

# LP2B – Multimedia: Digital Representation

- LP2B Tutorial Class n°1 -

## Your First steps with Unity Engine

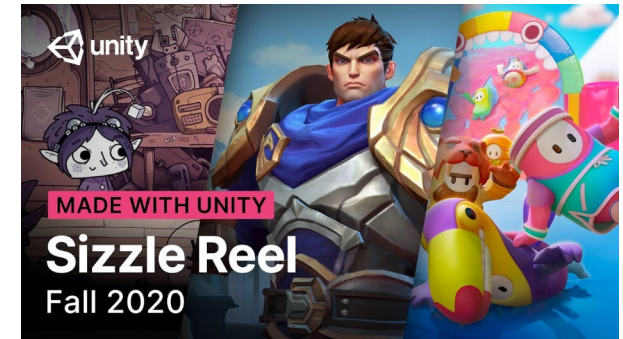
**GameObjects, Components and Assets**

This first class will give a quick overview of :

- *Installing Unity and setting up a License -*
- *Importing Images as Assets -*
- *Creating GameObjects to use the Assets -*
- *Notion of Component. Adding Components -*
- *Creating simple C# scripts -*
- *Basic features of the physics engine -*

# What is Unity Engine ?

- Unity started in 2005, as a 3D engine for video game developers  
That's why it is still sometimes called « Unity 3D », even though it's much more flexible now
- The engine has steadily gained popularity over the years  
Thanks mainly to an increasing number of successful video games being made with it
- Some very famous companies such as Blizzard or Ubisoft are now using it  
Hearthstone, Assassin's Creed Identity and many others are made with Unity Engine
- Thanks to this success, it started diversifying its features  
2D features, assisted shader creation, render pipelines, integrated animation rigging...
- Today, it has expanded beyond the world of video games  
An increasing number of companies are using it for industrial and social applications
- The engine is now at peak popularity and keeps growing  
Unity should remain popular in the coming years, making learning it a good investment !



A unity promotional video that gives a good idea of how powerful this engine can be :  
<https://www.youtube.com/watch?v=eciP7ixTNec>

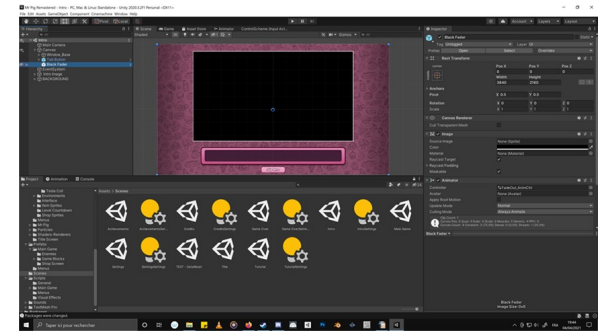
Genshin Impact  
©MiHoYo



Humankind  
©Amplitude Studio

# Why this Success ?

- Easy to use. Especially for “quick and dirty” prototyping and experimenting  
Testing the feasibility of the concept quickly is a key step for any product idea
- The « Personal » license is free and allows everyone to try it easily  
You can use it for free, forever, as long as you don't sell anything you made with it
- A “designer tool” with emphasis on GUI and accessibility, whenever possible  
People who lack formal coding education ( designers, artists... ) can easily get into it
- Can be used to develop applications for almost all devices, and port between them  
PC, Mac, Linux, and almost all of the game consoles and mobile devices on the market
- Very low “floor”, but very high “ceiling” : if you get good enough, you can go VERY far!  
With bigger budgets and teams, Unity can perfectly be used to create AAA applications



*Unity. An intuitive and efficient GUI...*

```
my_animator.SetBool("enabled", isEnabled);

if (_hasIntroAnimation)
{
    isBusy = true;
}
else
{
    isBusy = false;

    //Jump directly to either "enabled" or "disabled" animation state
    if (isEnabled)
    {
        my_animator.Play("Enabled");
    }
    else
    {
        my_animator.Play("Disabled");
    }
}
```

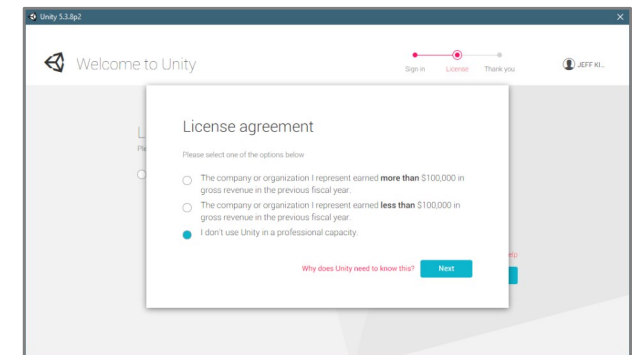
*...Combined with the power of C# code!*

# How can I get Unity ?

- Go to this address : <https://unity3d.com/get-unity/download>
- You should download « Unity hub », rather than Unity alone  
The hub will allow you to manage your Unity updates and your projects more easily
- Once you install the hub, you will have to create an account  
You just need a valid e-mail address. Make sure to write your password somewhere!
- When asked, choose « I don't use Unity in a professional capacity »  
That way, you will get the free 'personal' license, which you can use for as long as needed
- **DON'T FORGET TO UPDATE YOUR LICENSE WHEN NEEDED**  
The day you sell a product made with Unity, you **MUST** do so to avoid legal problems

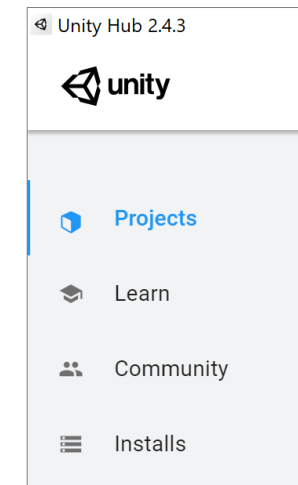
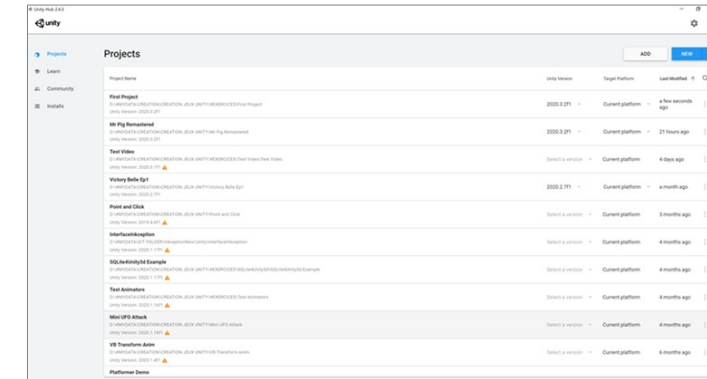
If you have trouble with the installing/licensing process :

Use this video as reference : <https://www.youtube.com/watch?v=cEzCcncu-ol>



# Using Unity Hub

- Unity Hub was a gadget. But now it is a mandatory part of using Unity  
Launching Unity Editor from outside the hub will redirect you to there.
- Unity Hub is divided into different tabs, which you can see on the left  
Projects, Learn, Community, and Installs. On launch, Project tab is shown first
- The 'learn' and 'community' tabs provide links to useful resources  
Official unity tutorials and forums where you can ask for help respectively
- The 'installs' tab is where you can install a new version of Unity  
As many versions as you want can coexist on the same machine
- Go to the « installs » tab now. We need to install a version of Unity editor.  
We'll talk about 'projects' tab later, once we have a version to make a project with !



*Depending on the installation method you chose, you might already have a unity editor.*

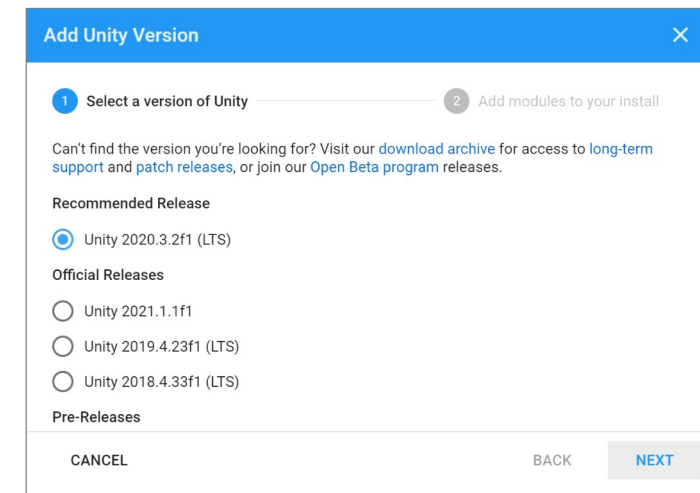
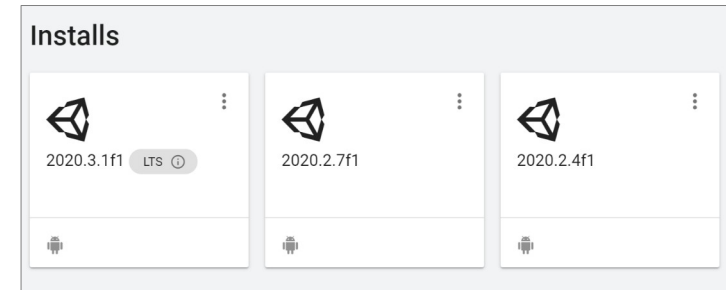
*But we will look into downloading a new one anyways.*

