Practical 2

Prerequisite

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

Objectives

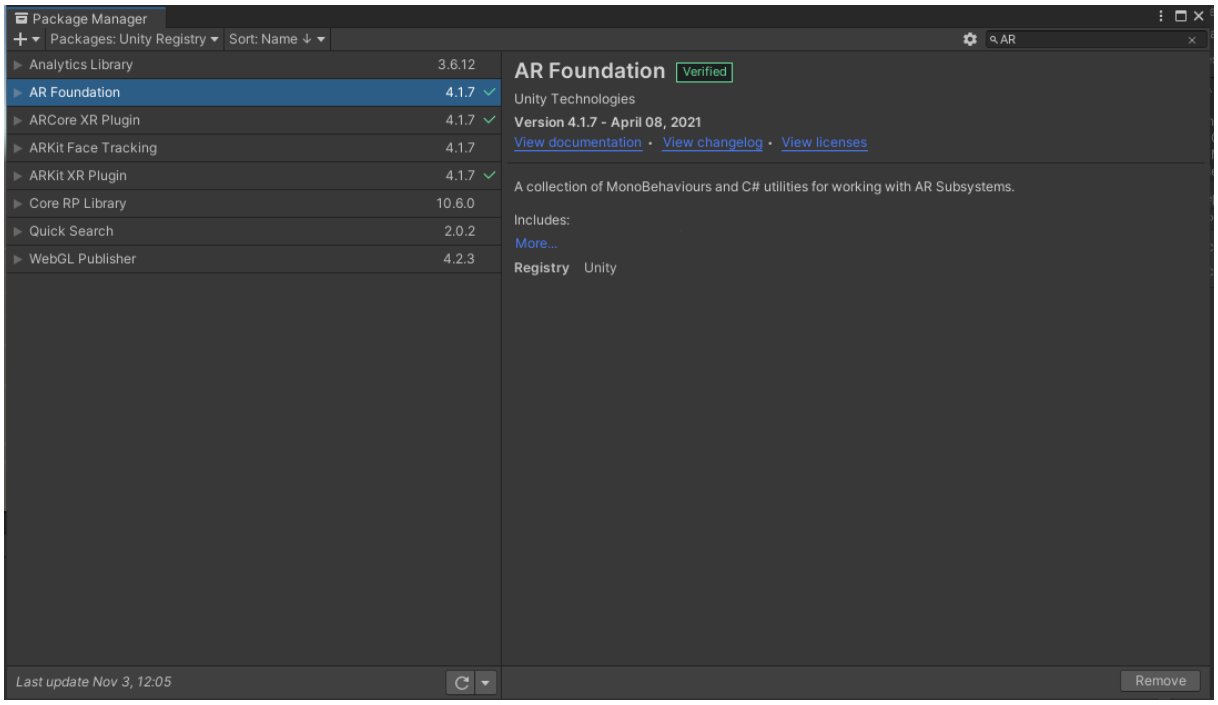
Create an AR application made of two different scenes. Each scene will introduce different Unity and development elements.
The exercise will be mainly based on the AR Foundation Unity framework, as it allows developers to build AR applications for Android, iOS and others.
The final application will be built to be run on Android.

Learn how to make a basic AR application.
Learn to instantiate game objects.
Learn about and use Raycasts.
Learn about and create a GUI.

Please make sure to read the instructions; do NOT just copy the screenshots.

Scene setup

In the package manager, be sure to have the AR Foundation library downloaded and installed into the unity project.



