Practical 1

Prerequisites

1. Windows (preferably) or Mac
2. Unity Hub
3. Unity 2020.3+ Module Android (SDK & NDK, Open JDK)
4. Visual Studio + Unity Tool

Objectives

Make a small 3D platformer
Learn the **software** interface
Learn about and create a new **Material**
Learn about and create a new **Prefab**
Learn about Unity **Physics** system
Learn basic **C# scripting**
Learn about the **Input System**
Learn about **Realtime Lighting**
Learn about and create a new **Build** for PC or Mac

Making a new Unity project

Open Unity Hub

Check in Installs if Unity 2020.3 + Android module is installed

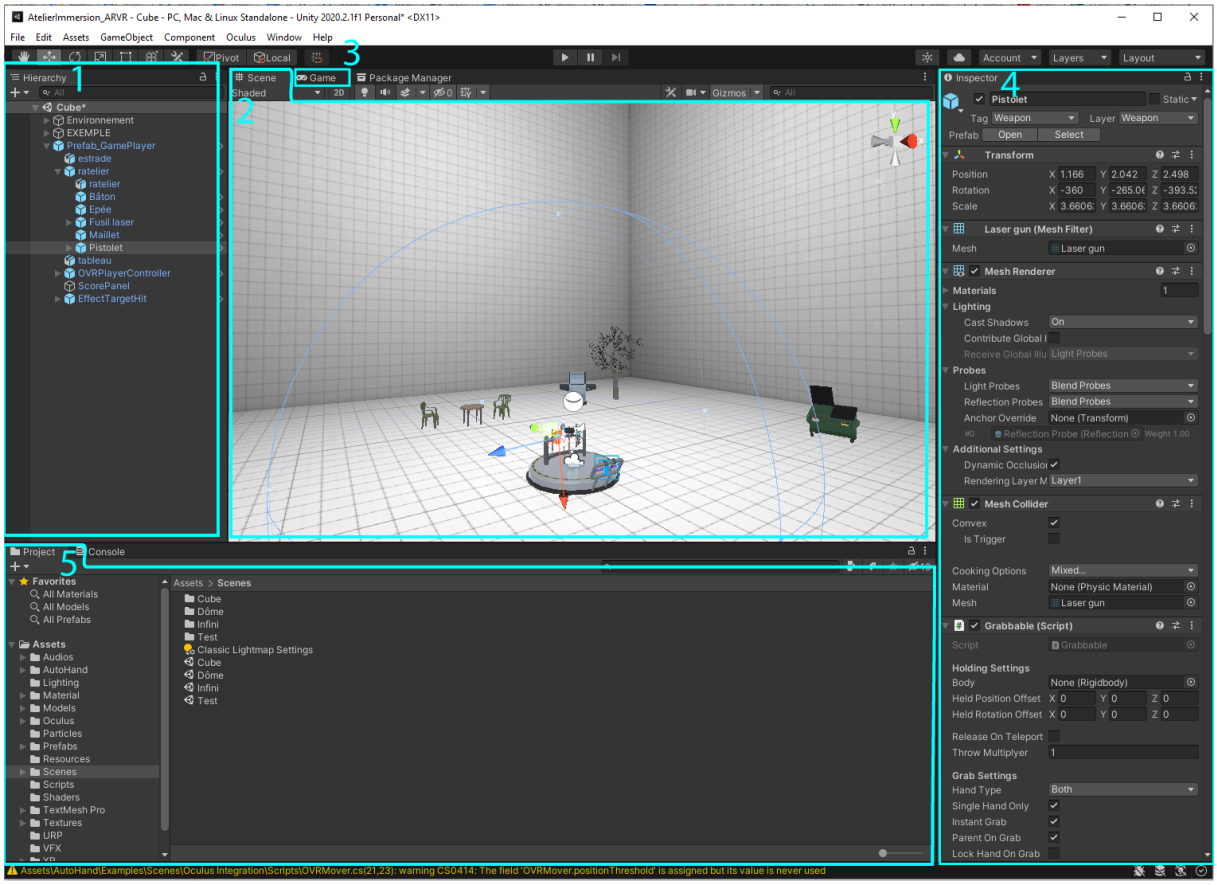
If it's not installed :
Click on Add
Select Unity 2020.3 and click Next
Check Microsoft Visual Studio / Android Build Support and click Done

Go to Projects, click New and select the version 2020.3

Select the 3D template, rename project and click on Create

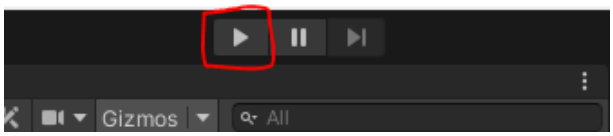
Unity User Interface

Main windows (Scene, Game, Project, Console, Inspector, Hierarchy).