*Unity Hub gets its own updates. If a newer version is available, it will be written At the top of the window*



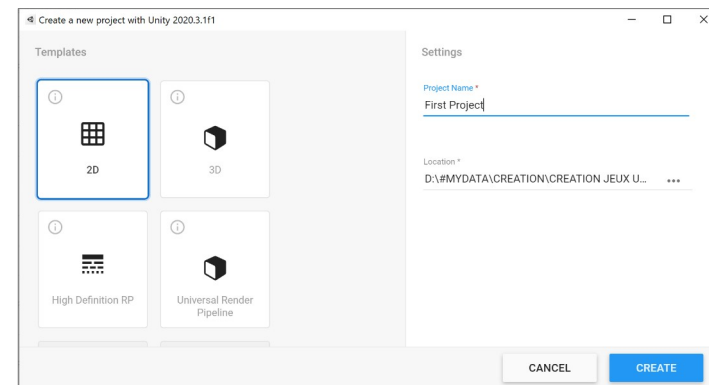
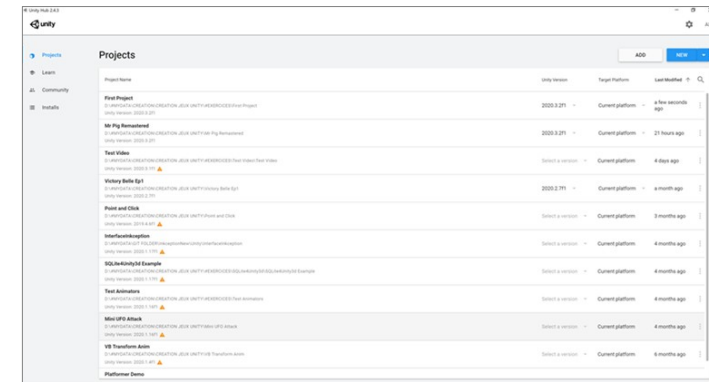
# Choosing your Unity version

- The 'Installs' tab shows the versions you have installed on your machine  
If you just installed the hub alone, you might have none available yet
- Click the « add » button to select a version to install  
This will open a sub-window which lists recommended, official, and alpha versions
- Pick the recommended version. It is the latest 'LTS' version  
LTS versions are stable and bug-tested. The latest is at most one year old
- In 'official releases', there is always a stable version more recent than LTS  
Using this one is usually safe enough, but for this class, please use 2020.3 LTS
- 'Pre-releases' are alpha. They're to use at your own risk !  
Usually not worth it until you really NEED to get access to a newest feature right now



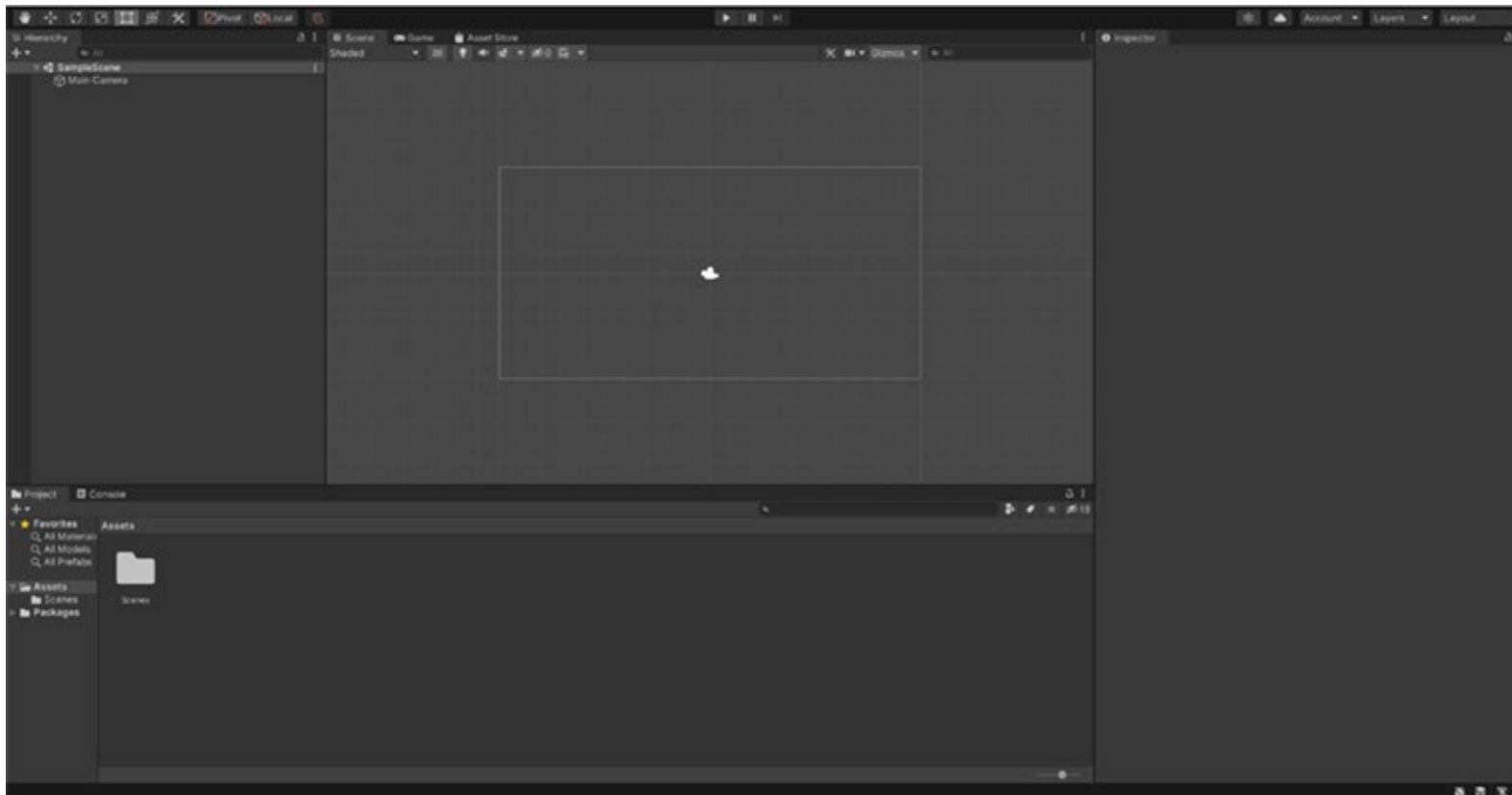
# Creating your First Project

- In 'projects' tab, you can see all projects active on the machine  
They were either created on the machine, or added to the list by importing unity folders
- To create a project, click “new”. You can then choose a template  
If you click the arrow part, you can pick what unity version to use (amongst available)
- For this first Unity class, please pick the “2D” template  
Templates are not 'final', but changing later can prove difficult. Choose correctly
- “Add” doesn't create a project. It's for importing one!  
Use this when you copy-paste a project from another machine. This will add it to the list
- Because folders use the project name, projects can not be renamed  
OK. Technically, they can. But this is a chore. Please pick a name that makes sense
- Once you're done with setup, click “create” and Unity editor will launch  
A folder with the same name as your project will be created, and filled with initial data



# Welcome to Unity Editor!

- Here is, roughly, what you should see on your screen :