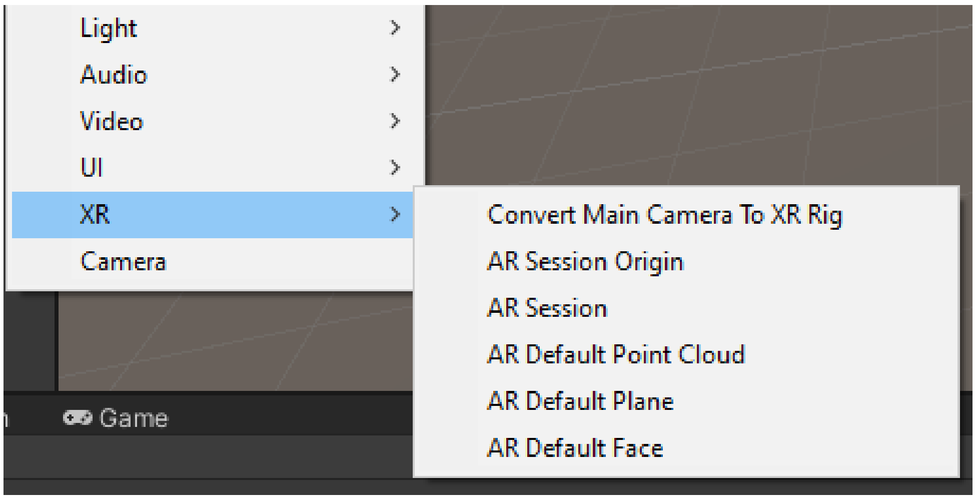
Analytics Library	3.6.12
AR Foundation	4.1.7 ✓
ARCore XR Plugin	4.1.7 ✓
ARKit Face Tracking	4.1.7
ARKit XR Plugin	4.1.7 ✓
Core RP Library	10.6.0
Quick Search	2.0.2
WebGL Publisher	4.2.3

AR Foundation Verified
Unity Technologies
Version 4.1.7 - April 08, 2021
View documentation • View changelog • View licenses

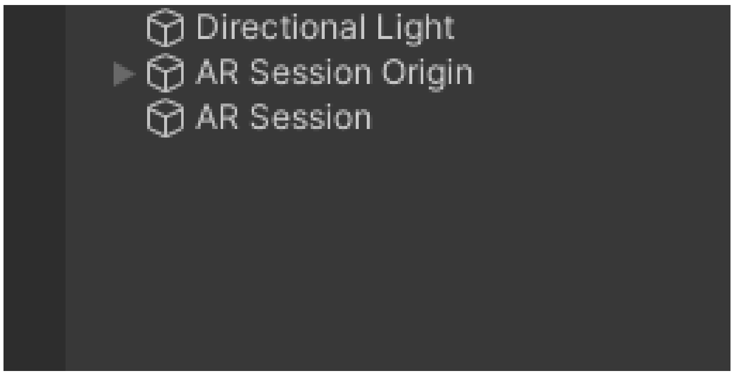
A collection of MonoBehaviours and C# utilities for working with AR Subsystems.

Includes:
More...
Registry Unity

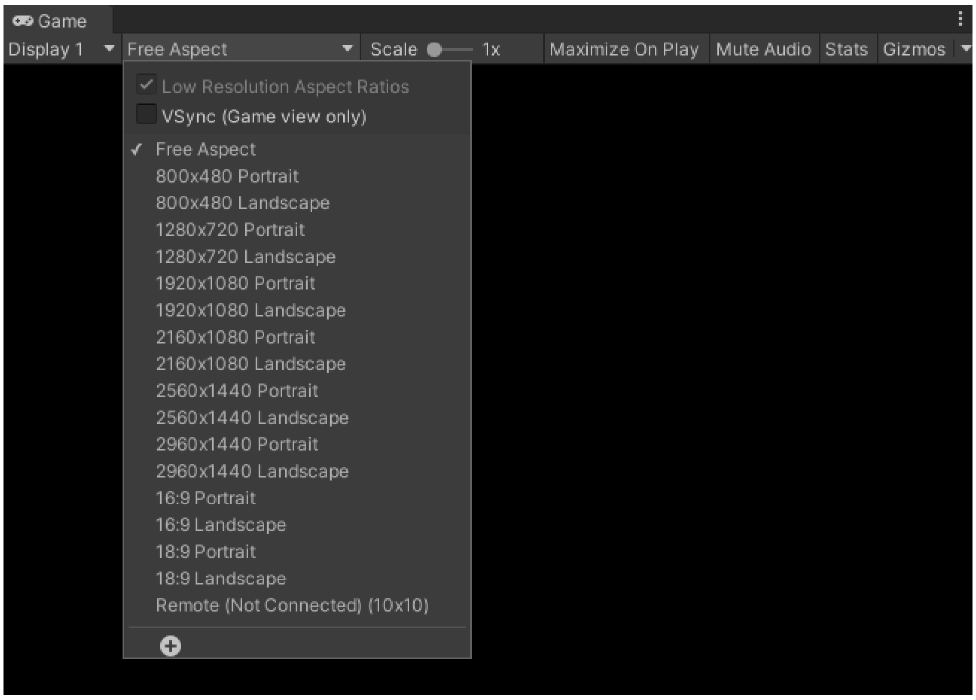
Once you have successfully installed AR Foundation, you can right click into the hierarchy and create an AR Session Origin and AR Session from the XR tab.



