LP2B – Multimedia: Digital Representation

- LP2B Tutorial Class n°6 -

Coroutines and Other Upgrades

Doing what we already did, but better

This class will give you an overview of :

- Coroutines : another way to handle real-time -
- Unity Time and manipulating its speed -
- Collisions : using tags to identify objects -
- Animator : using various types of conditions -
- Importing Assets through code -
- Canvas : Creating a Drag-and-Drop system -

About this tutorial...

- You probably have noticed that this tutorial is about a lot of different things...

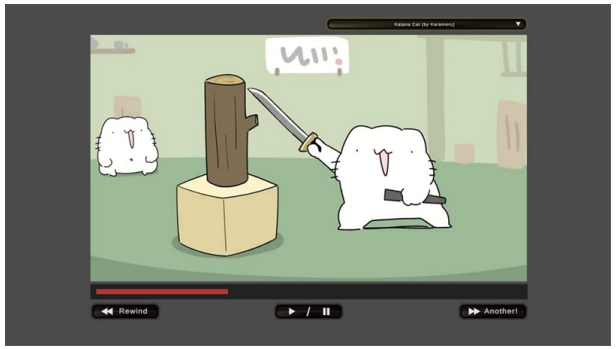
- We are going to talk about various upgrades for our previous projects
  This is the common thread between all those seemingly unrelated ideas

- The big new idea on which we will spend more time on is COROUTINES
  Coroutines are another way to handle real-time, besides Update() and event-based code

- Because of this themed of revising and upgrading, you'll need our older projects
  By that I mean the tutorial projects : "Apple Catcher" and "Video Player"

- What we see in this tutorial is NOT requested to appear in your final project
  Using this knowledge might make your code easier to manage, though. So I encourage you to give it a try. However, not doing so will not be considered a mistake.

- Open Apple Catcher. We'll use it to talk about Coroutines first



Coroutines : Methods that can “Wait”

- All running methods must notify the computer that they are done, or the next frame can’t be displayed
  This is why traditional loops will “block” the app until they are done : they must complete before the next frame can happen

- Coroutines are methods that can notify the system that they are “done”, when in fact they are not. This will allow them to wait until the next frame. Once that next frame starts, they will resume exactly where they stopped.
  All computers NEED their methods to be “done” before displaying the next frame. Coroutines work around this in a clever way

- The command that allows the coroutine to send this “fake done signal” is : `yield return null;`

- A coroutine needs at least one occurrence of “return yield”. If there is none, an error will be displayed

```
IEnumerator LoadScene_Game()
{
    AsyncOperation asyncLoad = SceneManager.LoadSceneAsync("Game");

    // Wait until the asynchronous scene fully loads
    while (!asyncLoad.isDone)
    {
        yield return null;
    }
}
```

We have already written a Coroutine in the past : the method used to load another Scene

This time, we are going to reverse engineer it, and you will finally understand it fully!

Reverse Engineering an Example

```
IEnumerator LoadScene_Game()
{
    AsyncOperation asyncLoad = SceneManager.LoadSceneAsync("Game");

    // Wait until the asynchronous scene fully loads
    while (!asyncLoad.isDone)
    {
        yield return null;
    }
}
```

asyncLoad is an AsyncOperation, which uses Coroutine of its own to load the Scene over several frames.
When the Scene is loaded, its attribute isDone switches from false to true.

Here, the "yield return null" will stop the function until the next frame. Once that next frame comes, the while will resume and look again at asyncLoad.isDone.
AsyncLoad.isDone will automatically become true when the scene is loaded, which will end the while.

Coroutines VS Update : what to choose ?

- In theory, everything that is done with a Coroutine could be done with Update...
  And the opposite is also true, so why are there two options for real time?

- The big difference is that Update will run forever as long as the script is on the Scene, while a Coroutine can wait for a while before ending on its own
  When the code execution reaches the end of a Coroutine, the coroutine actually ends!

- Which is why the choice is usually done like this :

Using Update is Better

For continuous processes that occur during most of the object's lifetime
(Ex : reading input, default behavior...)

Using Coroutines is Better

For processes that are triggered in specific circumstances and/or end on their own
(Ex : a timer we don't always need, waiting...)

Be careful if you use many real time processes !
If they interact with the same attributes, the actions of Update and/or of your coroutines could conflict with one another!