1. **Hierarchy** : Shows all GameObjects in the scene
2. **Scene View** : To manipulate and design the game
3. **Game View** : Camera view of the game
4. **Inspector** : Gives access to the components of a selected object
5. **Project** : Displays the file tree of the project folder and contains all project items

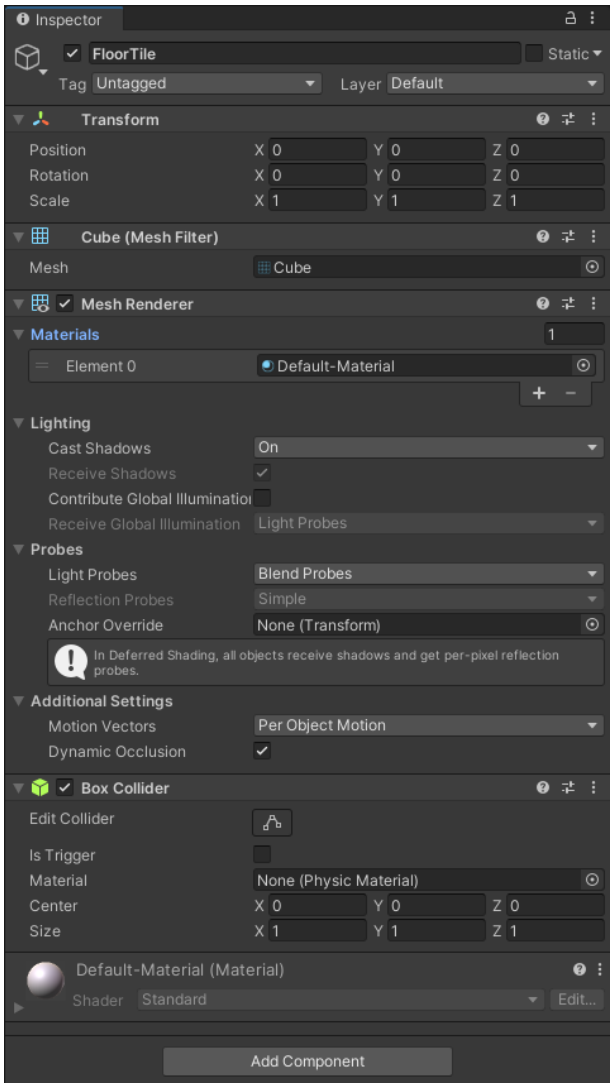
To simulate the game, you have to click on the Play Mode Button. Be careful to check if you are not in Play Mode to modify your scene, Unity will not save any changes in Play Mode. To Exit Play Mode just click again on the same Button.



Create a new Cube in the Hierarchy. Rename it "FloorTile".

His components in the Inspector are:

- **Transform** : it's the Position, Rotation, Scale of the GameObject. Every gameObject has at least a Transform.
- **Mesh Filter** : indicate the type of Mesh it should be rendering.
- **Mesh Renderer** : indicate how to render this Mesh.
- **Box Collider** : tells the Physics Engine how it would be colliding with other objects in the scene.



Move around the scene and move / rotate the Camera and the FloorTile to learn about the Transforms and tools (translation, rotation, scaling).

Materials

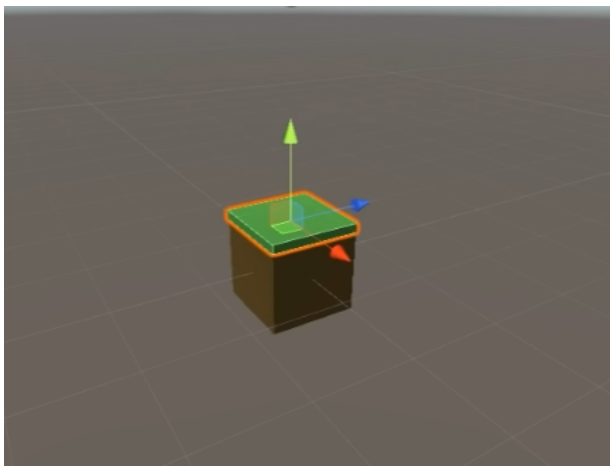
Create a new Folder in the Asset project, and rename it “Materials”.

Create a new Material in this new Folder, name it “FloorDirt”.

Modify the FloorDirt Material (for example, change the color to brown and the smoothness to 0).

Drag and drop the FloorDirt Material on your FloorTile (in the Scene view or in the Hierarchy).

Now you can create the grass on Top of the FloorTile by creating a new GameObject and a new Material.



Make the Grass gameObject a Child of FloorTile.
See the movements with a parent / child gameObject and see the difference between the World and Local Transform value.

Prefabs

A Prefab is a Pre Fabricated GameObject.

Create a new Folder named “Prefabs” in the Assets.
To create a Prefab of the ‘FloorTile’, just drag and drop the FloorTile gameObject (Hierarchy) to your new folder Prefabs (Project).

Now you can just drag and drop the new Prefab into your scene to add more FloorTiles.

GameObjects in the Hierarchy from Prefabs are Blue.

Put all your FloorTiles gameObjects in a parent named ‘Level’.

Prefabs are good to be modified, and the modifications will apply on all gameObjects prefab

For our example we want to remove the Grass Box Collider of the FloorTiles.

Open the Prefab Editor by double clicking on our Prefab ‘FloorTile’ in the Project window.
Delete the Box Collider of the Grass object.