Delete the main camera by right clicking on it and click on Delete. If you've done everything correctly until now your scene hierarchy should look like this.



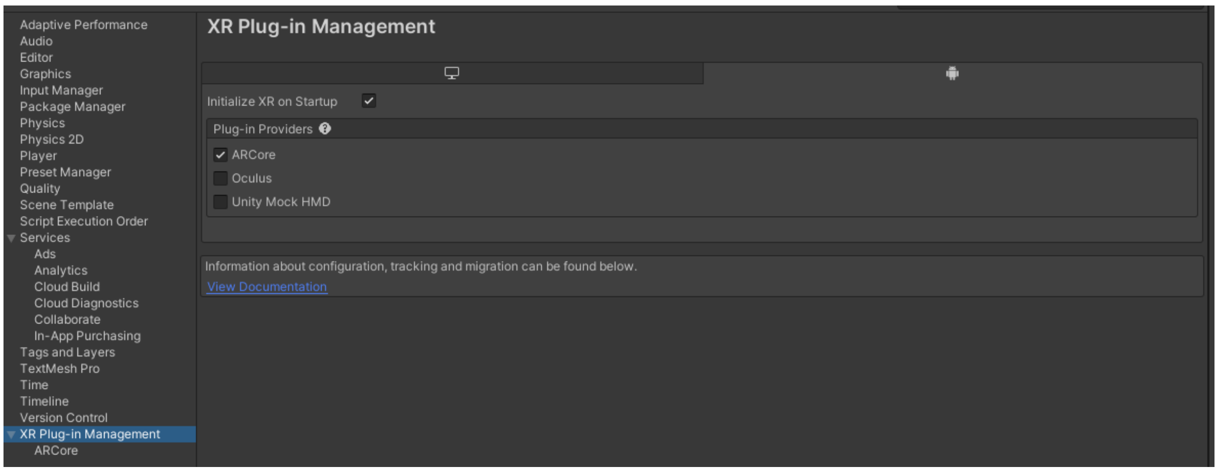
One useful feature we might want to use is the aspect ratio feature from the Game window.
In order to change this, you need to click on the 'Free Aspect' dropdown on the menu bar at the top of the Game window.
Since we will be working on your phone for this practical, we will set the aspect ratio similar to your phone resolution. You can either enter your exact phone resolution or choose a preset close enough:



Project Settings

There are multiple things to set up in the project settings.

Under 'XR Plug-in Management', then under the Android tab, make sure ARCore is ticked, like so:



Adaptive Performance
Audio
Editor
Graphics
Input Manager
Package Manager
Physics
Physics 2D
Player
Preset Manager
Quality
Scene Template
Script Execution Order
Services
- Ads
- Analytics
- Cloud Build
- Cloud Diagnostics
- Collaborate
- In-App Purchasing

Tags and Layers
TextMesh Pro
Time
Timeline
Version Control
XR Plug-in Management
- ARCore