Convenient tools exclusive to Coroutines

- For ease of use, Coroutines are able to use special classes that would not work in a regular context

  Since Coroutines operate on several frames, they have access to exclusive tools based on this “real time dimension”

- Because Coroutines are so often used to wait, the creators of Unity simplified that process with easy-to-use classes :

```
yield return new WaitForSeconds(3.5f);          // Waits for 3.5 seconds. Will be affected if we change Unity's time speed
yield return new WaitForSecondsRealtime(3.5f);  // Waits for 3.5 seconds. Will NOT be affected by change of time speed

yield return new WaitForEndOfFrame();           // Waits until end of frame
yield return new WaitForFixedUpdate();          // Waits for the next cycle of physics engine (separate from frames)

yield return new WaitUntil( myBoolFunction );   // Waits until the provided function returns a "true" boolean
yield return new WaitWhile( myBoolFunction );   // Waits until the provided function returns a "false" boolean
```

- I introduced you to the whole family, but the only one you’re likely to need is WaitForSeconds

  But the good coders among you can probably already see potential use cases for the others!

- You can see here that Unity has its own time, which can potentially be different from real time

  By default, they are the same. But Unity time can be slowed or accelerated, creating “bullet time” effects easily!

Let's Make an Example Together

- In “Apple Catcher”, the game Scene loads immediately when the user pushes any key on the title screen, interrupting all sounds and effects

- We will improve a lot on this transition by relying on Coroutines :

    1) When pushing a key, the music should stop and a confirm sound should play

    2) Create a “fade to white” effect using a big white rectangle sprite

    3) To have this fade effect really work, we need to have a “fade from white” effect added to the Game Scene. *(we'll need a second coroutine)*

    4) Tweak the timing of the first apple so it doesn't spawn during the fade

Now, it's your turn!

- Implement the transition above. There are several ways to do it : either the white rectangle fades by itself, or an external entity controls it
- To fade, we'll use the color property of the SpriteRenderer. The alpha (transparency) of the color will be applied to the Sprite

Controlling Unity Time

- By talking about Coroutines, I introduced the idea of “Unity time”. What’s that?

- Unity time is a modifier to the speed of time. It affects most things that are running over several frames in the app : animators, physics engine, Time.deltaTime, etc…
  This is useful for creating many features and/or to improve feedback

- Changing the speed of Unity Time is done through Time.timeScale
  When it it is equal to 1, Unity Time is the same as real time, since it is a multiplication

- Copy the code to the side on the player character to experiment with this
  Note the GeyKeyUp() method that is used to detect when a key is released

- Unity still has access to real time after Unity Time is changed. But for that, we have to use different properties and/or set the components correctly
  Example : a pause feature is often done by setting Unity time to 0. So any kind of pause menu will need to use real time so it can operate while Unity Time is stopped

- The next slide will explicit how you can edit components you know so that they always work in “real time”, and ignore the changes to “Unity Time”

```
//We slow down time if the spaceBar is pushed down
if ( Input.GetKeyDown(KeyCode.Space) )
{
    Time.timeScale = 0.5f;
}
else if ( Input.GetKeyUp(KeyCode.Space) )
{
    Time.timeScale = 1.0f;
}
```

Reproduce this code and Unity Time will be set to 0.5 while the Spacebar key is pushed

```
//We stop time if the spaceBar is pushed down
if ( Input.GetKeyDown(KeyCode.Space) )
{
    Time.timeScale = 0f;
}
else if ( Input.GetKeyUp(KeyCode.Space) )
{
    Time.timeScale = 1.0f;
}
```

The fact Update still runs means stuff can still happen while time is stopped. This is why this variant works.

You’ll need some experience to see what’s influenced by TimeScale and what’s not

Unity Time VS Real Time

- 3 things we talked about together that are affected by Time.timeScale : Time.deltaTime, Animator, and Rigidbody2D
  So, if we stop time by putting timeScale to 0, those things will become useless

- With the exception of RigidBody2D, there are simple ways to convert those to “unscaled real time”. See below!

- Always remember that Update() and Coroutines run based on frames, not on Time.
  They will keep running the same even if you change Unity time. However, the value of Time.deltaTime inside them will change!