*This is the Unity Editor.  
The place where things get done!*

*This interface can be fully  
customized by moving the tabs  
and windows.*

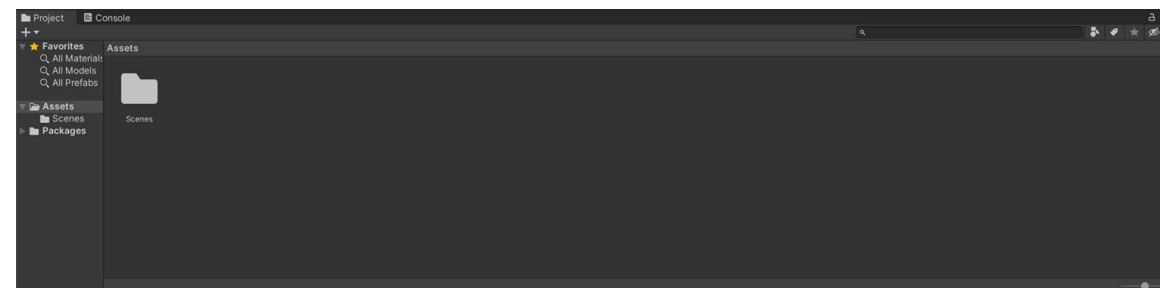
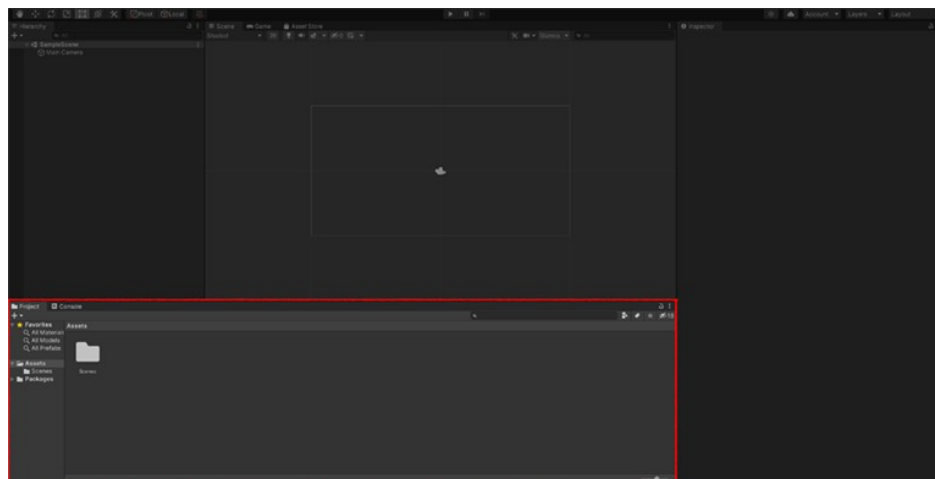
*However, as a newcomer, you  
should keep the default settings  
for a while until you know what's  
what*

*Once you get used to the tool,  
feel free to customize to your  
heart's content!*



# The Project Tab (asset browser)

- All the files and resources (= assets) ready to be used are visible here  
Assets are not just multimedia content : even code is considered as an asset !
- Unity can handle a lot of formats natively. Images, sounds, videos...  
If a format is not recognized properly, there probably is a package somewhere to solve that !



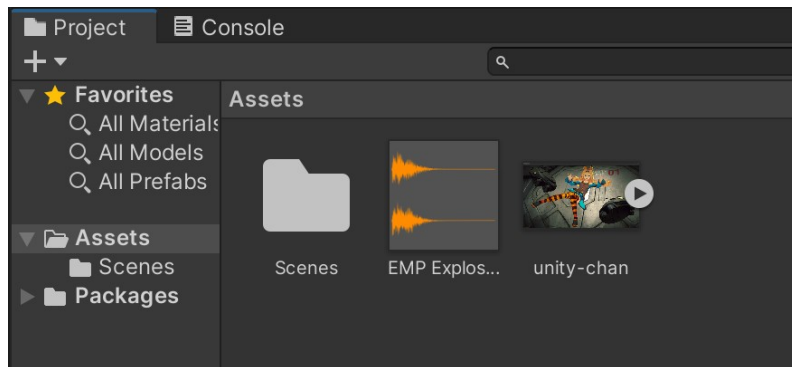
*Of course, on a new project, the “assets” folder is mostly empty. It should only contain a “scenes” folder which stores the empty scene you can see above.*

*Note that there also is a “packages” folder, but you should not add content to that one manually. Let Unity and the “package manager” handle it.*

- Most formats you have seen in class will work natively... except vector graphics !  
Vector graphics can be used with an official package, but it is a beta package, and is fairly tricky to use

# Importing an Asset

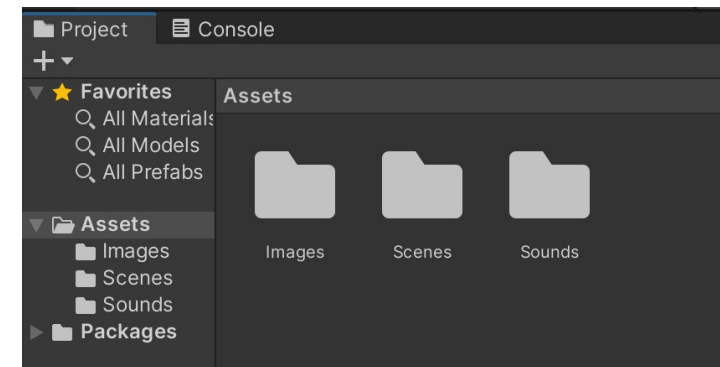
- First option : right click in the asset tab, and choose « import new asset »  
This will open a file browser, allowing you to choose file(s) to import in your disk
- Second option : drag and drop files in the « Assets » tab  
This is usually the fastest and most convenient. But some people prefer the other ways
- Third option : copy-paste data into the asset folder directly  
(Right click => «show in explorer») will allow you to see the asset folder on the hard drive
- You can also create sub-folders to organize assets  
(Right click => Create => Folder). Very important to do as your projects get bigger !



*Imports successful! Good!*

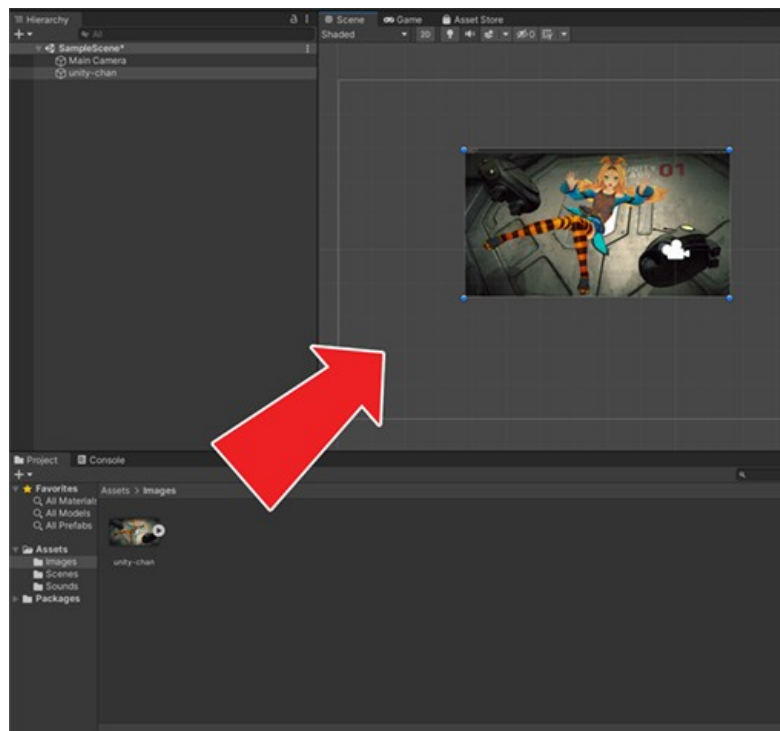


*Creating some sub-folders and  
dragging stuff inside...*



*Imports organized! Even Better!*

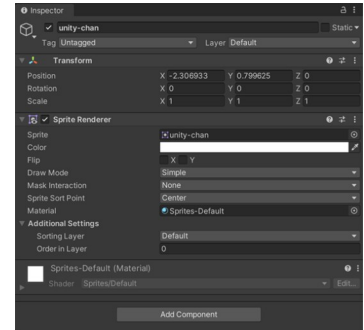
# Putting an Asset in the Scene



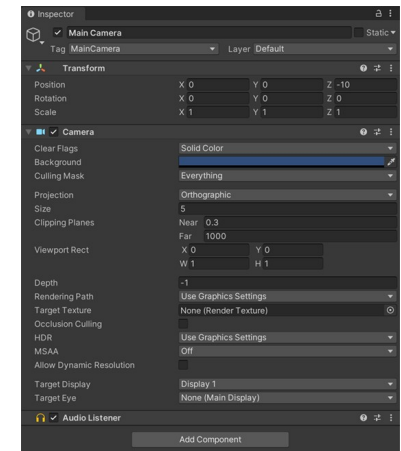
- We'll start with an image, as it is the easiest to understand  
Make sure you have an image asset ready into your asset folder
- Simply drag-and-drop the image into the scene tab  
By default, the scene tab is opened and at the center of the screen
- Your image is now on the scene. You can then alter its properties  
The properties of the selected entity appear on the 'inspector' tab
- Move your image and notice how the “position” property updates  
That property can be seen in the “transform” component in the inspector
- Notice that the image has now appeared in the list on the left  
This is a list of all the entities in your scene. Which are called **GameObjects**

# GameObjects + Assets = Unity!

- **GameObjects are the core of the Unity engine logic**  
They use their attached « components » to perform actions and/or use our assets !
- **Our image on the scene ? A GameObject with a “Sprite renderer” !**  
Sprite Renderer is a component used to display sprites. ...Such as our image asset !
- **All GameObjects have a 'Transform' component attached**  
Transform stores position, rotation, and scale of the GameObject
- **The Camera is also a GameObject. So it also has a Transform**  
Note that even if a GameObject has no visual element, it still has a transform
- **'Transform' also allows GameObject to have 'children'**  
If the parent's transform changes, it will also affect the children
- **Images, Sounds, videos... it's all done with GameObjects !**  
Just add the right components. Or let Unity do it by drag-and-dropping !