XR Plug-in Management

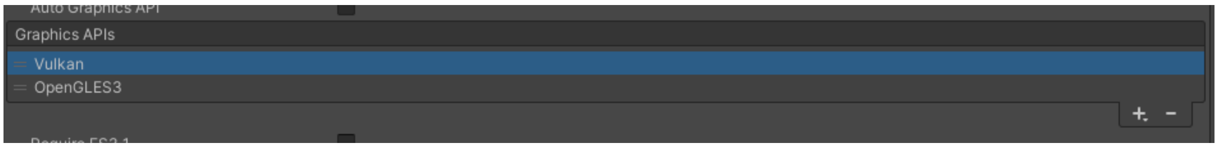
Initialize XR on Startup ✓

Plug-in Providers

- ✓ ARCore
- Oculus
- Unity Mock HMD

Information about configuration, tracking and migration can be found below.
View Documentation

Under 'Player', then under the Android tab, and then under 'Other Settings', there are two things to change: First, remove 'Vulkan' from 'Graphics APIs' by selecting it and clicking on the minus sign at the bottom right:



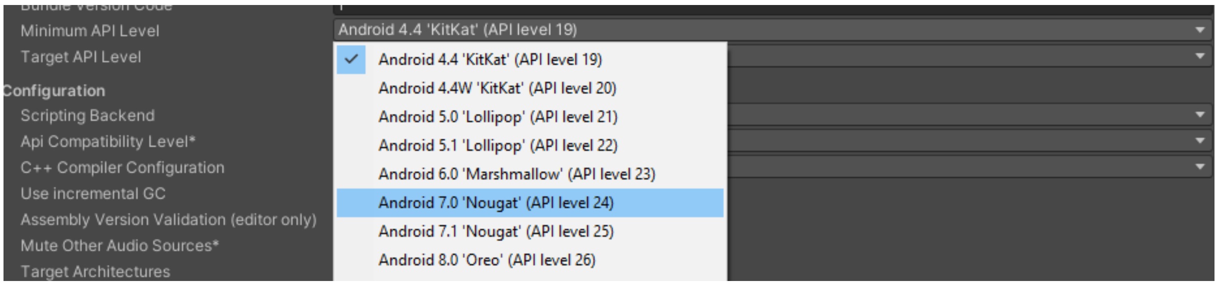
Auto Graphics API

Graphics APIs

- Vulkan
- OpenGLES3

If you don't see the 'Graphics APIs' option, you first need to un-tick 'Auto Graphics API'. This should make the option appear.
In case there is no 'Vulkan' in the list, you can skip this step.

Finally, change 'Minimum API Level' from API level 19 to API level 24:



Minimum API Level	Android 4.4 'KitKat' (API level 19)
Target API Level	
Configuration	
Scripting Backend	
Api Compatibility Level*	
C++ Compiler Configuration	
Use incremental GC	
Assembly Version Validation (editor only)	
Mute Other Audio Sources*	
Target Architectures	

- ✓ Android 4.4 'KitKat' (API level 19)
- Android 4.4W 'KitKat' (API level 20)
- Android 5.0 'Lollipop' (API level 21)
- Android 5.1 'Lollipop' (API level 22)
- Android 6.0 'Marshmallow' (API level 23)
- Android 7.0 'Nougat' (API level 24)
- Android 7.1 'Nougat' (API level 25)
- Android 8.0 'Oreo' (API level 26)

