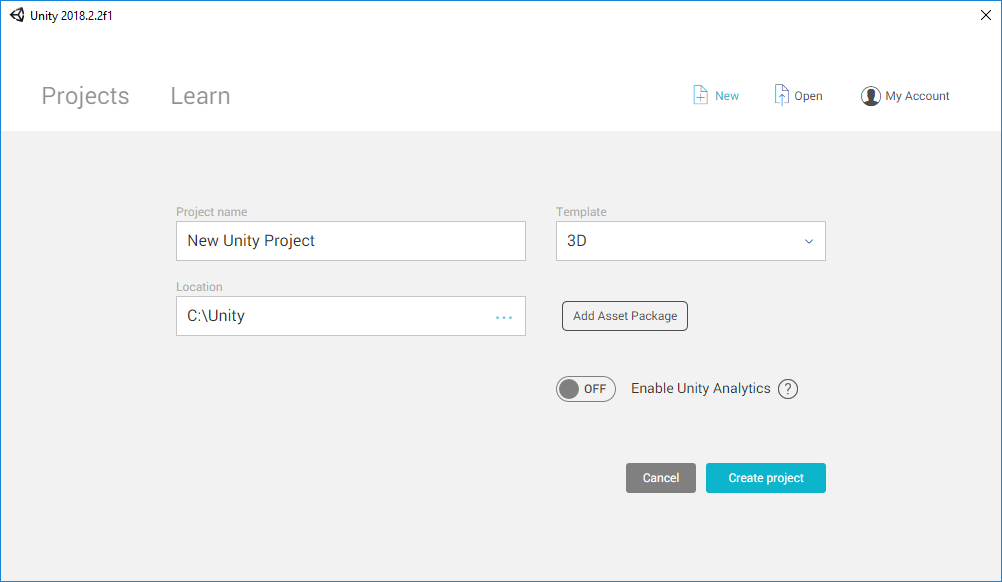
Lab Session 1

**1. Creating a New Project**

The first step to creating any Unity game is to first create a **project**.

1. Start **Unity**.
2. Select **New** (top right) from the project window.
   * If you are making a 2D game, select **2D** from the **Template** drop-down menu.  
     That option will not prevent you from making 3D games; it is used by Unity to set up the camera in new scenes (*perspective* vs *orthogonal*).
   * Choose a name for your project.  
     It cannot be easily changed, but it does not affect the final name of your game in any way.



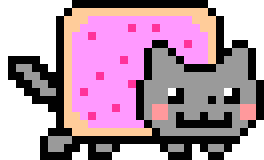
### 2. Importing an Image

This week we will create a simple object that can be moved across the screen with the keyboard.

#### **Finding an Image**

Find an image that you want to use for your character. For the rest of this tutorial, the *Nyan Cat* will be used.

⚠ This has to be a PNG with a *transparent background*.



#### **Importing the Image into Unity**

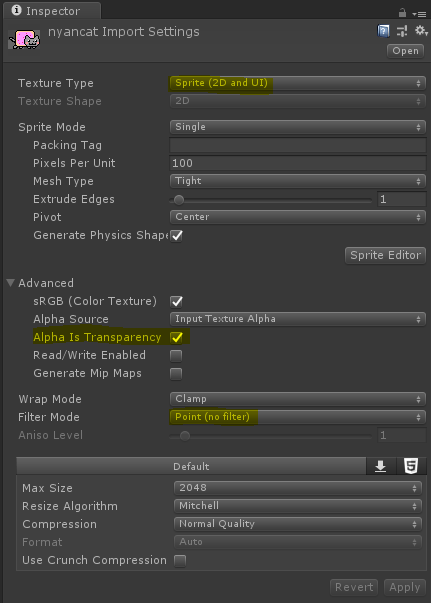
When your newly created project is open, drag the image drop the desktop onto the **Project** window. This will import the image inside Unity.

⚠ Files should only be imported using the Unity editor. Do not copy/move/remove files from your project folder using Explorer (Windows) of Finder (Mac) as this might break your project.

Click on the newly imported image. From the **Inspector** window (right), you can change the settings used to import the image. If you are working with images for 2D games, you want to import them as **sprites**. A sprite is the technical name Unity gives to 2D assets, and they are treated differently, compared to textures that you might want to use for 3D models.

Make sure that:

* **Texture Type**: Sprite (2D and UI)  
  To import the image as a 2D sprite.
* **Alpha is Transparency**: Checked  
  To make sure that transparent pixels from the original image appear transparent in the game as well.
* **Filter Mode**: Point (no filter)  
  To make sure the image is not blurred when zooming in.