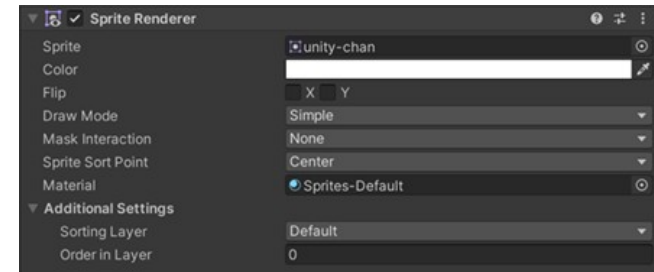
*GameObject + Sprite renderer = Image!*



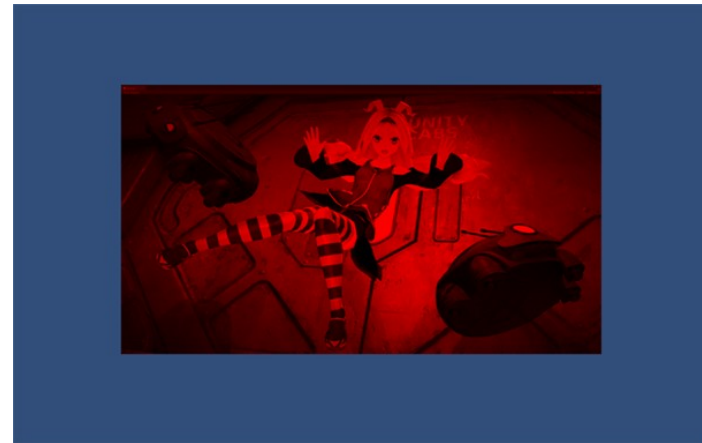
*GameObject + Camera component = a Camera!*

# The SpriteRenderer Component

- SpriteRenderer is a component which will display a Sprite Image  
It interprets the data of the asset to color the pixels on the screen, displaying the image
- The component has a bunch of attributes which will alter its behavior  
Try playing with those settings and see what happens. **You can cancel any action with CTRL+Z**
- The ones you should really understand now are : flip and Color  
Flip X and Flip Y is pretty obvious, and color's effect is pretty clear too
- Color operates a **multiply** operation on the color channels :
  - Any color is the combination of a red, green, and blue value (RGB)
  - White being (R=1 G=1 B=1), it multiplies all values by 1 : the sprite is unchanged!
  - Black being (R=0 G=0 B=0), it multiplies all values by 0 : the sprite becomes black!
  - Pure red being (R=1 G=0 B=0), the red values of the sprite will be unchanged, but the others will be set to zero. We then "only see the red"!
  - This attribute is often used to allow the customization of color on greyscale sprites



The Sprite Renderer Component as it appears in the inspector  
(you must select your sprite to see it there)

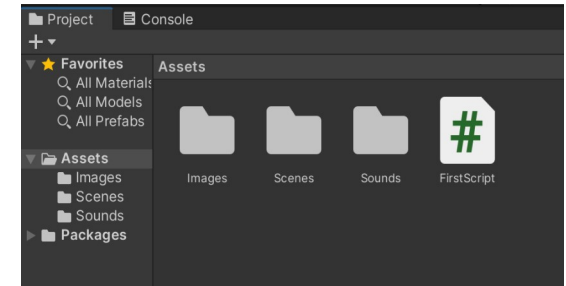


The "unity-chan" sprite with the color set to pure red.  
With this setting, we only see the values of each pixel for red.  
We can also say that we see the red channel of the image

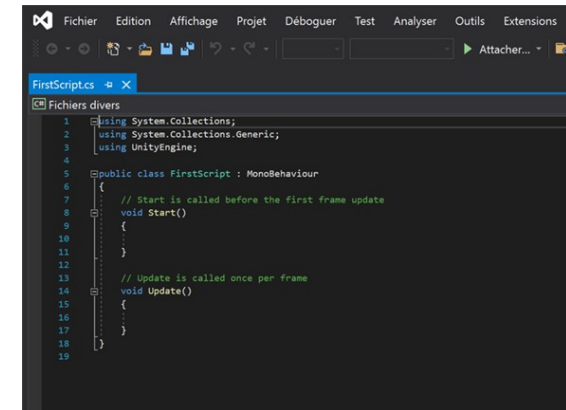


# Scripts : our Custom Components!

- Code is written in scripts. Scripts are then attached to game objects  
Code can be seen as 'custom components' that we can write to perform the actions we want
- Create a script : *(right click into the asset panel => Create => C# Script)*  
This will create a default script template suited for Unity. You have to name it right away
- It is best to give your script a name that makes sense, immediately  
Changing name later would require the extra step of changing the class name inside too
- For this example, please name the script « FirstScript »
- Double click your script and it will be opened in Visual Studio  
Visual studio is a free development tool by Microsoft, it is now provided with Unity



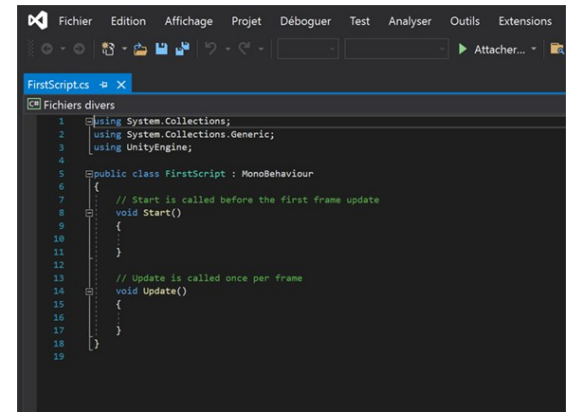
Our Script in the Project Tab



Our Script opened in Visual Studio

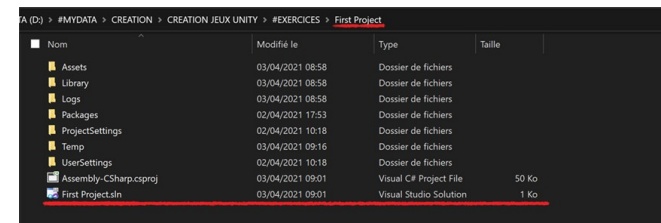
# Visual Studio and C# : first contact

- To use Visual studio, you need a Microsoft account  
If you don't have one already, you can create it during the visual studio first launch
- Just like Unity, creating a free account is the only barrier to entry  
Other C# tools could be used to code for Unity. But Visual Studio has some direct integration
- Once the registration part is done, You will be able to see your script  
It is not empty, as Unity has created a template for us with some basic guidelines
- The script already has the proper imports to allow Unity coding  
Unity has other packages that are less common and that you can import later if needed
- A Visual Studio project file was also generated in the project Folder  
In the future, you should launch visual studio by opening this file. For some reason, code auto-completion might not work properly if you launch Visual Studio from Unity like we did (hopefully they solve this problem soon)



*If MonoBehaviour appears white, it means the project has not properly updated dependencies.*

*The code will work, but you will miss on very, very convenient auto-completion features.*



*Opening the visual studio project through the \*.sln file in your project folder solves this issue.*

*Note that this problem might not happen on all systems.*

# MonoBehaviour : the “magic” class

- You can see that your Class « FirstScript » has been automatically setup to inherit from a class named « MonoBehaviour »
- MonoBehaviour is a Unity class that allows our script to be seen by the engine as a “Component”. Yep. Just like “Sprite renderer” or “Camera”!
- This will allow us to attach it to our GameObjects later.
- MonoBehaviour also come with a lot of nice features, such as predefined methods and events that make our life much easier!
- You can see that 2 of those methods have been automatically added to our script : “Start” and “Update”.

```
Script Unity | 0 références  
public class FirstScript : MonoBehaviour
```

*This part here means that the class “FirstScript”  
Inherits from “MonoBehaviour”  
( MonoBehaviour is now green because I solved  
the issue mentioned in previous slide )*

```
FirstScript.cs  
Assembly-CSharp  
1 using System.Collections;  
2 using System.Collections.Generic;  
3 using UnityEngine;  
4  
5 Script Unity | 0 références  
6 public class FirstScript : MonoBehaviour  
7 {  
8     // Start is called before the first frame update  
9     Message Unity | 0 références  
10    void Start()  
11    {  
12    }  
13  
14    // Update is called once per frame  
15    Message Unity | 0 références  
16    void Update()  
17    {  
18    }  
19 }
```