Doing it in Unity Time

Code running with *Time.deltaTime*
(In Update or Coroutines)

Animator is set to “Normal” update Mode

Physics Engine calculations with Rigidbody2D
(Automatic gravity and impact reactions)

Doing it in Real Time

Use *Time.unscaledDeltaTime* instead!
(this variant ignores Time.timeScale)

Set it to “Unscaled Time” update mode instead

NOT POSSIBLE WITH BUILT-IN METHODS!
(you have to code it yourself, which can be very hard if Unity Time = 0)

Animator : using more Conditions types

- Because the tutorial on Animator was so long and complex, we decided to stick to only one type of transition condition : the trigger type

- Because triggers can only work as a “change animation now” transition, they are not always the best way to control an animation.
  They can’t be combined easily, and they do not reflect “states” that may last several frames

- When we look back on our character, we can see that he is either moving or not moving, and that it influences its animation. …So why not keep track of that?

- To do so, create a boolean parameter “isMoving” in the character’s animator

- Let’s continue with this logic : when moving, the character is either moving forwards (to the right) or NOT forwards (to the left). It’s boolean too!

- Create another boolean parameter called “isForwards” in the character’s animator

- With this, we’ll transform our animator to use only 2 parameters instead of 3!

The 3 animations can also bee seen with a “moving/not moving” and “forwards/not forwards” logic
A BOOLEAN LOGIC!

The new Animator Parameter “isMoving” after its creation. The checkbox shows it is of type bool. (unlike circles for triggers)

Once we’re done with our change, we will be able to delete the 3 triggers

Animator : updating our Conditions

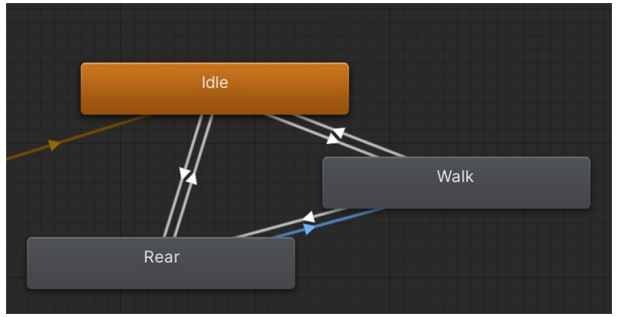
- You need to update your existing transitions so that they now use those booleans

- Once you're done with that, you will need to update your code so it does to!

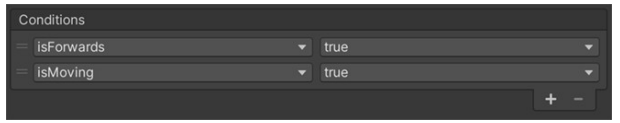
- To manipulate the bool parameters of an Animator, you need to use SetBool() :

```
MyAnimator.SetBool( "parameter name" , true/false )
```

(Blue texts are explanations for the parameters. Not what you should type)



To edit a transition, click on the arrow representing it in the Animator tab



A transition can have more than one condition. In that case, they must ALL BE TRUE ("and" boolean operation)

Now, it's your turn!

- Update the transitions in the animator so they use the two booleans to operate. Instead of the triggers
- Change the code so the value of "isMoving" updates properly : true when moving. False otherwise.
- Change the code so the value of "isForwards" updates according to the direction of movement
- Unlike triggers, it's fine if you change bool parameters every frame!

Importing Assets through Code

- Until now, when we wanted to provide a script with a prefab or sound, we always had to use a public attribute as a way to provide the reference to the script
  You know, the infamous “drag and drop” technique that we used so much this semester!

- ...But what if we could load assets directly, just by providing the file path?

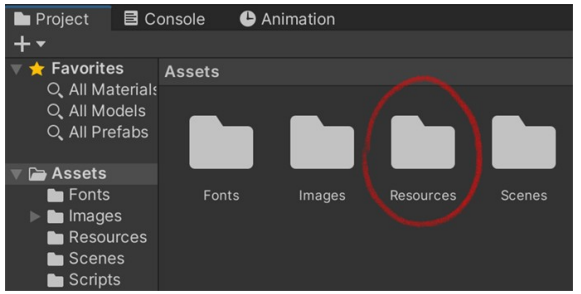
- In Unity, this only works for files placed within a dedicated “Resources” folder
  You have to create this folder manually in the Asset folder. Give it the correct name and the system will identify it automatically as the folder we can import stuff from

- The Resources folder can have sub-folders. So it is possible to organize it
  However, putting all your assets in it isn’t a good idea, for optimization reasons
  Only put stuff you plan to import with code there