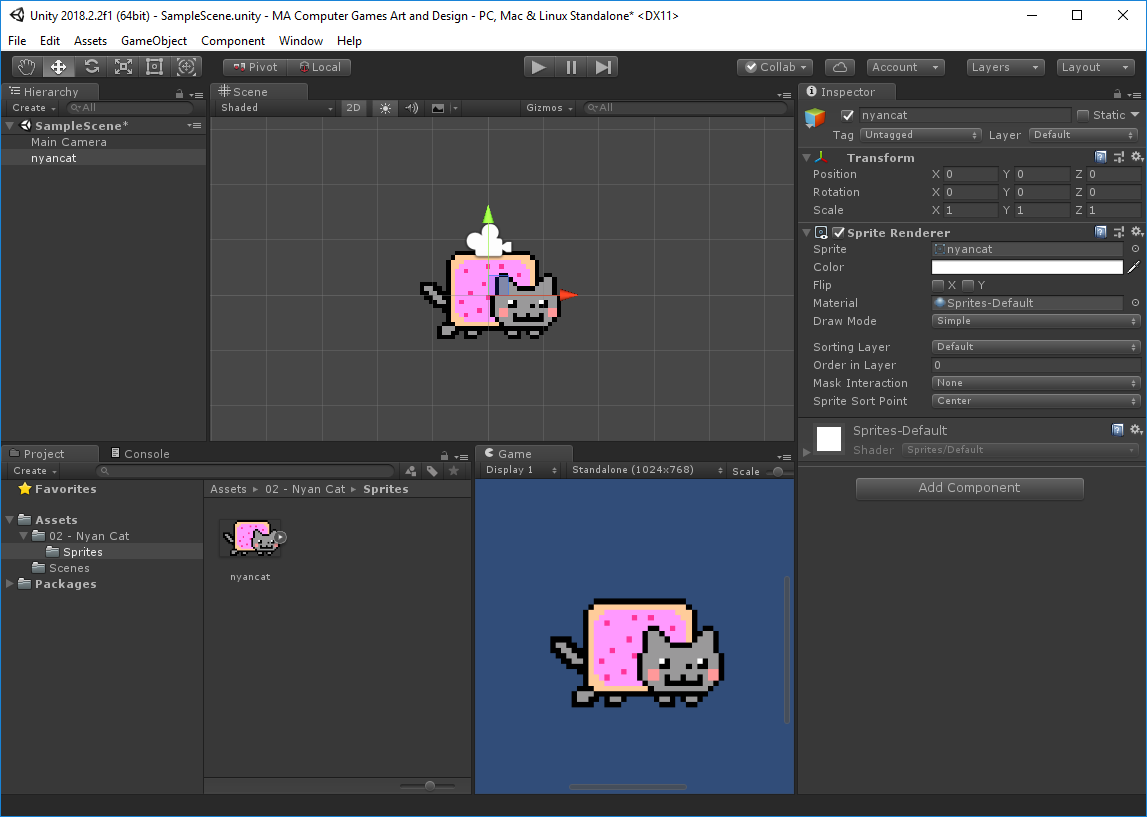
⚠ If the image appears to be blurred, make sure **Filter**is set to **Point**.

### 3. Using a Sprite

Now that the image has been imported as a sprite, it is possible to use it inside a game.

#### **Using a Sprite**

Locate the sprite from the **Project/Asset** window. Drag it onto the **Scene**window. This will import the sprite as a **game object**.



Find the newly created game object on the **Hierarchy**window (left) and click on it. This will select it and show all of its properties on the **Inspector** window (right).

You will find:

* **Transform**  
  Stores the position, scale and rotation of the sprite
* **Sprite Renderer**  
  Allows controlling the sprite properties (colour, sorting order when multiple sprites are overlapping, ...)

⚠ When a game object is imported by *drag and drop*, its position might not be where you want it to be. Make sure its position is correct. In the example above, the game object has been manually positioned at (0, 0, 0).

### 4. Moving a Game Object

Moving a game object in a scene means changing its **position**, which is stored in its **Transform** component. It is possible to access it using a **C# script**.

#### **Creating a Script**

Scripts can be created by right-clicking in the **Project/Assets** window and selecting the option **Create>C# Script**. At the moment of its creation, you need to find a name. We can call this script *Move*.

⚠ Virtually all assets in Unity can be renamed in the editor, without any consequence. This is not the case for scripts. The name of the file (for instance, *Script.cs*) must be used as the name of the class (for instance,*class Script : MonoBehaviour*).

⚠ The name given to the script file appears in the C# code. This poses strong restrictions on what names you can use. No spaces are allowed. The **camel case** naming convention ([Wikipedia](https://en.wikipedia.org/wiki/Camel_case)) is highly encouraged.

Once the script is created, it might take a few seconds for Unity to *compile* it. When that is done, you can drag it in the Inspector to add it as a *component* of your game object.

#### **Writing Your First Script**

When a new script is created, Unity initialises it with some basic code.

using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
public class Move : MonoBehaviour  
{  
    // Use this for initialization  
    void Start()  
    {  
    }  
    // Update is called once per frame  
    void Update()  
    {  
    }  
}

The position is stored in the transform as a set of three numbers. Unity calls this data structure **Vector3**.

To change the position when the game starts, we can replace the current Vector3 with a new one that we create. The code below is written in **Start**, so that it is executed the first time the game starts. **transfom.position** accesses the position vector of the **Transform**component.

    void Start()  
    {  
        transform.position = new Vector3(5, 0, 0);  
    }

The new **Vector3** that is created has three values: 5, 0 and 0 which indicates that the new position of the object is (5, 0, 0).