*The “Start” and “Update” Methods may be empty,  
But they're here. Ready to be customized!*

# Let's Write our first Line of Code !

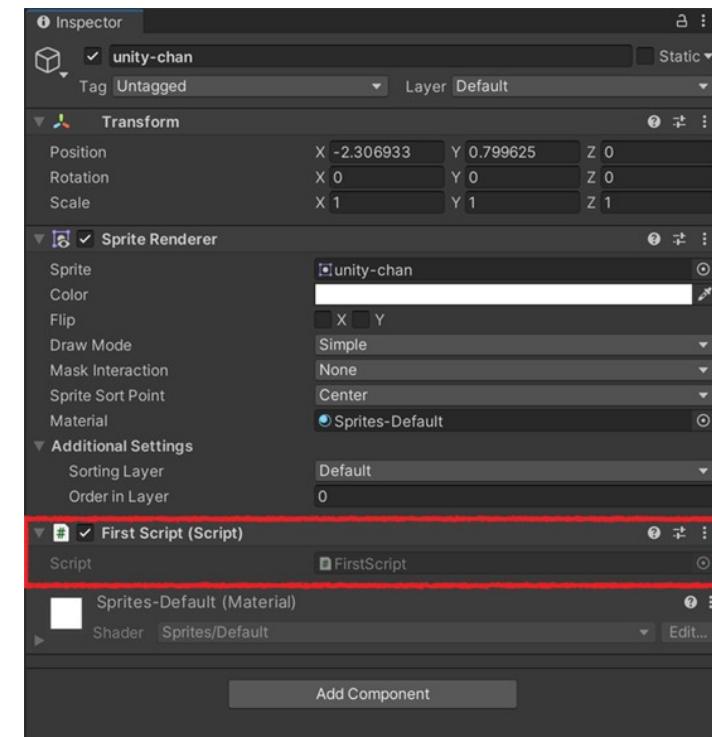
- The « Update » method of a MonoBehaviour is called once per frame  
A « frame » is an individual image displayed on screen. Usually at the rhythm of 60 per second on computers
- Code runs between frames, allowing us to create movement  
Same principle as the cinema or any video. But here, we can control and modify the movements on the fly
- To see this in action, add the following line of code inside "Update" :

```
void Update()  
{  
    transform.Translate(0.01f, 0, 0);  
}
```

This code accesses the "transform", and then calls the "Translate" method available for it.  
As parameters, we have 3 numbers : one for x, one for y, one for z.  
Can you guess what will happen ?

# Attaching the Script... and Testing it!

- Our Script won't do anything if we don't attach it to a GameObject  
Remember : our script is a MonoBehaviour. And MonoBehaviours are components
- Let's attach our script to the image we put on the Scene Earlier!  
Make sure the image is selected first : click it on the scene or click its name in the object list
- You have Two ways to attach your script to the Image GameObject :
  - 1) Click the “add Component” button. Your script can be found in the “Scripts” subcategory. Click it and it will be added. *(you can also use the search field)*
  - 2) Drag and Drop the Script from the “Projects” tab, into the “Inspector” tab on the right. Where you can see the components. It will add your script
- Now, click the “play” button above the scene, and see what happens!  
Make sure the image is within the bounds of the camera (white rectangle), or you won't see

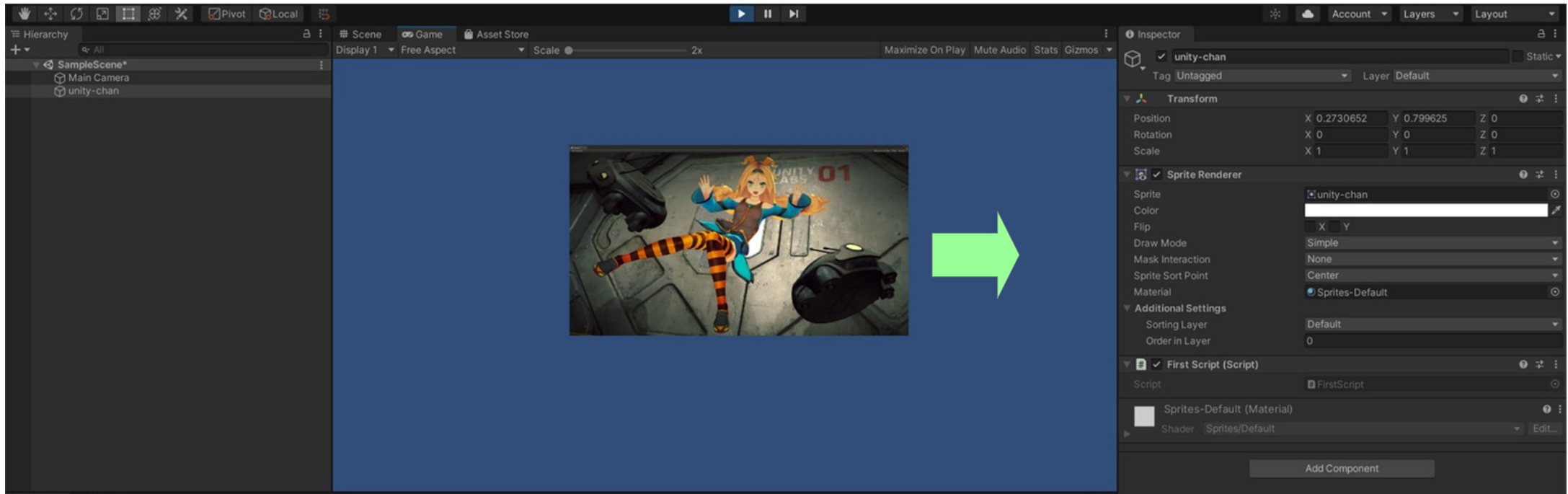


*“FirstScript” is now visible in the inspector, as a component : proof it is now successfully attached to my GameObject!*



# Tadah! It works!

If you guessed before that the sprite would be translating to the right over time, you were correct!



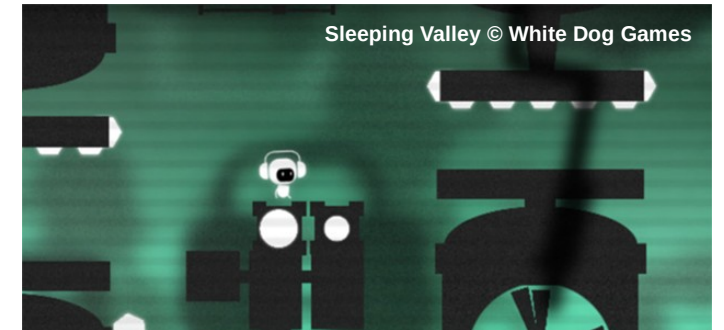
Of course, Unity has a lot more to offer than this. But the core principles you discovered with this simple example will always remain true. GameObjects, components, and assets : they are what makes Unity “tick” !

# The Problem with our Line of Code

- Even though it was a good first step towards understanding Unity, our line of code actually has a big problem...
- The code moves the image right of 0.01 units to the right each frame. But what would happen if the frame-rate changed?
- ...That's right. Our image would move at a different speed than intended!
- Our current code is **frame dependant**, and that's very, VERY **BAD**!
- Whenever a behaviour happens over several frames, it should always be in **real time**, rather than **frame dependant**
- Leaving frame dependent code in an app is the perfect way to generate bugs, unintended behaviors, and make the life of your user miserable!  
**AVOID IT AT ALL COSTS!**
- So... How can we fix this?

```
void Update()  
{  
    transform.Translate(0.01f, 0, 0);  
}
```

*Our Update function. This will run once each frame.  
This means the image will translate by 0.01 EACH FRAME,  
and not EACH X SECONDS, like we would like it!*



*Believe it or not, some indie developers on Steam  
still make the mistake of leaving frame-dependent code  
in their games : run them on a powerful machine and everything  
will go much faster than intended!*

*This mistake alone is often enough to make the app or game  
Completely unusable, and ruin the experience!*

# The Time Class : our ticket to real time!

- Fortunately, the solution is actually pretty easy : we are going to use the Time Class!

"Time" is a pre-existing class of Unity. It is a static class, so it can be accessed from anywhere!

- Time.deltaTime gives us the time that elapsed since the last frame, in seconds

Because there are so many frames in a second, the number is smaller than 0. Usually about 0.016 at 60 FPS

- We can use this to update our line of code in the following way :

Make sure you save your script changes in visual studio before returning to unity, or they'll be ignored!

```
Message Unity | 0 références  
void Update()  
{  
    transform.Translate( 1.5f * Time.deltaTime , 0, 0 );  
}
```

- This will make our image move 1.5 units to the right **PER SECOND**. We are now **REAL TIME!**

Feel free to mess around with the 1.5 value to see the results. ...what about trying a negative number?

