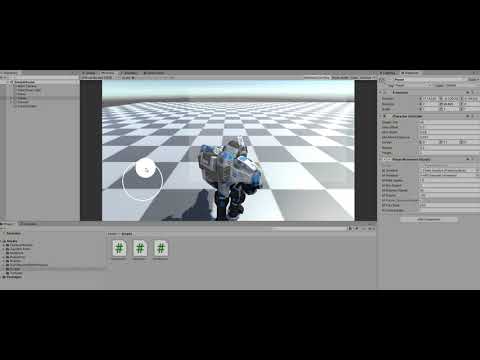
{

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

}

We added a variable mFollowCameraForward that will allow us to toggle between two implementations of the camera. If set to true, then the camera will automatically orient itself to the camera's forward (LookAt) direction.

Now, click Play and test the behaviour. Select **MainCamera** and select Follow\_Independent from the script drop-down. Then select **Player** and check the mFollowCameraForward to enable the camera's direction tracking.

[](https://www.youtube.com/embed/rMmhJ5eIDis?feature=oembed)

Export to your Android device and observer the behaviour. Make sure you make the appropriate selection of the camera type and the mFollowCameraForward to ensure that you have the right settings before you export to your Android device.

This section concludes our Worksheet 2 for this week. Next week we will start with Worksheet 3 where, we will proceed with programming exercises by implementing the Player states. We will touch on the concepts of **interfaces**, **delegates** and some **design patterns**.

We will continue with where we leave this week and implement the aiming, shooting and physical bullet implementation.