💡 Writing **new Vector3(5, 0, 0)** represents a vector with three elements **(5, 0, 0)**. That same vector can also be the results of the product between the vector (1, 0, 0) and the scalar 5. So you can also write that as:

transform.position = **new Vector3(1, 0, 0) \* 5f**;

💡 Unity contains already a reference to **(1, 0, 0)**, which is called **Vector3.right**. So:

transform.position = Vector3.right \* 5f;

which can be read as "*move 5 units to the right*".

💡 Multiple movements can also be added together. The following statements are all functionally equivalent:

transform.position = new Vector3(5,4,3);

transform.position = new Vector3(5,0,0) + new Vector3(0,4,0) + new Vector3(0,0,3);

transform.position = new Vector3(1,0,0) \* 5f + new Vector3(0,1,0) \* 4f + new Vector3(0,0,1) \* 3f;

transform.position = **Vector3.right** \* 5f + **Vector3.up** \* 4f + **Vector3.forward** \* 3f;

#### **Moving Continuously**

If you put that line of code in **Update**, instead of **Start**, the results will apparently be the same. This is because, each frame, the position is set back to (5, 0, 0), giving the illusion the object is not moving at all.

To move it continuously, we need to *add* a small offset to the current position. We can do this using the operator **+=**.

    void Update()  
    {  
 transform.position += new Vector3(1, 0, 0);  
    }

While **=** sets the position to a specific value, += adds the new **Vector3** to the existing value.

💡 That is equivalent to:

transform.position = transform.position + new Vector3(1, 0, 0);

Running the game, however, will cause the sprite to move right at an incredible speed. This is because **Update** is executed each frame; if the game runs at 30 frames per second, the sprite will move 30 units per second. That is equivalent to a speed of 108 Km/h.

The real problem is that the speed of the sprite depends on how many frames per second the game is running at. If the game runs at 60 fps, for instance, the movement speed is doubled. In this case, is said that the gameplay is **framerate dependent**.

#### **Framerate Indendepence**

If your game is framerate dependent, that will be marked as an issue. Luckily, Unity provides an easy way to move independently of the framerate. This is done using **Time.deltaTime**.

    void Update()  
    {  
 transform.position += new Vector3(1, 0, 0) **\* Time.deltaTime**;  
    }

Multiplying the Vector3 by **Time.deltaTime** changes the behaviour from "*moving 1 unit per frame*" to "*moving 1 meter per second*", which is now an acceptable speed.

⚠ If you are moving an object manually, don't forget to use **Time.deltaTime** for continuous movements. However, be aware that some other ways of moving objects (such as the one involving *physics*and **rigid bodies**) might not always require **Time.deltaTime**. Check the documentation to see when you need it and when you do not.

### 5. Reading Inputs From Keyboard

In the previous chapter, we saw how to move an object continuously. We can now extend that concept introducing keyboard inputs.

#### **Reading Key Events**

Most of the input functionalities in Unity are accessible using the **Input**class ([Unity Scripting API](https://docs.unity3d.com/ScriptReference/Input.html)). The following code is used to print a message when a specific key is being pressed:

if (**Input.GetKey**(KeyCode.LeftArrow))

Debug.Log("Left arrow being pressed!");

💡 There are several methods contained in **Input**that can be useful to read data from the keyboard:

* + **GetKey**: used to detect when a key is being pressed;
  + **GetKeyDown**: used to detect the frame in which a key has been pressed down;
  + **GetKeyUp**: used to detect the frame in which a key has been released.

💡 You can find a full list of keys accepted by**Input.GetKey** on the **KeyCode**class ([Unity Scripting API](https://docs.unity3d.com/ScriptReference/KeyCode.html)).

We can use Input.GetKey to make sure the position is changed only when the relative key is being pressed:

if (**Input.GetKey**(KeyCode.LeftArrow))

transform.position += Vector3.left \* Time.deltaTime;

which causes the object to move to the left by *one unit per second*, while the left arrow is being pressed on the keyboard.

#### **Reading Input Axes**

Explicitly writing which key you want to intercept can put constraints on your code, especially if you are developing on different platforms. Unity allows to *abstract* the concept of keyboard input with **input axes**. In most games, the player can move up/down and left/right; Unity has defined two axes: **Vertical**and **Horizontal**, respectively.

The method **Input.GetAxis** is used to query an axis. It usually returns a value, from -1 to +1.

float **y** = **Input.GetAxis**("Vertical");

For instance, **y** has a value of +1 when the player is pressing either the up arrow, or W. But it also responds to *joysticks*and *analog sticks* on a controller moving up. All controls that are traditionally associated with "going up" are mapped to an increase of the Vertical axis. The same applies to the Horizontal axis.

💡 Unity also applies some gentle smoothing to the input, so that the controls intermediate values are also possible to simulate an ease-in and ease-out effect.

The following code uses Input.GetAxis to change the position of an object based on the current vertical and horizontal axes:

float **x** = Input.GetAxis(**"Horizontal"**);

float **y** = Input.GetAxis(**"Vertical"**);

transform.position += **x** \* Vector3.right \* Time.deltaTime + **y** \* Vector3.up \* Time.deltaTime;