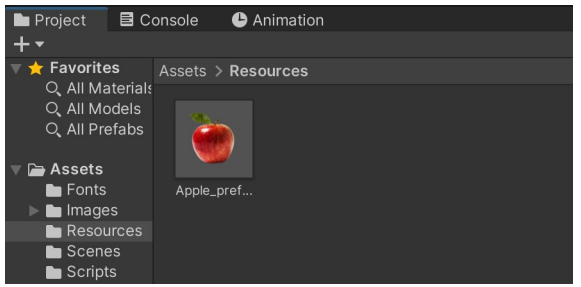
- To import something from your resources folder, use Resources.Load :

```
Resources.Load<Type of Object to import>(local path)
```

(Blue texts are explanations for the parameters. See examples next slide)



The Resources folder must have ./Assets/Resources as a local path in the project. Or it will not work



If I put my Apple prefab in this folder, then it will be ready to be imported anywhere in my code later

Examples of Resource Folder Imports

- Resources.Load is pretty similar to GetComponent : we have to notify the type of asset we want to import, and the result is a reference to the asset.

Therefore, like any other reference, WE HAVE TO SAVE IT IN A VARIABLE OR ATTRIBUTE. Or it will be lost on the next code line!

```
//Importing a prefab/gameobject
GameObject ref_prefab = Resources.Load<GameObject>("my_prefab");

//Importing a sprite image
Sprite ref_sprite = Resources.Load<Sprite>("my_sprite");

//Importing an audio clip
AudioClip ref_audioclip = Resources.Load<AudioClip>("my_sound");
```

Prefabs, Sprites, Sounds...
All assets visible in the Project tab can potentially be imported if you use the correct types

- The paths must be given relative to the Resource folder. Also, you do not have to write any file extension

For this to work, I wrote “my_sound”, and NOT “my_sound.mp3”. Because of that, you can’t give 2 assets the same name

Now, it's your turn!

- Using this knowledge, edit the Apple Spawner so it fetches the apple prefab through this method

Drag-and-Dropping in Unity

- The last UI concept we'll review together is Drag-And-Drop

- Drag-and-Drop uses predefined Unity Events. We need code to handle them
  This code requires to use Interfaces and multi-inheritance. It's pretty complicated to understand in full detail, but you don't need to fully understand them to use them!

- Create a new Scene, then create a Canvas and put an Image in it
  Set whatever you want as its sprite, as long as it isn't offensive of course

- Create a C# script named "DragObject_Script" and add it to the GameObject
  It's inside that script that most of the work to achieve drag-and-drop will be done

- We are going to add an interface to that Script : IDragHandler
  Interfaces are added through inheritance. This is why it is added at the same location than the inheritance from Monobehaviour *(remember that from the first tutorials?)*

- Once IDragHandler is added, it triggers an error. Why?
  Interfaces require specific methods to be added to the script in order to work
  Don't worry, we're going to solve this issue in the next slide!

I am going to use an image of Unity's unofficial mascot, Unity-chan, once again!

(her 3D model is free. Consider it when you want to attempt the jump in the 3rd dimension later!)

```
Script Unity | 0 références
public class DragDrop_Script : MonoBehaviour, IDragHandler
{
    public Canvas ref_canvas;
    private CanvasGroup ref_canvasgroup;
    private RectTransform myRectTrans;

    // Start is called before the first frame update
    Message Unity | 0 références
    void Start()
```

Here, IDragHandler is at the right place... and yet an error is triggered!

It's actually normal : we have to add something!

Adding an Interface's Signature Method(s)

- A big role of Interfaces is ensuring that some methods are present
  If you keep working on Object-Oriented Programming, you will learn more about this eventually

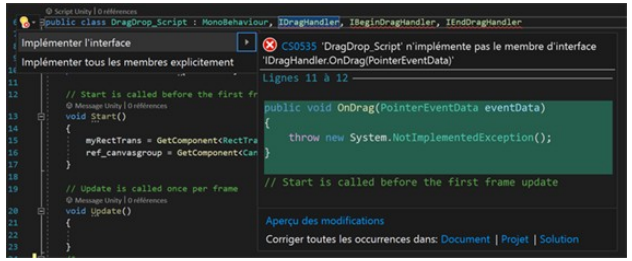
- Because the required methods are a part of the Interface, Visual studio can help us
  Put your cursor over an error and press Alt+Enter. Visual studio will over you help

- In the case of this interface error, one of the help options is "Implement interface"
  Select it and Visual Studio will automatically add the required method(s) to your code

- In the case of IDragHandler, the method is OnDrag(PointerEventData eventData)
  It should have been added automatically by Visual Studio. But its contents are not good!

