Contents

[Sidescrollers 1](#_Toc428526158)

[Super Mario Bros. 1](#_Toc428526159)

[Follows the player 1](#_Toc428526160)

[Margin space 1](#_Toc428526161)

[Locking down the level size 1](#_Toc428526162)

[Coding the Camera 2](#_Toc428526163)

[Finding the player 2](#_Toc428526164)

[Initial movement of the camera 3](#_Toc428526165)

[Adding the Margin and Smooth 4](#_Toc428526166)

[Minimum and Maximum world size 5](#_Toc428526167)

# Sidescrollers

A sidescroller is a game that generally works in a 2D space, so making a camera for it should be easy right? Well…. Yes, and no.

It is quite easy to create a very simple camera script for a first person game, it only involves adding a few simple lines of code. But there is a lot you can add to make your sidescroller camera look smoother and more interesting. How much you want to add will affect how complicated it will get.

## Super Mario Bros.

Have a look at the game Super Mario Bros. 3. The camera has been worked on and enhanced to create the best experience it can for the player.



### Following the player

When the camera follows you the camera also doesn’t just keep you dead centre but gradually moves towards the player when the player changes direction.

### Margin space

When moving forward the camera usually keeps you centred but also seems to give you a bit of wiggle room where the camera does not follow you.

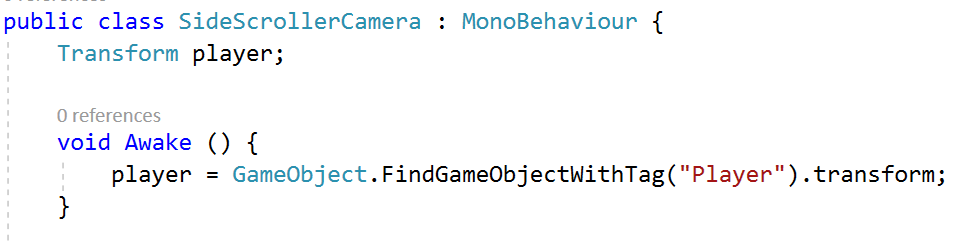
### Locking down the level size

A way to tell the player that there is nothing ahead or behind is to make it so that the camera does not keep moving past a certain point.

# Coding the Camera

First we need a Camera to use. We can use the main camera or create a new one.

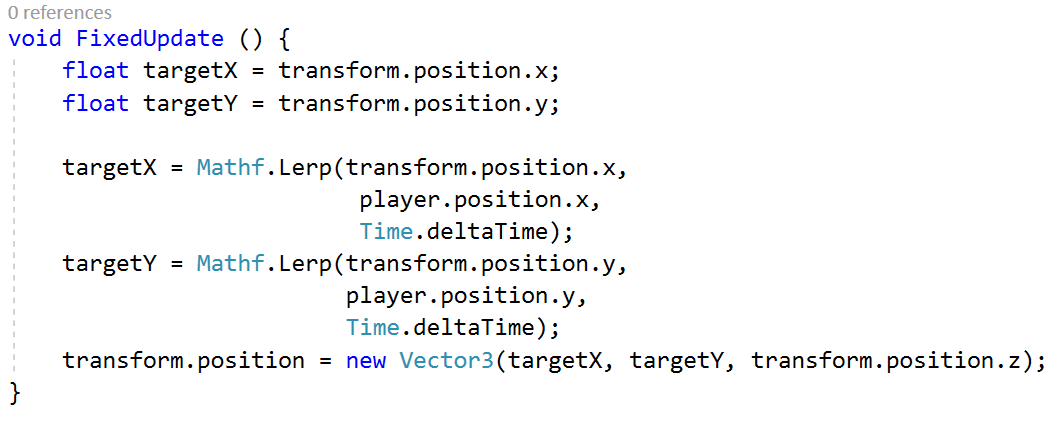
Just remember to disable the camera that is not in use. Make sure to click so the tick shown in the inspector is gone and it will be disabled

1. Create a C# script called **SideScrollerCamera**
2. Attach it to the Camera

Next we need to find the player.

1. Create a private Transform called player
2. Create the **Awake()** function**.**
3. Use the FindObjectWithTag to find the “Player” tag
   1. Pass it into the player transform

## Initial movement of the camera

When coding we want to build up the functionality in steps. So let’s start by getting the camera to just follow the player. No fancy stuff yet.

1. Create a FixedUpdate() function.
2. create two coordinate floats and store the cameras x and y positions.

We will then change these coordinates so they find a spot to move to in between the cameras position and the player’s position using a Math function called Lerp.

### Lerp

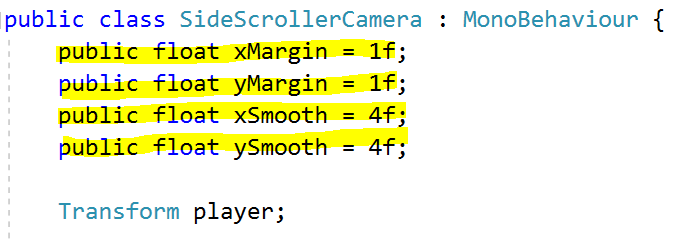
A Lerp Linearly interpolates between two numbers by another number. If we input the values to as **a=1** and from as **b=4** and we set the other number **t**. If **t = 0** it will return the value of **a**, if **t = 1** it will return the value of **b**. If **t = 0.5** it will return the ***average*** of **a** and **b**

We use **deltaTime** to help gradually bring the to value (camera) closer to our from value (player).