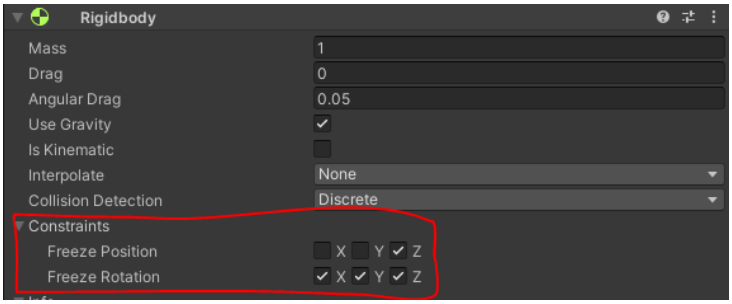
Player & Physics

Let's add a Player into our scene!

Create a new Capsule gameObject, rename it, replace it and resize it.

Add a new Rigidbody component to your Player. Check if 'Use gravity' is checked on and Play the scene to see your Player fall on the ground.

Add some constraints to your Rigidbody:



The frozen position depends on the axis on which the scene is oriented. Meaning that if your player is moving along the Z-axis you will want to freeze the X axis, however if the player is moving along the X-axis you will want to freeze the Z axis.

User Input & C# Scripting

1. **Setup Script Editor:**
   First of all, check in Edit > Preferences > External Tools > External Script Editor and select Visual Studio.

2. **Create a new Script:**
   Create a new Folder in your Assets 'Scripts'.
   Create a 'C# Script' in your folder. Name it 'Player'.
   Put the script 'Player' on your Player gameObject.
   Open the script by double-clicking on it.

   When creating a script, you are essentially creating your own new type of component that can be attached to Game Objects just like any other component.

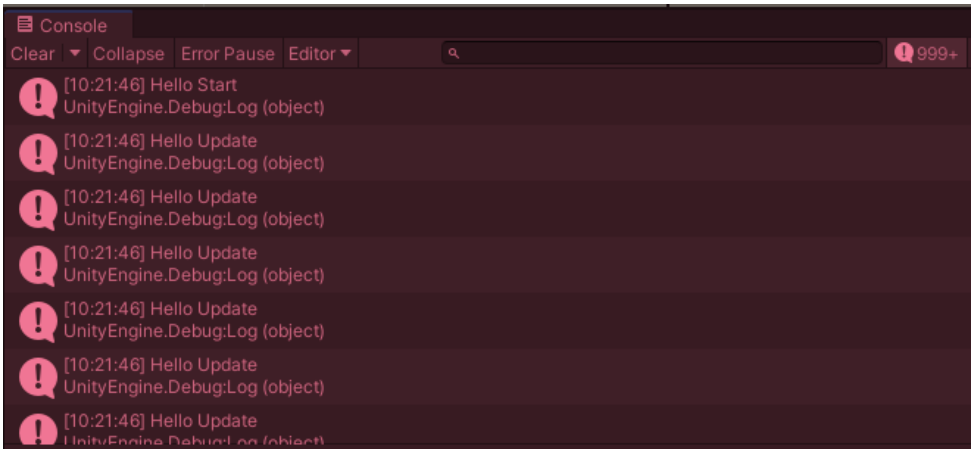
3. **Methods 'Start' & 'Update':**
   See the Namespaces, the class, and the Methods.

   First we are going to see the Debug.Log
   Debug.Log is used to print informational messages that help you debug your application.
   You can see the Debugs, Warning and Errors in the Console.

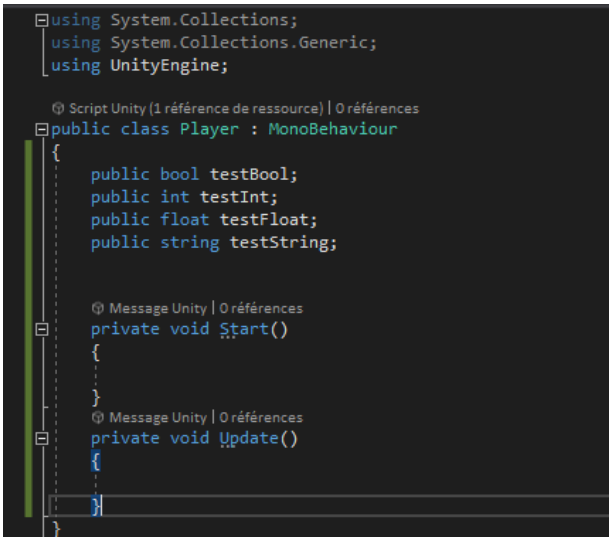
   The Method 'Start' is called at the end of the first frame.
   The Method 'Update' is called at all frames.