- Replace the content by a "Debug.log". Test the app. Does the drag and drop work?
  Answer : yes and no. We can see the OnDrag method is called, but there's no movement!

- OnDrag is called every frame while a "Drag action" occurs. But "Drag Action" just means moving the mouse with the button held down
  So it is up to us to implement the movement part. Let's dig into that...



Pressing Alt+Enter over an Interface error will open this menu. Select "Implement Interface" (or the equivalent in your language)

```
0 références
public void OnDrag(PointerEventData eventData)
{
    throw new System.NotImplementedException();
}
```

The "OnDrag" method just after its automatic generation by Visual Studio
The way it throws an exception is actually bad!

```
0 références
public void OnDrag(PointerEventData eventData)
{
    Debug.Log("Dragging occured!");
}
```

To test if this works, you should change the content to something like this!

Drag action and Movement

- Dragging does not automatically move the object. And for good reason !
  Dragging can have other functionality in an app. Especially when designing for phone or tablet

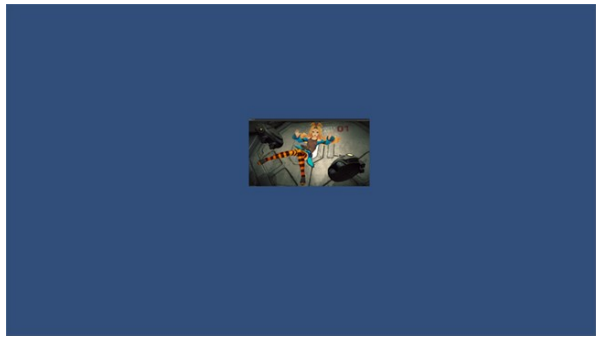
- In the case of Drag and Drop, we want the item to move with the cursor during drag
  Because the item is a Canvas item, we'll have to manipulate its position with RectTransform

- Create an attribute to record a reference to RectTransform. Initialize it in "Start()"
  RectTransform is a component like any other : you can use GetComponent like usual

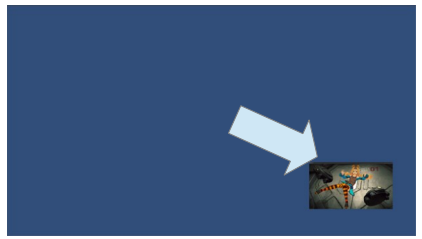
- The eventData parameter of onDrag contains info on the drag movement
  In Particular, eventData.delta contains the change of position that occurred since the last frame

- By combining all the ideas above, we obtain this line of code to put into OnDrag():
  It will move the item with the drag delta each frame, resulting in what we wanted!

```
myRectTrans.anchoredPosition += eventData.delta;
```



My image isn't moving!
...But I got the Debug.Log message, so the OnDrag function was actually called!



Now, the image can move!
BUT...

Now, it's your turn!

- Setup everything so this line of code works and allows us to drag our image!

Wait… is that Drag actually accurate?

- If you test the drag enough, you can see the Image doesn't exactly follow the cursor

- The reason for this difference is that the delta is unscaled, but the Canvas is!
  The Canvas is scaled to match the screen, so its scale value isn't the standard 1!

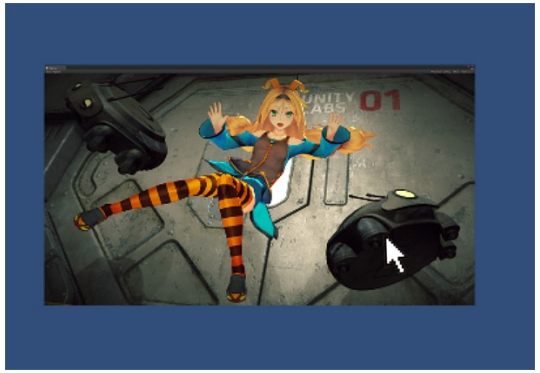
- The solution is to divide the delta by the scale of the Canvas, like this :

```
myRectTrans.anchoredPosition += eventData.delta / ref_canvas.scaleFactor;
```