# Basic Input Reading

- Let's improve our code so that the image only moves right when we push the right arrow  
To do that, we are going to use the "Input" class, which will return the status of a key as a boolean
- Once you have copied and understood this code, add another "if" for moving to the left  
The Key Code for the left arrow is "KeyCode.LeftArrow". But you also need to change something else...

## More details on that Code :

"Input" is the basic Unity Input class.

Nowadays, there is an "Input system" package, which is much more powerful and is universal for all devices!

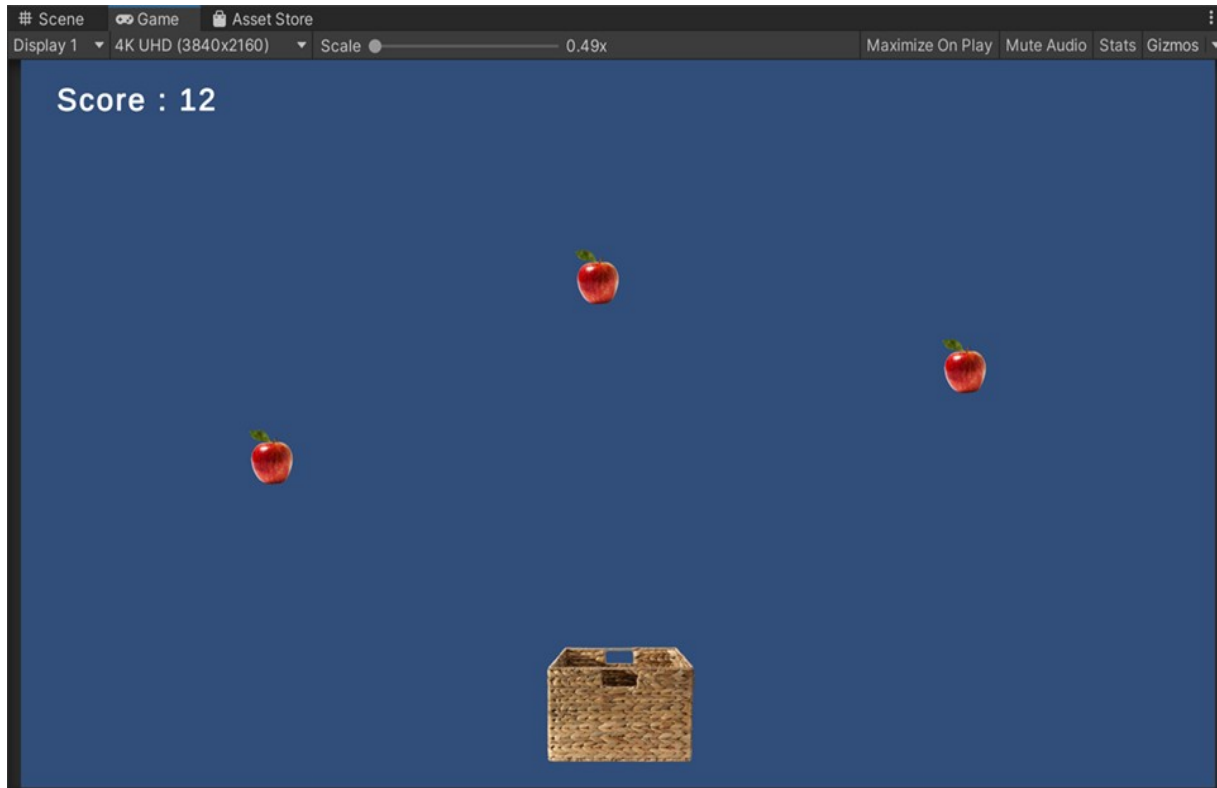
...But because this system is much more complex, we'll stick to the default basic package for this time.

"Input.GetKey" returns true if the key matching the KeyCode provided as parameter is currently pressed.

We input those codes using the "KeyCode" class, which is very easy to use if auto-completion is available : just type "KeyCode." and the system will suggest you all the options!

```
// Update is called once per frame
Message Unity | 0 références
void Update()
{
    if ( Input.GetKey(KeyCode.RightArrow) )
    {
        transform.Translate(1.5f * Time.deltaTime, 0, 0);
    }
}
```

# Let's make a simple mini-game!



The Rules : Apples fall randomly from the sky, and the player has to catch them by moving a basket with the arrow keys. 1 apple caught = 1 point

As basic as it may seem, you still have to learn the following things in order to do it :

- *Adding attributes (variables) to our script*
- *Using the physics engine to detect collisions*
- *Record a game object as a “prefab” so copies of it can be created later*
- *Basic Random Number Generation*
- *Creating a text and updating its contents*



# Attributes and Variables

- In C#, variables must be defined before use  
The usual types of C programming are of course available : int, float, string...
- Variables are local to the method you define them in  
If a variable is defined in "Update", for instance, it will be unavailable in other methods
- Attributes are variables associated with the whole class/script  
They can be used in all methods, and persist as long as the object carrying the script exists
- Attributes require an extra keyword to define the level of access  
3 levels of access exist : public, private, and protected. We'll explain later what they mean
- Score needs to persist between frames, so it is an attribute  
Attributes should be defined inside the class, but before the first method. For clarity.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class FirstScript : MonoBehaviour
{
    protected int score = 0;

    // Start is called before the first frame u
    void Start()
    {
    }

    // Update is called once per frame
    void Update()
    {
        int localint = 0;

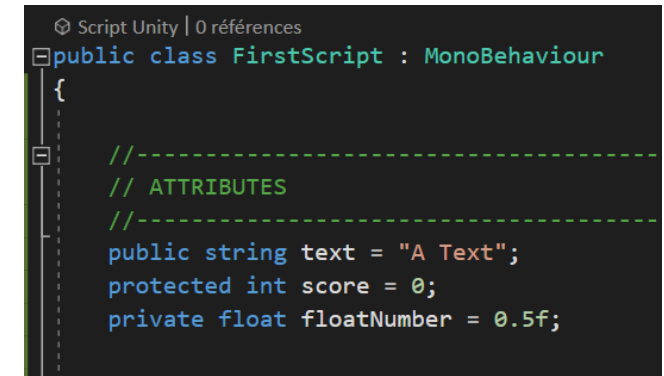
        if (Input.GetKey(KeyCode.LeftArrow))
        {
        }
    }
}
```

*The integer variable above is an attribute.*

*The one below is a local variable that is only accessible from within the Update() method*

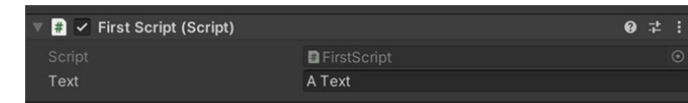
# Access Level of Attributes

- **Access levels are an important part of Object-oriented programming**  
They allow the code to be more clear, better organized, and helps auto-completion work
- **“Private”** : only methods within the class can access this attribute  
This attribute will not even be usable by classes which inherit from your class
- **“Protected”** : only methods within this class and its inheritors can access  
If you plan to use inheritance in your code (you should), this is very useful
- **“Public”** : ALL methods, even on foreign objects, can access the attribute  
However, remember that you will still need a reference to an instance of the class to do so!
- **In Unity, “Public” also makes the attribute visible in the editor!**  
Very useful to provide scripts with data and references to other objects, and for testing!