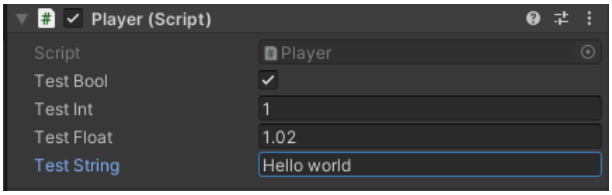
```
Message Unity | 0 références
private void Start()
{
    Debug.Log("Hello Start");
}

Message Unity | 0 références
private void Update()
{
    Debug.Log("Hello Update");
}
```



4. **Variables:**
   Just like other Components often have properties that are editable in the Inspector, you can allow values in your script to be edited from the Inspector too.
   In C#, to see a variable in the Inspector is to declare it as public.
   For example:





5. **Conditional statements:**
   For example the IF Statement:

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

Script Unity (1 référence de ressource) | 0 références
public class Player : MonoBehaviour
{
    public int myValue;

    //private void Start()
    //{

    //}
    Message Unity | 0 références
    private void Update()
    {
        if(myValue < 10)
        {
            Debug.Log("My Value is lower than 10");
        }
        else if(myValue > 10)
        {
            Debug.Log("My Value is higher than 10");
        }
        else if(myValue == 10)
        {
            Debug.Log("My value is equal to 10");
        }
    }
}
```

   Test this code in Unity and change the value of your new Variable 'myValue' to see the different Logs.

6. **User Input:**
   Unity allows us to use different Inputs (mouseInputs, keys, axis, joysticks ...)
   For example:

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
       Script Unity (1 référence de ressource) | 0 références
5  public class Player : MonoBehaviour
6  {
           Message Unity | 0 références
7      private void Start()
8      {
9
10     }
           Message Unity | 0 références
11     private void Update()
12     {
13         if (Input.GetKeyDown(KeyCode.Space))
14         {
15             Debug.Log("Space key was Pressed Down !!");
16         }
17     }
18 }
19
```

When the space key is pressed down it will print our message in the Console. Now if we want to make the Player jump when the space key is pressed down we have to apply force on the Rigidbody!

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

Script Unity (1 référence de ressource) | 0 références
public class Player : MonoBehaviour
{
    Message Unity | 0 références
    private void Start()
    {

    }
    Message Unity | 0 références
    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            GetComponent<Rigidbody>().AddForce(Vector3.up, ForceMode.VelocityChange);
        }
    }
}
```

Save your script and try it in Unity.
We can see, when we press Space the Player is jumping. (But not enough).
Multiply the magnitude of the Vector3 to increase the force.

```
GetComponent<Rigidbody>().AddForce(Vector3.up * 5, ForceMode.VelocityChange);
```

And try to Jump again!
It's not very optimized to 'GetComponent' the Rigidbody in the Update, so to optimize we can declare the Rigidbody in the Start method.

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

Script Unity (1 référence de ressource) | 0 références
public class Player : MonoBehaviour
{

    private Rigidbody rb;

    Message Unity | 0 références
    private void Start()
    {
        rb = GetComponent<Rigidbody>();
    }
    Message Unity | 0 références
    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            rb.AddForce(Vector3.up * 5, ForceMode.VelocityChange);
        }
    }
}
```

The Player can jump but can't move left and right yet.
We have several options to move our Player, the Transform Position and Add Force to the Rigidbody. We will see the two options.

To move the Player left or right Unity offers us different inputs like Horizontal. (Look into the Input Manager to see the inputs, Edit > Project Settings > Input Manager).

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

Script Unity (1 référence de ressource) | 0 références
public class Player : MonoBehaviour
{

    private Rigidbody rb;
    private float horizontal;

    Message Unity | 0 références
    private void Start()
    {
        rb = GetComponent<Rigidbody>();
    }
    Message Unity | 0 références
    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            rb.AddForce(Vector3.up * 5, ForceMode.VelocityChange);
        }

        horizontal = Input.GetAxis("Horizontal");
        Debug.Log(horizontal);


    }
}
```

Test in Play Mode and press the left and right arrows or the A and D key. We can see the value go to -1 to 1 in the Console.

First option: **Transform Position**

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

Script Unity (1 référence de ressource) | 0 références
public class Player : MonoBehaviour
{

    private Rigidbody rb;
    private float horizontal;

    Message Unity | 0 références
    private void Start()
    {
        rb = GetComponent<Rigidbody>();
    }
    Message Unity | 0 références
    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            rb.AddForce(Vector3.up * 5, ForceMode.VelocityChange);
        }

        horizontal = Input.GetAxis("Horizontal");
        transform.Translate(Vector3.right * horizontal * Time.deltaTime);
    }
}
```

Second option: **Apply Force to the RigidBody**

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

Unity Script | 0 references
public class Player : MonoBehaviour
{
    private Rigidbody rb;
    private float horizontal;

    Unity Message | 0 references
    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    Unity Message | 0 references
    void Update()
    {
        if(Input.GetKeyDown(KeyCode.Space))
        {
            rb.AddForce(Vector3.up * 5, ForceMode.VelocityChange);
        }

        horizontal = Input.GetAxis("Horizontal");

        if (horizontal != 0)
        {
            rb.AddForce(Vector3.right * horizontal * 10, ForceMode.Acceleration);
        }
    }
}
```

The Player can Jump with the space key, and move left and right with the left or right arrows (or A key, and D key).

Again, if you are moving along the Z axis, you will want to put Vector3.forward instead of Vector3.right.

The right way to do it with Physics (rigidbodies) is to add forces not in the Update

Method but in the FixedUpdate Method. FixedUpdate is called every fixed framerate frame (0.02 seconds).
This brings a bunch of new problems, and it can quickly become a real headache, so for now just copy this code:

```
public class Player : MonoBehaviour
{
    private Rigidbody rb;
    private float horizontal;
    private bool inputJump;