1. Create a Lerp for the x and y positions
2. Pass the position values into a new Vector3
   1. Pass in the current cameras z position for z

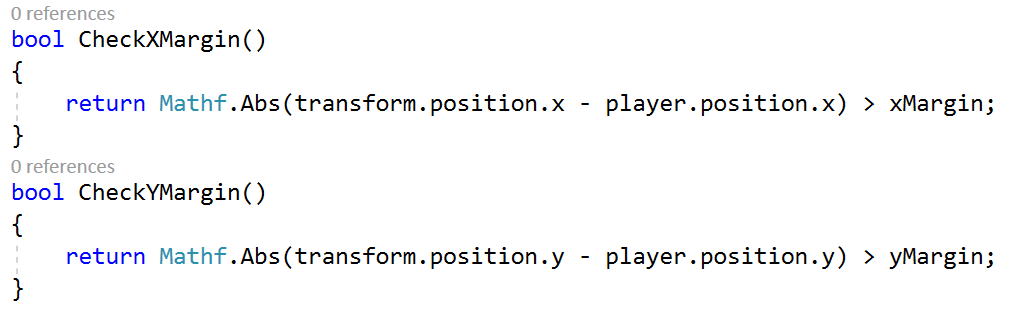
Test out the game. Your camera should now be following the player.

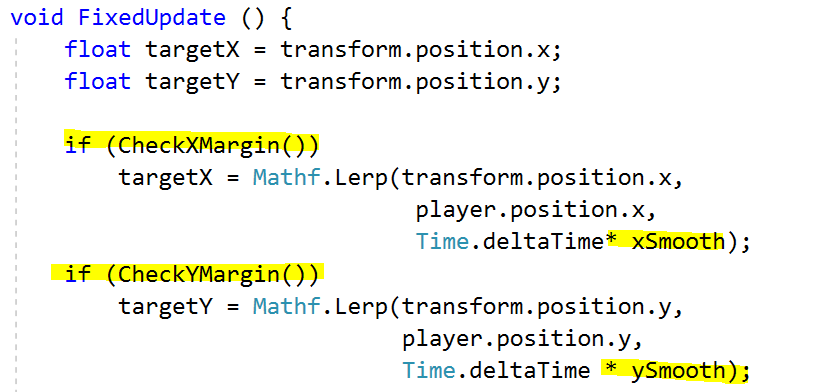
## Adding the Margin and Smooth

We are now going to add a margin of space our player can move before the camera is following, we will also add a smooth to help speed up the camera movement

1. add variables for our margins and our smooths.
   1. One each for **x** and **y**.

Now we want to create some **Boolean** **functions** for use to check our margins in, this will mean we have to type less when we call it. One for **x** and one for **y**

1. Create two Bool functions that check the **distance** between the **camera** and the **player** and returns true if they are greater than the margin.

* In the Fixed Update function

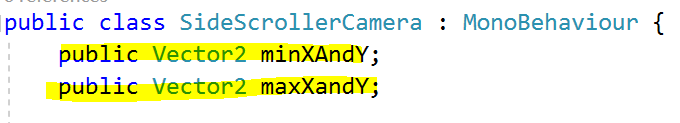
1. Add some **if statements** around our Lerped targets
   1. Check if in Margins

We have also multiplied the deltaTime by our new smooth variables. This will help determine how quickly and smoothly the camera catches up with the targets. Remember one for the x and one for the y.

1. Multiply deltaTime by the correct Smooth variables

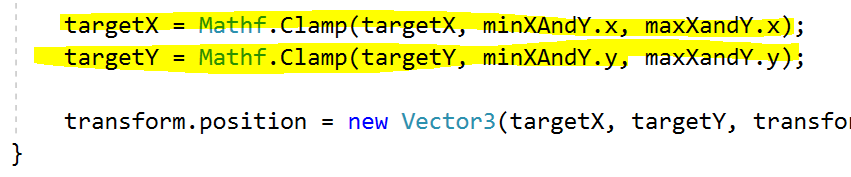
Test the game. We should have a little bit of wiggle room in the middle of the screen now. We should also be able to adjust how the camera moves with the player with our smoothing.

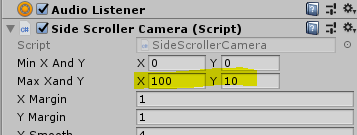
## Minimum and Maximum world size

Now the camera will follow the player all around the game world. But what if we want to lock the area the camera can function in. Just like in the Mario Games. This can help us hide any empty areas in the game world and let the player know they cannot move any further in that direction.

1. Create two **Vector2s**. maxXAndY and minXAndY

A **Vector2** stores the x and y coordinate. This is useful for 2D positioning.

1. Use **Mathf.Clamp** to limit the cameras position
   1. Target, and its min and max values
2. Set the values in the Inspector

**Mathf.Clamp** Clamps a value between a minimum float and maximum float value.

Save the script. Before you test it set the minimum and maximum screen positions. Make sure your player is in between these positions or your character will not be seen.