```
Script Unity | 0 références
public class FirstScript : MonoBehaviour
{
    //-----
    // ATTRIBUTES
    //-----
    public string text = "A Text";
    protected int score = 0;
    private float floatNumber = 0.5f;
}
```

*I put example attributes in my script with the different access levels...*

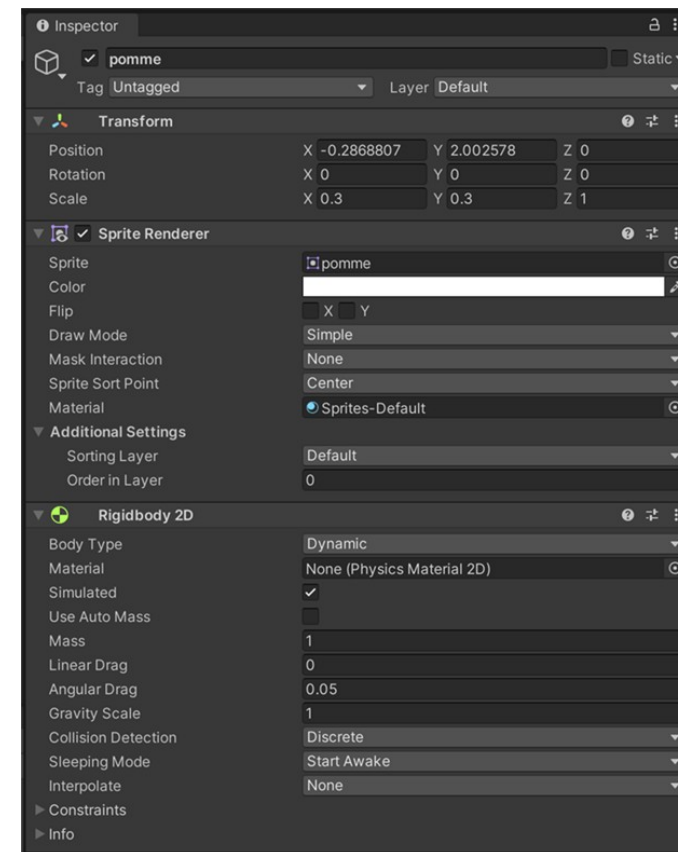


*If I save those changes and go back to Unity editor, here is what I can see in the inspector.*

*You see that I have a direct access to the “text” attribute, because it was defined as public. I can even modify its value from here!*

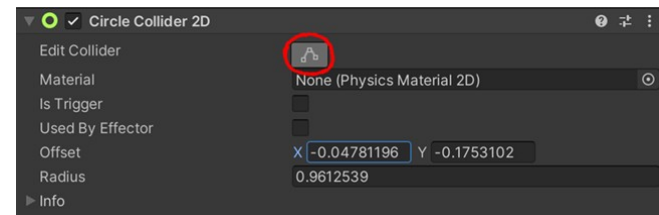
# Using Physics Engine Components

- We are going to create a falling apple ...without writing code!  
We are going to use the physics engine to add gravity to an object for us
- Import the provided image of an apple as a GameObject  
First, make the image available as an asset, then, drag and drop it to the scene
- Add a “Rigidbody2D” component to that apple GameObject  
Use “add component” and then look inside “Physics 2D”. Either that or do a search!
- If you test the game right now, the apple should be falling automatically!  
Rigidbody2D connects an item to the 2D physics engine, which also manages gravity
- “Gravity Scale” allows you to change gravity strength for this object  
It is often very useful to have different gravities on different objects. Your games do this!
- However, a Rigidbody2D alone isn't enough to handle collisions...  
To have collisions, you also need to add colliders. (which gamers know as “hitboxes”)

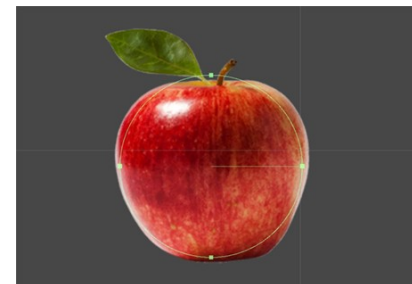


# Colliders and Collisions

- Add a "CircleCollider2D" component to your apple GameObject  
Unity will try to setup the collider automatically. But don't really count on it doing it well
- Edit the collider so it matches the red part of the apple closely  
Either edit the "gizmo" visible over the sprite, or edit the number values directly
- Repeat the whole process with the basket image, with "BoxCollider2D"  
Create GameObject from image. Add Rigidbody. Add Box Collider. Edit collider to match
- We do not want the basket to fall : change its gravity scale to 0  
With no gravity affecting the basket, it should remain at the same height, as we want it!
- Place the basket under the apple and launch the game  
The apple should fall on the basket, and then stay there as if it were on ground, right?
- Do we get the expected behavior? Can you guess why?  
Hint : a physics engine is all about calculating forces, both from gravity and impacts



Click this button (circled here in red) to edit the collider



CircleCollider2D setup on the apple GameObject

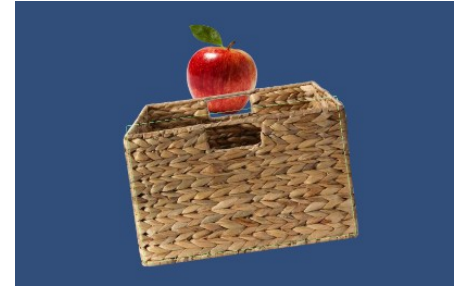


BoxCollider2D setup on the basket GameObject

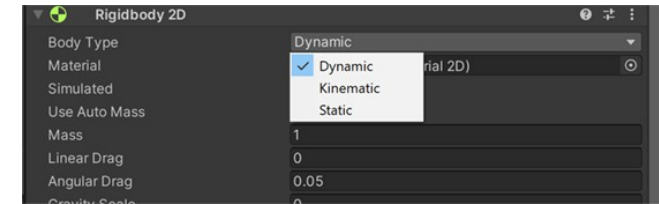


# The Types of RigidBody

- The impact of the apple with the basket moves the basket!  
The physics engine calculates a force from the impact, and applying it to the basket moves it!
- This problem is solved by changing the type of the basket's RigidBody2D  
The default type is “dynamic”. Which creates and applies forces from gravity AND impacts
- Change the basket's RigidBody type to “Kinematic”  
In this mode, no forces are generated. The RigidBody will only move from code instructions
- Test again : we now have the expected behavior!  
The next step is to make the apple “disappear” when it collides the basket
- The 3rd type, “static”, is optimized for Rigidbodies that do not move  
Examples : walls and static obstacles. Use “static” whenever possible for performance
- Now, add a copy of “FirstScript” to the basket GameObject!  
You can now move the basket with the arrow keys. We're getting closer to our minigame!



*Hey! That's not what we wanted!*



*The 3 choices of RigidBody type as they appear in Unity*

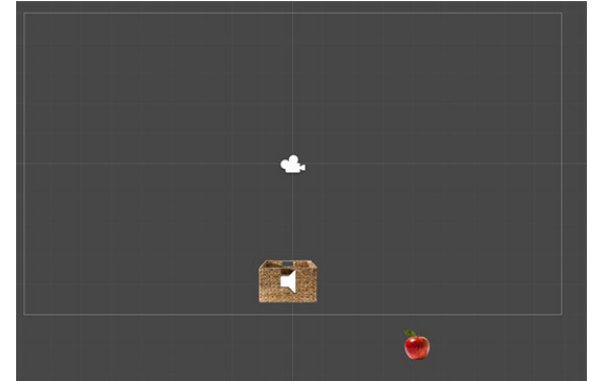


*With Kinematic mode : problem solved!*



# Removing Apples that fall too low

- Let the apple fall on purpose. Does it disappear once it exits the screen?  
No. It doesn't. You can see that by watching the hierarchy tab on the left
- The limits of the screen are NOT the limits of the "world"  
The limits of the screen depends on the camera. The camera only sees a very small area.
- In the future, we want to spawn many apples, so we have to solve this  
Otherwise, the number of apples in the game will grow over time, hurting performance!
- To do that, we have to create another script and attach it to our Apple!  
Please give the script a name that makes sense. Like for instance "Apple\_script"



*You can see here that the apple is outside the white border of the camera.  
...And yet it exists!*

## Now, it's your turn!

- Create an "if" statement that will test the position of the Apple on the y axis, and do so every frame
- If the Apple goes below a threshold of -10, it self-destructs.
- You can request the destruction of the gameObject carrying a script with this line of code:

```
Destroy(gameObject);
```

# Reacting to Collisions

- Now, we want the basket to react to the collision by gaining score

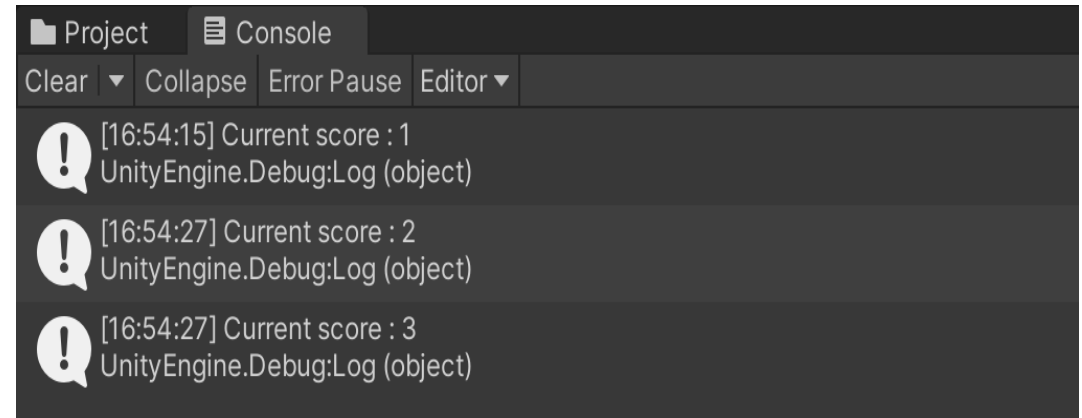
We will also talk about some debugging features, since we have no text on the scene to see that score

- Add a new method to “FirstScript” : `OnCollisionEnter2D`

It is a predefined method that reacts to a collision event. Make sure to copy the method definition perfectly!