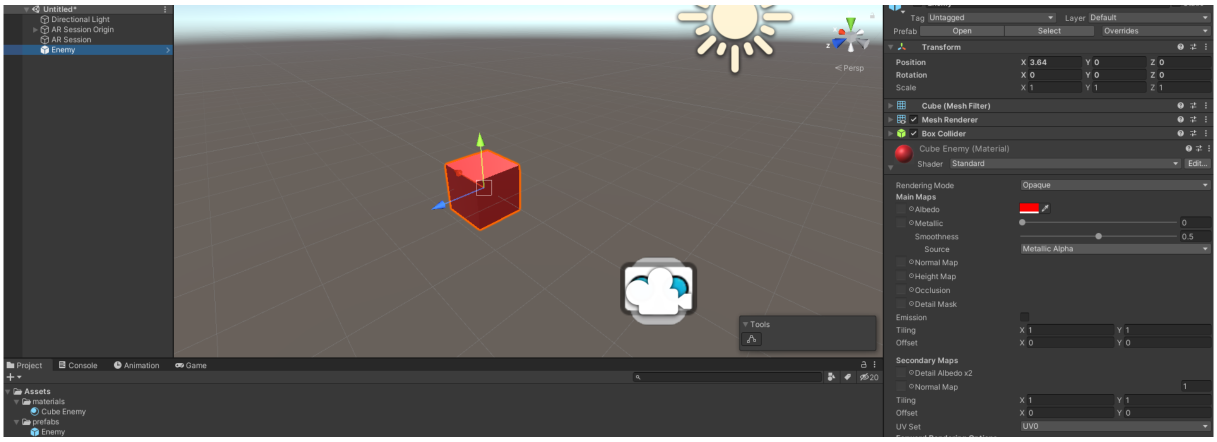
Phone setup

First, you need to make sure your phone supports ARCore.
Head over to THIS page and look for your phone in the list. If it's not here, your phone won't be able to execute our application. In that case, you can either ask for someone else's phone when you need to build the app, or get into another group with a valid phone.

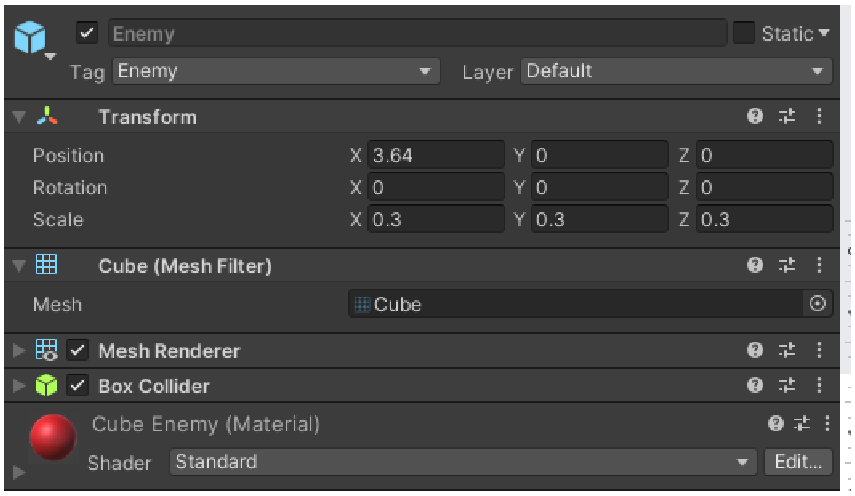
You will also need to put your phone in Developer mode and activate USB debugging. The steps to achieve this depend on your phone, so you have to look it up online. Usually you will find the USB debugging option under the Developer Options (once in Developer Mode).

Scene 1

In this scene, we want to create a minimalist application where cubes spawn around the player to be killed when they are clicked upon. We will start by creating a new material that will be used for our cubes, from this material, create the following cube prefab.



Since this cube will appear in AR it's default scale of (1,1,1) might be a bit huge. Let's reduce it to 0.3.



Enemy — Static
Tag: Enemy — Layer: Default

Transform
Position: X 3.64, Y 0, Z 0
Rotation: X 0, Y 0, Z 0
Scale: X 0.3, Y 0.3, Z 0.3

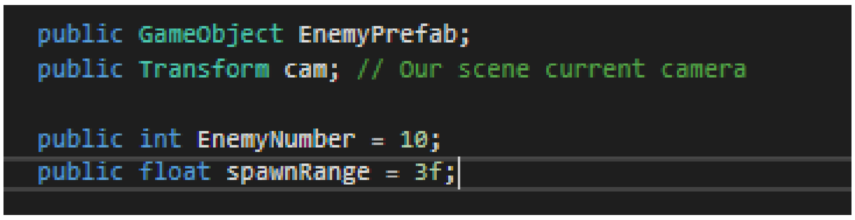
Cube (Mesh Filter)
Mesh: Cube

Mesh Renderer
Box Collider

Cube Enemy (Material)
Shader: Standard — Edit...

