Lerping

# Following a Target Smoothly

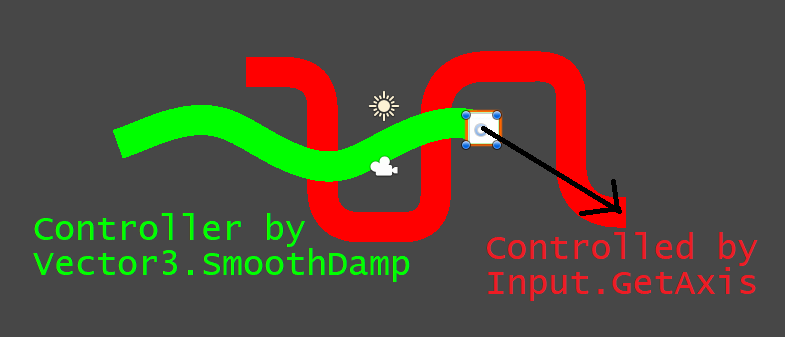
In the previous lectures, we have seen how to move an object using **Input.GetAxis**.

The resulting pattern, however, is not very smooth.

A possible solution is to move an empty GameObject with **Input.GetAxis**.

Then, using a function called **Vector3.SmoothDamp** to make the actual player follow it.

**Vector3.SmoothDamp** follows an object with a slight delay, smoothing the overall movement.



1. Create an empty GameObject, and attach a moving script to it.

|  |
| --- |
| public class MovingTarget : MonoBehaviour  {  [Range(0,10)] public float Speed;  void Update ()  {  float x = Input.GetAxis("Horizontal");  float y = Input.GetAxis("Vertical");  transform.position = transform.position + Speed \* new Vector3(x, y) \* Time.deltaTime;  }  } |

1. Attach a new script to your Player, which use Vector3.SmoothDamp

|  |
| --- |
| public class FollowTarget : MonoBehaviour  {  public GameObject Target;  public Vector3 Velocity = Vector3.zero;  [Range(0,10)] public float SmoothTime = 0.5f;  void Update ()  {  Transform.position = **Vector3.SmoothDamp**  **(**  **Transform.position, // Current position**  **Target.transform.position, // Target position**  **ref Velocity, // Current Velocity**  **SmoothTime // Reaching time**  **);**  }  } |

1. The **FollowTarget** script needs an object to follow. Drag the empty GameObject controlled by the keyboard in the **Target** slot of **FollowTarget**.

|  |
| --- |
| **ref**  Functions in C# can take many input values. **Vector3.SmoothDamp**, for instance, takes 4 inputs. Generally speaking, any change the function makes to a variable is **local**, meaning that it does not propagate outside the function.  While this is usually the desired behaviour, there are cases in which we want to modify an input value. This is the case of **Vector3.SmoothDamp**, which produces as an output both the new position and the new velocity of the moving object. However, C# does not allow to return two values at once. The way to fix this, is to use the **ref** keyword. When used on a variable like **Velocity**, it makes the function able to change its value. Without **ref**, any change that **Vector3.SmoothDamp** does to Velocity would not propagate outside the function itself. By using **ref**, instead, the value of **Velocity** is changed by the function. |

**Vector3.SmoothDamp** is often used to control the camera so that it can follow the player smoothly.

# Lerping Between Two Colours

The following example allows blending two colours together.

The function **Color.Lerp** takes two colours and a float **t** between 0 and 1.

By changing the value of **t**, it is possible to pick colours in between the other two.

|  |
| --- |
| **[ExecuteInEditMode]**  public class ColorLerp : MonoBehaviour  {  [Range(0, 1)]  public float Health = 1;  **public Color Color0;**  **public Color Color100;**  public Color ColorHealth;  void Update ()  {  **ColorHealth = Color.Lerp(Color0, Color100, Health);**  }  } |

|  |
| --- |
| **[ExecuteInEditMode]**  By adding this attribute to a **MonoBehaviour**, its **Update** method will be called in the editor even if the game is not running. This allows testing parts of the game without the need to run it. |

The first issue we encounter is that **Color.Lerp** needs a value between 0 and 1.  
What if the variable **Health** has a different range, for instance (0,100)?

In that case, we need to re-map numbers from the range (0,100) onto the range (0,1), so that it can be passed to **Color.Lerp**.



# Lerping Between Three Colours





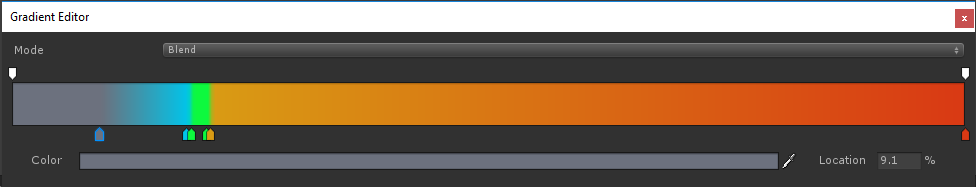
|  |
| --- |
| [Range(0, 100)] public float Health = 100;  public Color Color0;  **public Color Color50;**  public Color Color100;  public Color ColorHealth;  void Update ()  {  **if (Health <= 50) {**  **// Health: [0, 50 ]**  **// Color: [Color0, Color50]**  **ColorHealth = Color.Lerp  (**  **Color0, Color50,**  **Health / 50f**  **);**  **} else {**  **// Health: [50, 100 ]**  **// Color: [Color50, Color100]**  **ColorHealth = Color.Lerp**  **(**  **Color50, Color100,  (Health - 50) / 50**  **);**  **}**  } |

# Lerp Between Any Number Of Colours

If you want a better control over your gradient, Unity comes with a useful class called **Gradient**, which can be used to lerp between any number of colours.

|  |
| --- |
| [ExecuteInEditMode]  public class ColorLerp : MonoBehaviour {  [Range(0, 100)]  public float Health = 100;  **public Gradient ColorGradient;**  public Color ColorHealth;  void Update ()  {  **ColorHealth = ColorGradient.Evaluate(Health / 100);**  }  } |

Unity provides a friendly interface to control the gradient.



Similarly to **Color.Lerp**, The method **Evaluate** takes a number **t** between 0 and 1.

This means that you still have to remap your control values manually in the range (0,1).

# 

# Exercises

1. Draw the **interpolation diagram** that remaps the range (**a**, **b**) to (**c**, **d**).
2. Implement a method that takes an input **x** between **a** and **b**, and remaps it to the range (**c**, **d**).
3. Use the Inspector and the attributes **[Range]** and **[ExecuteInEditMode]** to visualise your solution.

# Lerping Between To Targets

Sometimes you might want to move an object in between to targets.  
For instance, the indicator of a Health bar.

We can still use linear interpolation, but on **Vector3**, instead of **Color**s.

Once again, **Vector3.Lerp** takes a number **t** between 0 and 1.

|  |
| --- |
| public class MoveLerp : MonoBehaviour {  [Range(0,100)]  public float Duration = 10; // In seconds  **public GameObject TargetA;**  **public GameObject TargetB;**  // Update is called once per frame  void Update ()  {  transform.position = **Vector3.Lerp**  **(**  **TargetA.transform.position,**  **TargetB.transform.position,**  **Time.time / Duration**  **);**  }  } |

# Exercises

1. Change the scripts above so that it includes a **StartingTime** and a **Duration**.