    Event function
    private void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    Event function
    private void Update()
    {
        if (!inputJump)
        {
            inputJump = Input.GetKeyDown(KeyCode.Space);
        }

        horizontal = Input.GetAxis("Horizontal");
    }

    Event function
    private void FixedUpdate()
    {
        if (inputJump)
        {
            rb.AddForce(Vector3.up * 5, ForceMode.Impulse);
            inputJump = false;
        }

        if (horizontal != 0)
        {
            rb.AddForce(Vector3.right * horizontal * 10, ForceMode.Acceleration);
        }
    }
```

Another problem we can observe, is that our player can jump multiple times in the air.

Here's how to disable the multiple jumps; you will need to create a new boolean variable called isGrounded in your Player class:

```
private void Update()
{
    if (!inputJump && isGrounded)
    {
        inputJump = Input.GetKeyDown(KeyCode.Space);
    }

    horizontal = Input.GetAxis("Horizontal");
}

Event function
private void FixedUpdate()
{
    if (inputJump && isGrounded)
    {
        rb.AddForce(Vector3.up * 5, ForceMode.Impulse);
        inputJump = false;
        isGrounded = false;
    }

    if (horizontal != 0)
    {
        rb.AddForce(Vector3.right * horizontal * 10, ForceMode.Acceleration);
    }
}

Event function
private void OnCollisionExit(Collision other)
{
    isGrounded = false;
}

Event function
private void OnCollisionEnter(Collision collision)
{
    isGrounded = true;
}
```

OnCollisionEnter is called when this collider/rigidbody has begun touching another rigidbody/collider. And OnCollisionExit is called when this collider/rigidbody has stopped touching another rigidbody/collider.

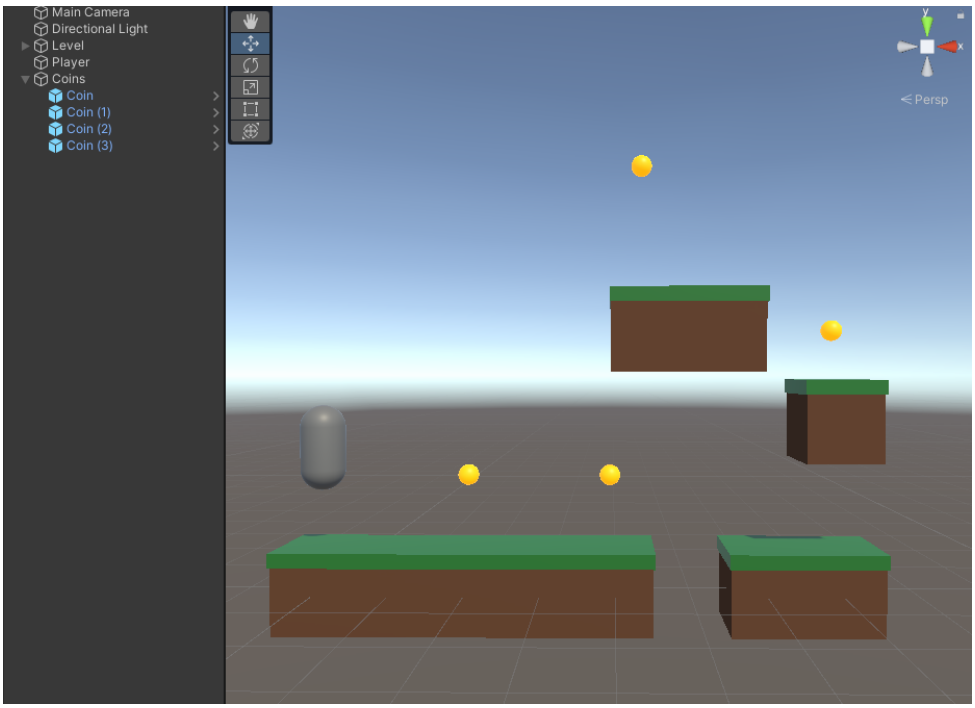
As we can see in Play our Player gets stuck between two colliders and keeps 'Enter'ing and 'Exit'ing the colliders, and can't Jump.

OnCollisionStay is called once per frame for every collider/rigidbody that is touching the rigidbody/collider.

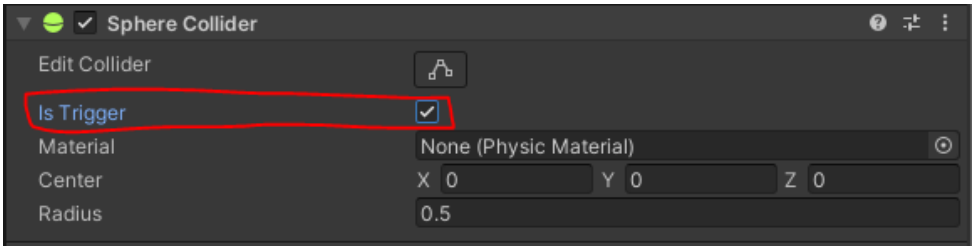
Replace OnCollisionEnter by OnCollisionStay to fix our ground detection.

Now, let's add some collectables!

Create a new Sphere GameObject and call it Coin, and make it a Prefab ! Apply a new material and place several in the scene!



To detect these coins we can use the OnTriggerEnter, when a GameObject collides with another GameObject, Unity calls OnTriggerEnter. For this you have to indicate to Unity the Trigger Collider.



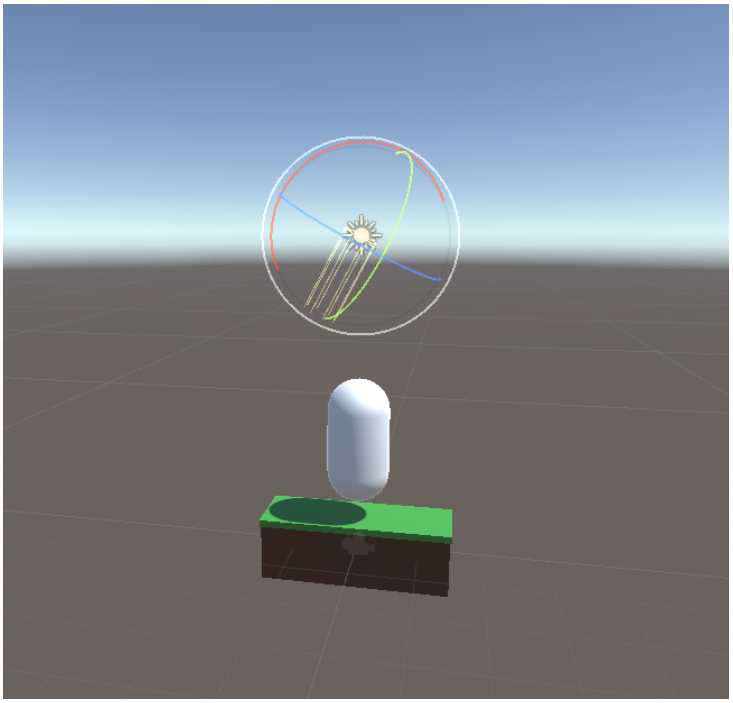
When the Player detects the Trigger of a coin, we destroy the coin.