```
//React to a collision (collision start)
Message Unity | 0 références
void OnCollisionEnter2D( Collision2D col )
{
    score++;
    Debug.Log("Current score : " + score);
}
```

*The `OnCollisionEnter2D` method to implement in the script. The actions inside could be entirely different, but the definition (void return type + name + parameter ) has to be exactly like this*



*Example of the `Debug.Log` messages displaying in the “Console” tab. The console tab is, by default, right next to the project tab in the bottom part of the screen*

- `Debug.Log` allows us to display text messages in the console tab during testing

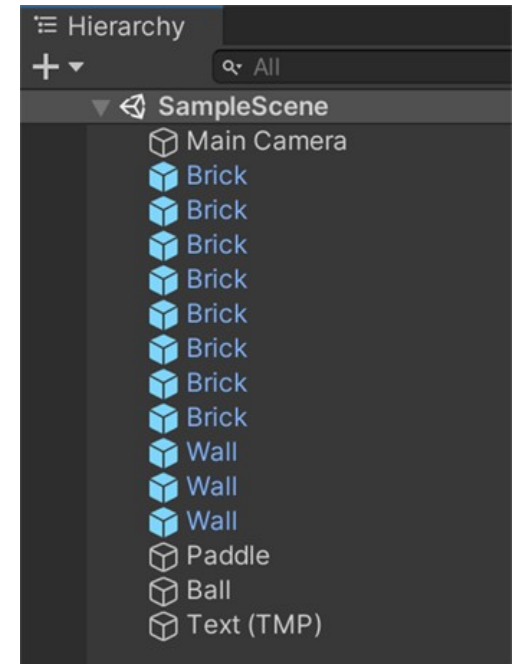
This is a very convenient way to see data and track code behavior without messing with our actual scene

# Identifying the Object collided

- OnCollisionEnter2D triggers whenever ANY collision with the object begins
- But often, we only want a behavior to trigger when colliding a specific object  
We do not want a collision with an enemy to do the same as one with an item, do we?
- There are several ways to do this, but the simplest is to use **GameObject Names**
- The names visible in the hierarchy tab are the names of the GameObjects  
You can rename them. The same way you would rename any entity in Unity : right click or F2
- The name is a string. You can give the same name to all entities of a given type.  
Since strings can be parsed, the names truly are a valid way to identify the collision!
- In OnCollisionEnter2D method, you get the name of the collided object with :

```
//React to a collision
@ Message Unity | 0 références
public void OnCollisionEnter2D(Collision2D col)
{
    Debug.Log("Name of collided object : " + col.gameObject.name);
}
```

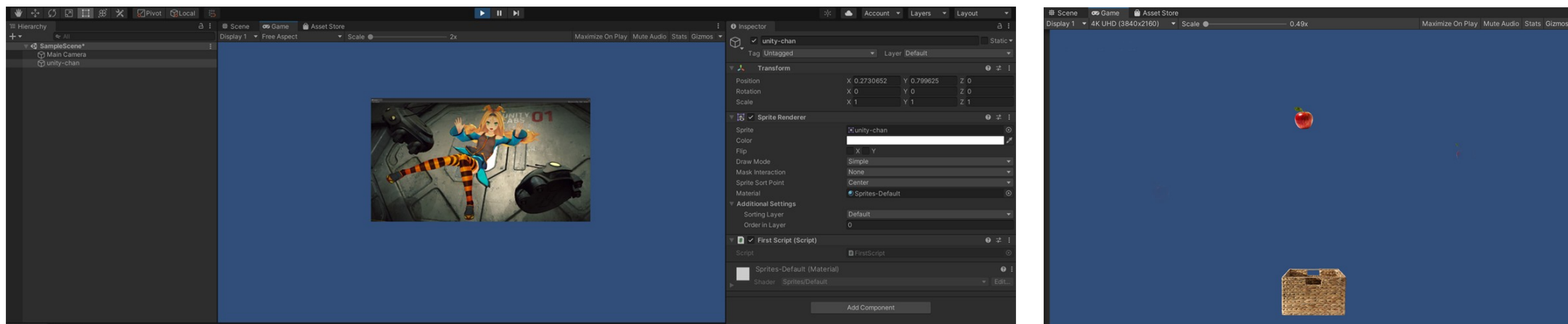
*We access the properties of the « col » parameter. It is of type Collision2D and contains all the info about the collision that occurred*



*As you can see on this screenshot, GameObjects can have the same name. It's not a problem !*

# Aaaand... We're out of time!

- Yes. I know. Our game isn't even playable yet. ...But we already learned a lot!
- Try to understand the “GameObject + Component + Asset” logic for next time  
It truly is the core logic of Unity. Once you understand it, you are clearly on the right path!

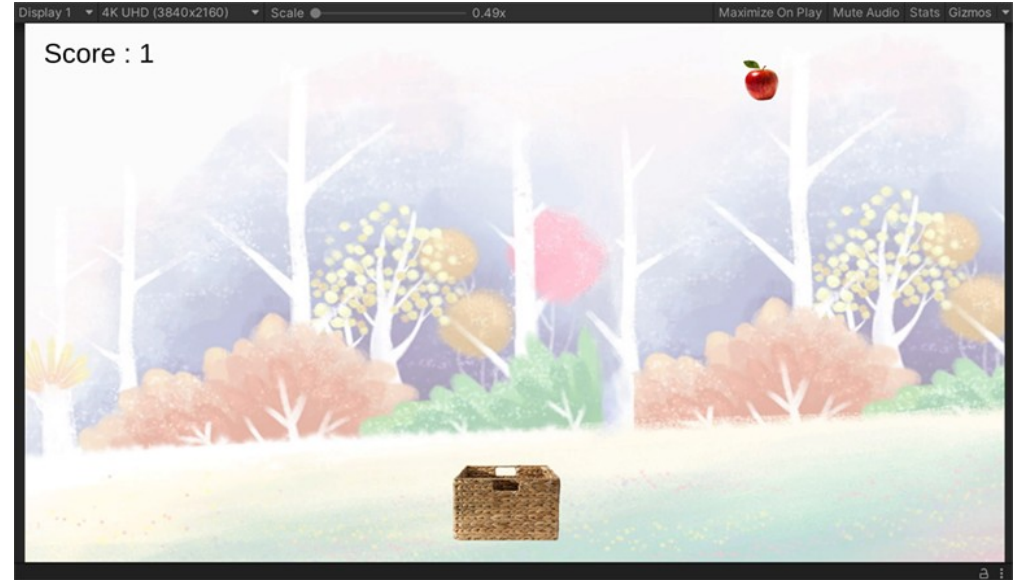
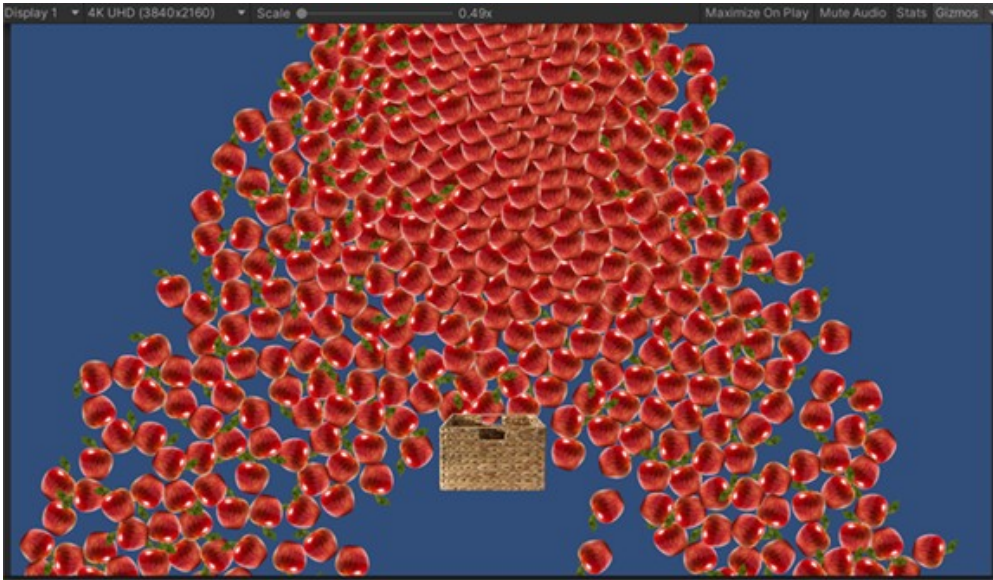


- When compared to other tools, Unity's GUI makes it pretty easy (and fun) to just try things
- So, if you have some spare time, don't hesitate to do mess around in it!  
It will help you get comfortable with the UI! Please just don't mess with our current project ;)



# Next time...

## We'll finish this mini game by...



## Instantiating New Objects!

+ RNG, Text, and using several scenes