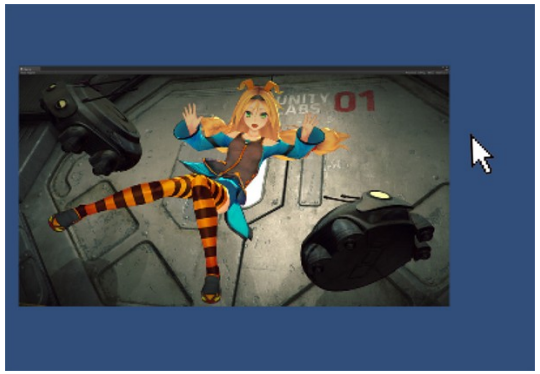
Now, it's your turn!

- Provide the Script with a reference to the Canvas
- Once it's done, update the OnDrag method so it uses the division above!

- Once you have done this, you'll finally have a drag that perfectly matches the mouse!
  Meaning the relative position of the cursor from the image will stay the same during the drag



I start the drag...



After some movement, this happens!
We can see the movement of the cursor and of the image do not match properly

Other Drag-related Interfaces

- Now, we can drag our image properly. ...But what about the drop part?
  We need to detect the area where the item is dropped, so we can react properly

- Adding Interfaces is once again the solution. In fact, we will need many!
  A Script can inherit from as Many Interfaces as desired. Just add them to the inheritance!

- Add the IBeginDragHandler and IEndDragHandler interfaces to the script
  They, of course, come with required methods. Use Visual studio's help to generate them

- Those Methods detect the start and end of a drag. We'll need them later!
  Delete the contents of the methods and replace them with a Debug.Log for now

- You will now create an area in which to drop our item, with IDropHandler interface

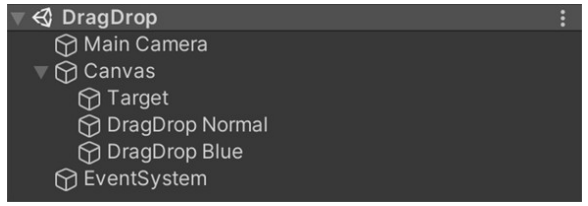
```
0 références
public void OnBeginDrag(PointerEventData eventData)
{
    throw new System.NotImplementedException();
}

0 références
public void OnEndDrag(PointerEventData eventData)
{
    throw new System.NotImplementedException();
}
```

```
0 références
public void OnDrop(PointerEventData eventData)
{
    throw new System.NotImplementedException();
}
```

The 3 methods of the 3 interfaces on this slide, after their Auto-generation. Their contents will throw exceptions (errors), so they must be changed

Now, it's your turn!

- Create a GameObject with a Canvas Image to serve as a target Area
- Make sure that area appears behind our drag item *(see tip on the side)*
- Create a new Script and attach it to that GameObject : "DropZone_Script"
- Add the IDropHandler interface to it, generate its method with Alt+Enter



In a Canvas, the order in the Hierarchy is one way to control what's in Front and what's behind. The more an item is down below in the list, the more "in front" it is!

An Unexpected Problem

- When we drop our item, the system checks what's under our mouse cursor with an operation called a raycast
  - It sends a virtual line (a ray) through the z axis and checks the first object the ray intersects

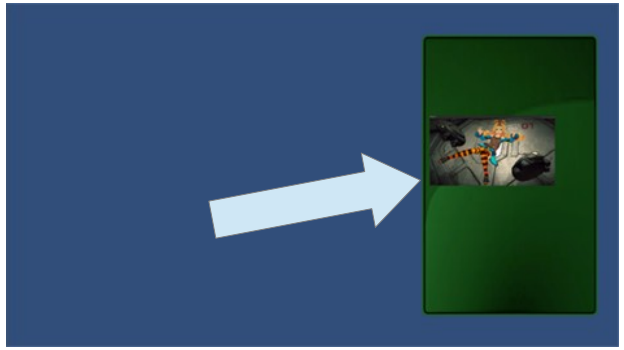
- But, by definition, the Mouse cursor will be on drag object at that moment!
  - Since our Image follows the cursor during the drag, it will be under the cursor when we let go!