```
Message Unity | 0 références
private void OnTriggerEnter(Collider other)
{
    Destroy(other.gameObject);
}
```

Lighting

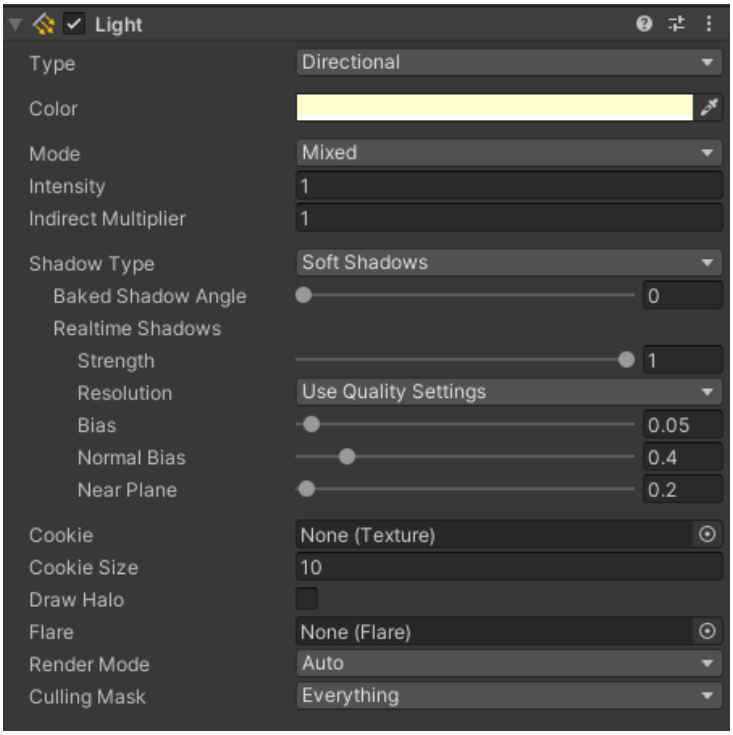
Lighting is an essential component of every scene. Its quality is also usually what sets professional projects apart from amateur ones.

Select the scene's Directional Light and rotate it around. You should be able to observe the lighting change in your scene. You might have already seen that the skybox' sun also rotates along that light.



As the light's type currently is "Directional", its position in the scene doesn't matter at all; simply put, all it does is simulate the sun and its rotation. It is possible to simulate a day and night cycle just by rotating that game object.

We will be exploring the different types of lights in a moment, but for now, play around with the light's settings:

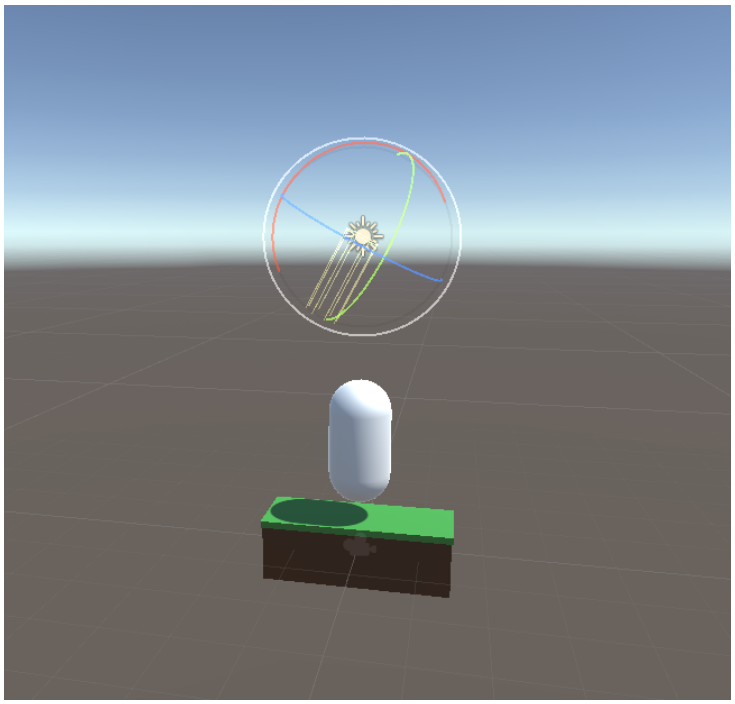


The most interesting settings are the Color, the Intensity and the multiple Shadow settings.

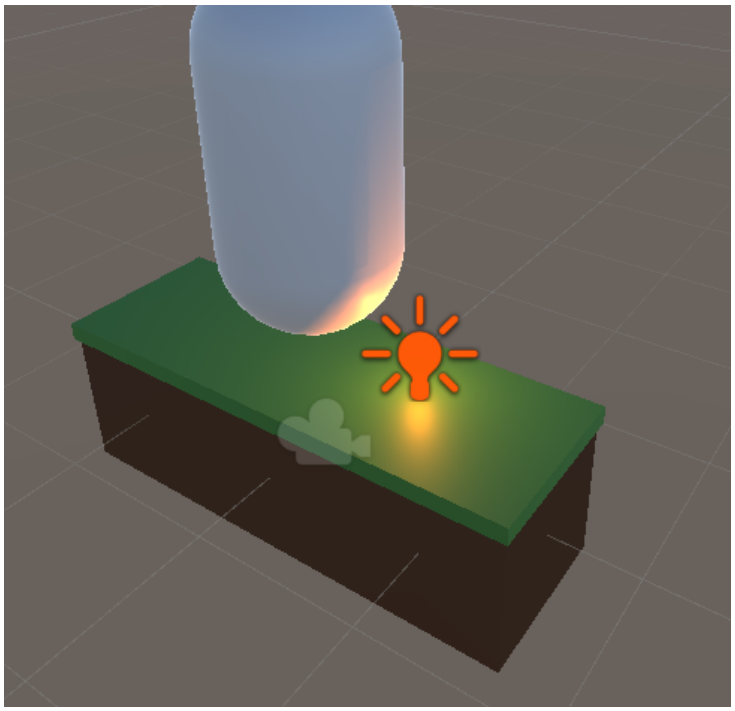
Important: do not change the 'Mode' to 'Baked', as it would cause the light to not be visible in the scene anymore. You would need to do some additional setup in order to 'bake' the light. Baking lights means calculating all the effects that light source is going to have on every object in the scene and then directly printing those to some additional "textures" (named lightmaps) of the objects. This is a very lengthy process and you unfortunately won't have time to try it out. However, know that in real-life projects this is necessary, as real-time