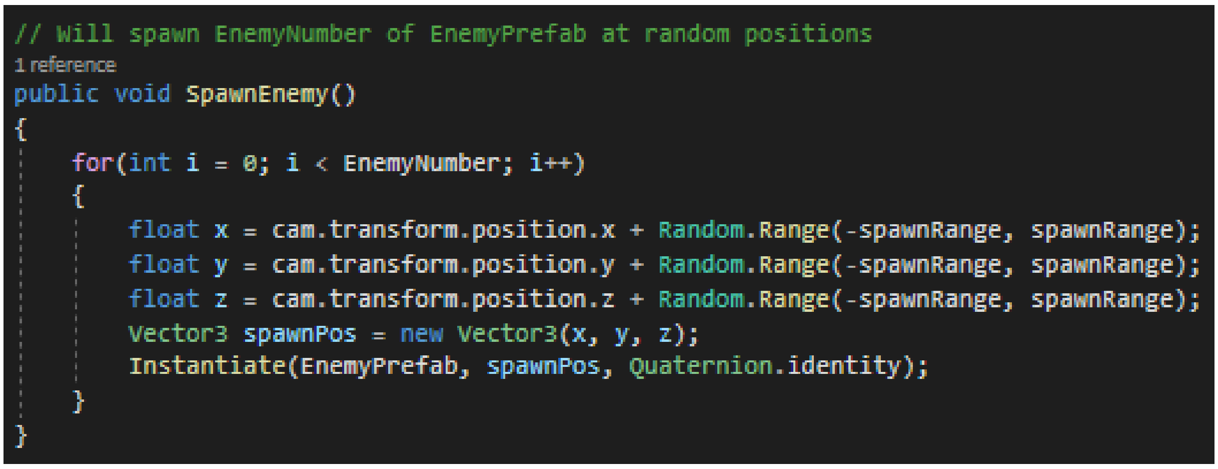
Application Manager

Our next step is to create a script named ApplicationManager. This script will handle the instantiation of our cubes with two customizable parameters, the enemy number, and the spawn range.On top of our two customizable parameters, our script will need our EnemyPrefab we just created and the Transform of our current camera to instantiate enemies around it.



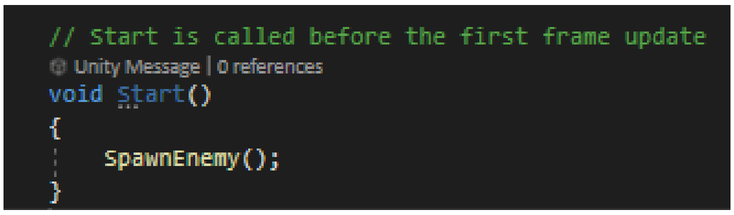
```
public GameObject EnemyPrefab;
public Transform cam; // Our scene current camera

public int EnemyNumber = 10;
public float spawnRange = 3f;
```

Our next goal is to instantiate our enemies around our camera. To do so, we will conceive a method called SpawnEnemy, in which we will instantiate as many enemies as our EnemyNumber current value.