- The solution? Make the drag Image "immune" to raycasts while it is dragged
  - This is where IBeginDragHandler and IEndDragHandler, which we added earlier, will be useful

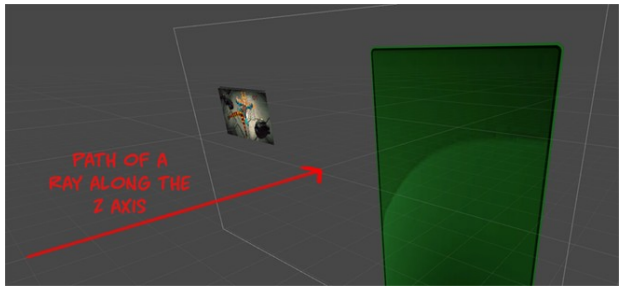
- Add a "Canvas Group" component to the drag image. We need it to disable raycast
  - As a component, you can access it with GetComponent<CanvasGroup>() in your code

- Once you have a reference to Canvas group, you can disable raycasts on the GameObject by using the following command :

```
ref_canvasgroup.blocksRaycasts = false;
```



I have my drag Image, I have my target area with the IDropHandler interface properly set... And yet nothing happens when I do this!



The way the computer detects the object on which the mouse was released is a Raycast In our case,that Raycast is actually intercepted by the image on top, preventing the green area, with OnDrop(), from receiving it!

(for this screenshot, I messed with the camera To show the raycast in the 3D space)

Finally detecting our Drop Area!

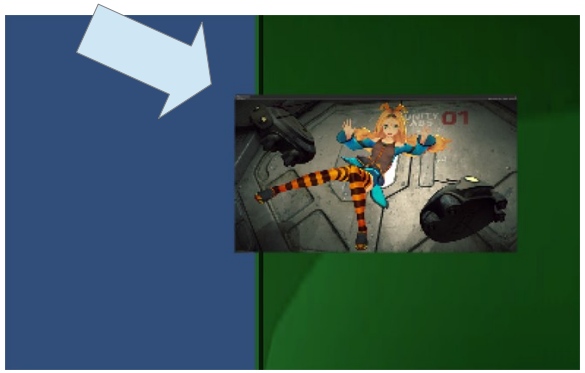
- Raycast is also what allows the system to detect when we start clicking on an item
  That's why we will need to reactivate "blockRaycasts" once the drag is over!

- Fortunately, we have 2 methods : OnBeginDrag and OnEndDrag, which are called at the start and end of a drag action respectively!
  Since our Image follows the cursor during the drag, it will be under the cursor when we let go!

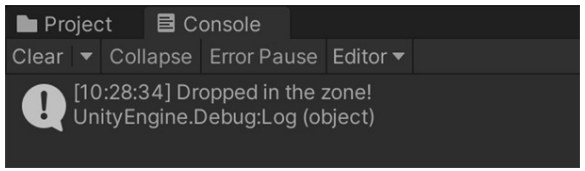
Now, it's your turn!

- Deactivate and reactivate raycasts at the correct times
- Place a Debug.Log message that says "Dropped in the zone!" in the OnDrop() method that exists in "DropZone_Script"
- Ensure the drag image is in front by altering the order in hierarchy

- Once you have done all that, test the app : the "Dropped in the zone!" message should appear when you drag and drop the image in the target area!
  Phew! That was some work! But now you finally have the principle of drag-and-drop!



You win if when you do this...



You get your Debug.Log message displayed in the console!

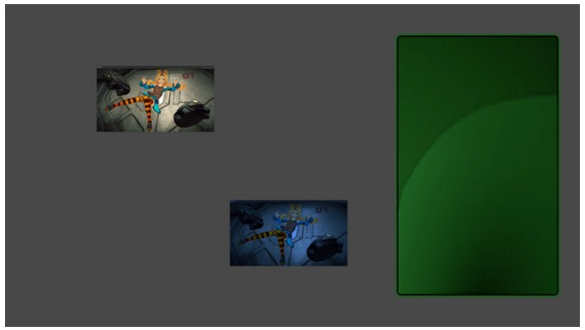
(you should also be able to drag and drop the image again even after placing it in the zone)

Identifying the Object dropped

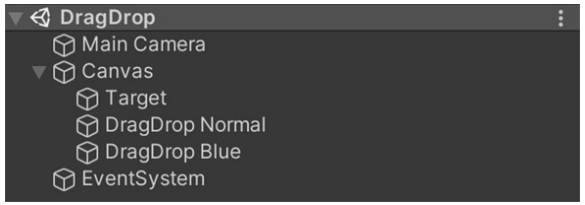
- Create another dragable object. Give it a different name and graphic
  Either use a different Sprite for the image, or use the same with a color modifier

- In the OnDrop method of the drop zone, we can access the object dragged in
  To do so, we have to use the information inside the eventData parameter

- You can access the GameObject dropped with eventData.pointerDrag
  Since this returns a GameObject, you can access properties available on a GameObject on it



My other object is a copy of the first one, on which I changed the color.
In Unity, you can duplicate with Ctrl+D



Rename your GameObjects with clear Names, as we will use those to identify them

Now, it's your turn!