lighting is very expensive in terms of computer power; even though it might take days, or even weeks for the baking to finish. (Note that it could potentially only take around one minute given the scale of this project, but it is very dependent on your computer's hardware, and it could easily jump to upwards of 10 or 20 minutes. It also brings along some more problems, such as lightmap overlapping, which is way too advanced for an introductory course like this one).

Now, we will play around with the different types of lights. Feel free to experiment with those! Also, if you want more information, head to this page of the Unity Documentation.

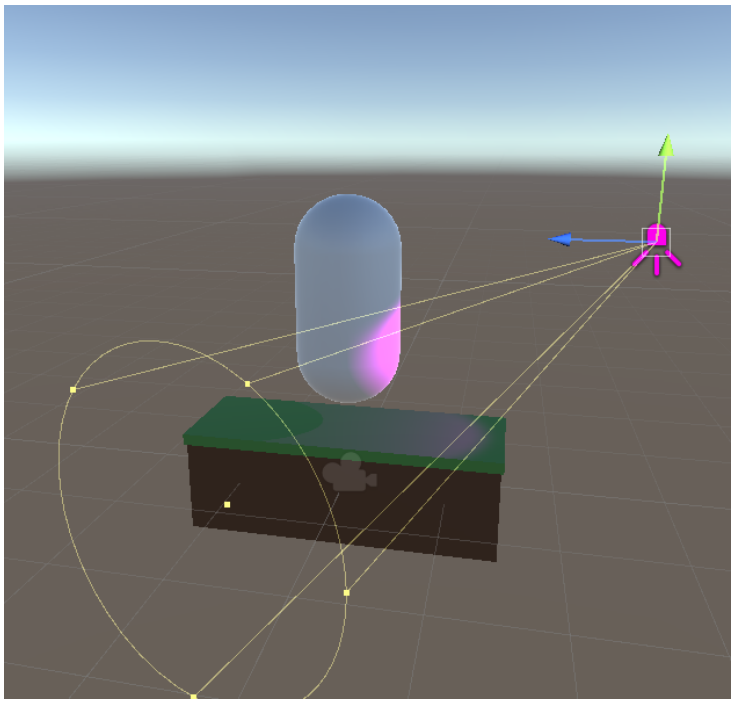
- **Directional** - This is the current type. This light is located infinitely far away, and it emits light in one direction only.



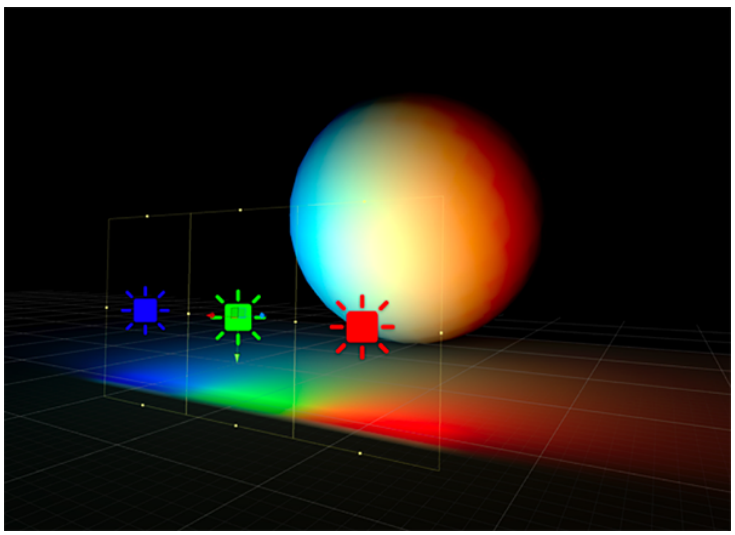
- **Point** - This makes the light a point in the scene, that then emits light all around itself equally. You can adjust the size of the light. You can use it for campfires, lanterns...



- **Spotlight** - Similar to the point light, however it is constrained to a certain angle. Emits light in a cone, therefore it's the go-to solution for simulating flashlights.

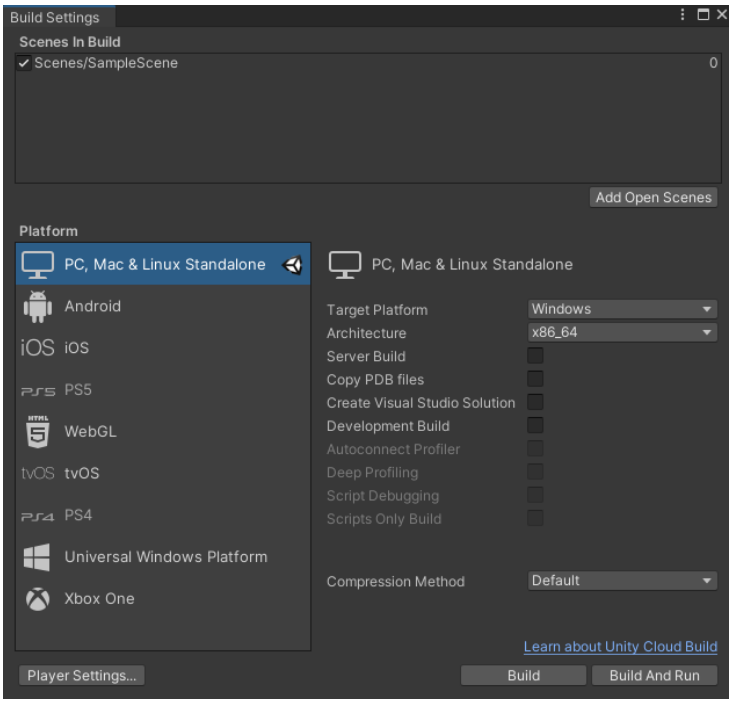


- **Area** - This is a special type of light that only works in 'Bake' mode as it is particularly expensive, therefore you won't be able to experiment with this one. It is similar to a spotlight, but emits light in a rectangular shape, has infinite range with diminishing effects on objects the farther away they are from the light.



Build

Now, you are going to tell Unity to export your project in the form of an executable file. In order to do that, head to 'Build Settings' under the 'File' menu in the menu bar. This is what you should see:



As you can see, Unity is capable of exporting your project to multiple different platforms, but for this practical we are going to target the PC platform. Make sure that the platform currently selected is 'PC, Mac & Linux Standalone'. There should be a small Unity logo next to the name, like in the screenshot above.

To the right, there are multiple checkboxes with plenty of different options; just leave everything unchecked. If you want to know more about those options, Unity's documentation here might be helpful.
Look at the top of the window, at the 'Scenes In Build' category. Make sure your scene is there and ticked. If your scene is not there, you can click on the 'Add Open Scenes' button. Unless you have closed your scene, it should suddenly appear already ticked.

Now, click on either 'Build' or 'Build And Run' at the bottom right. This will prompt you to select or create a folder where to build. Go ahead and create a new folder 'Build' and select it.

Wait for the build to finish.

If you clicked the 'Build And Run' button, your application should be executed automatically. Otherwise, it should automatically display your 'Build' folder with your executable file in it. You just need to launch it.

And there you go!

Please note that the game has no end for now, so if you fall, you will get stuck. If that happens, you need to restart your game.