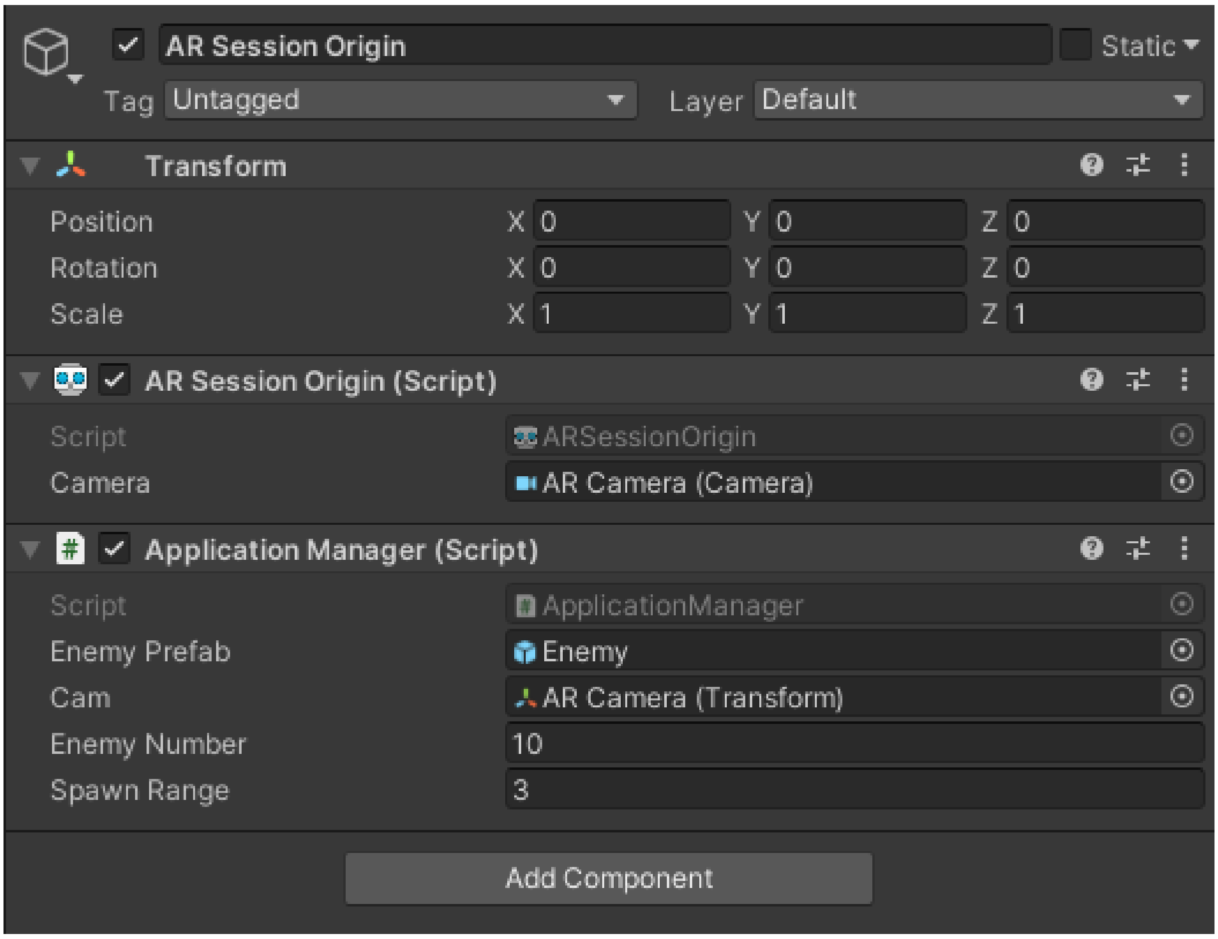
```
// Will spawn EnemyNumber of EnemyPrefab at random positions
1 reference
public void SpawnEnemy()
{
    for(int i = 0; i < EnemyNumber; i++)
    {
        float x = cam.transform.position.x + Random.Range(-spawnRange, spawnRange);
        float y = cam.transform.position.y + Random.Range(-spawnRange, spawnRange);
        float z = cam.transform.position.z + Random.Range(-spawnRange, spawnRange);
        Vector3 spawnPos = new Vector3(x, y, z);
        Instantiate(EnemyPrefab, spawnPos, Quaternion.identity);
    }
}
```

To end our script, we will call SpawnEnemy into the Start method. This way our cubes will be generated as soon as our script is started.



```
// Start is called before the first frame update
Unity Message | 0 references
void Start()
{
    SpawnEnemy();
}
```

Now we're gonna attach our newly created script to the AR Session Origin.