- Adapt DropZone_Script so it plays a different sound depending on the object dropped on the zone
- Simplest solution is to rely on GameObject Names to identify them
- You may look back at the tutorial on sound if you need to

- Now that you can identify the object, you can freely expand on the drag-and-drop system by yourself!
  From there, you can set up more advanced algorithms that will create the behavior desired for your game/app!

A Gateway towards Advanced Concepts

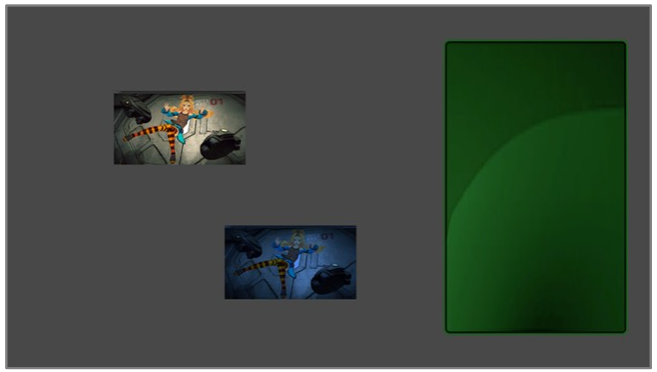
- When talking about the UI, we talked about both Events and Interfaces

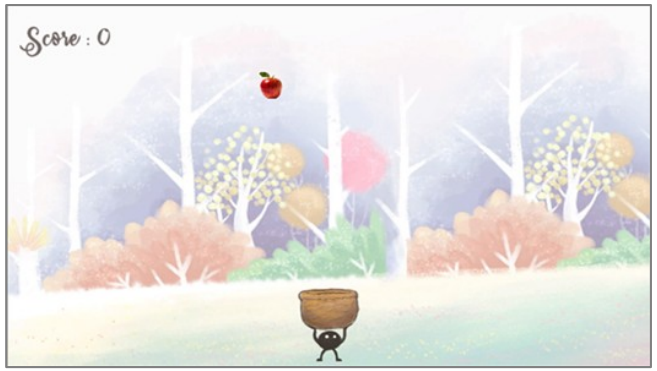
- Those are pretty big programming concepts. In this class, we only manipulated them indirectly.
  If you want to up your game. You can try to look deeper into those concepts

- In fact, this is the general idea of this class : if you dive deeper into everything we have seen together, you'll discover many things
  Unity is a great tool to experiment and learn complex stuff in a more "visual" way

- However, you should also remember that you do not need to use anything from tutorial 6 in the final project.
  Using advanced stuff only makes your project better if it's done right. So it's a risk!

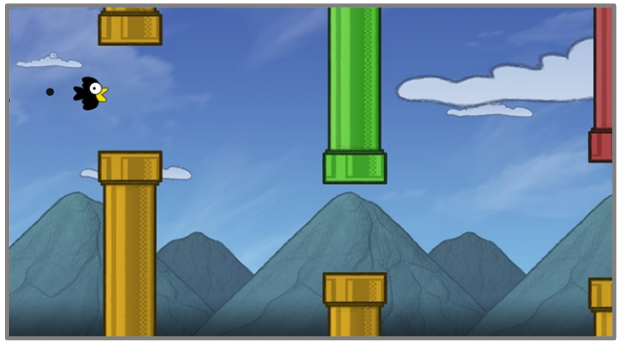
- Coroutines are the most likely to help you doing the Flappy Bird minigame. But in theory, even them are not required.





And now, towards the Final project!

The Techniques we have seen today will help you make it even better!



(But using them is not mandatory either. The first 5 tutorials cover all the concepts you absolutely need)

That's all for this Semester!

The time left will be dedicated to questions and reviewing together the points on which you still have problems.

I hope this Class has motivated you to keep studying the great tool that is Unity in the future!