AR Session Origin — Static
Tag: Untagged — Layer: Default

Transform

	X	Y	Z
Position	0	0	0
Rotation	0	0	0
Scale	1	1	1

AR Session Origin (Script)

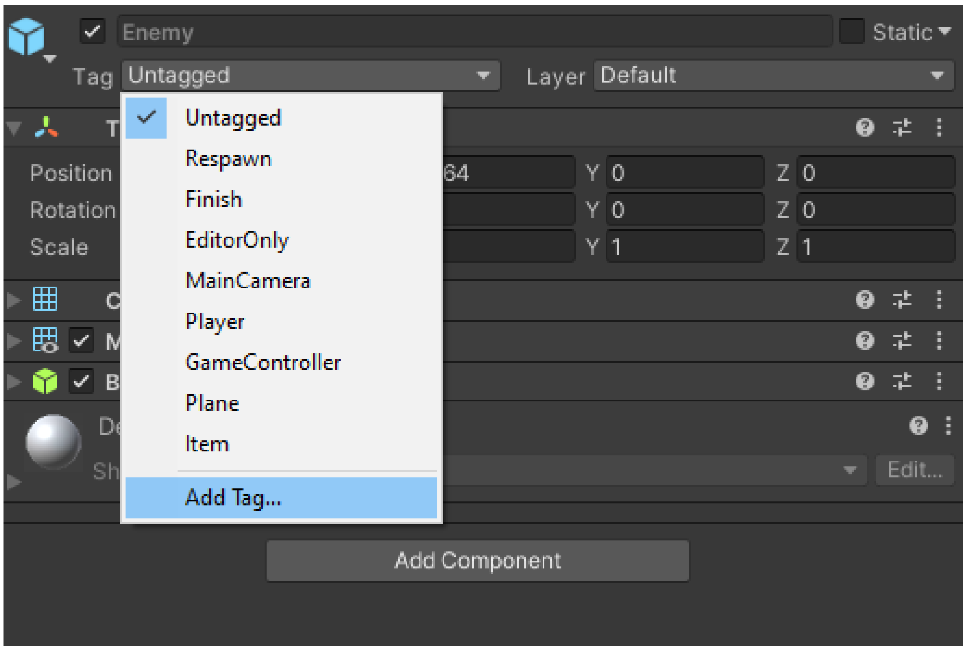
Script	ARSessionOrigin
Camera	AR Camera (Camera)

Application Manager (Script)

Script	ApplicationManager
Enemy Prefab	Enemy
Cam	AR Camera (Transform)
Enemy Number	10
Spawn Range	3

By entering play mode you can check if everything works correctly. Before worrying about killing our enemies, we have to make sure we can recognize them from any other gameObject. One way to achieve that is to give them a tag.

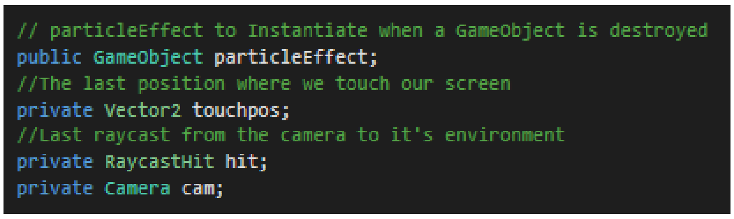
What is a tag ? it's a string we assign to one or multiple GameObject. Doing so allows us to check if a GameObject we clicked on belongs to the enemy category or not. Create a tag named "Enemy" and assign it to the prefab.



Killing Camera

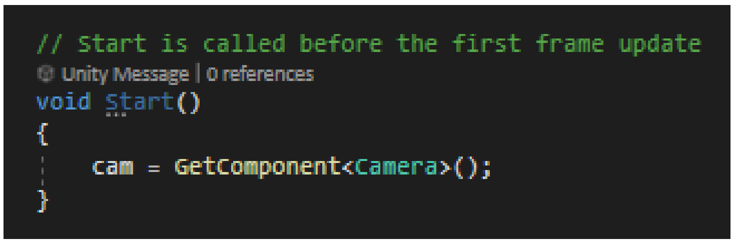
The following step is to create a KillingCam script that will allow our player to kill cubes by touching them on the screen. For our script we will use a particle effect to add some fun when a cube gets destroyed. Let's get a particleEffect GameObject in our script as well as 3 private variables :

1. a Vector2 named touchpos that we will use to get our screen touch position
2. a RayCastHit named hit who represent the ray hit between our camera and the target of our screen touch
3. a Camera named cam who is our Camera object that we will use to transform the screen touch position to a ray



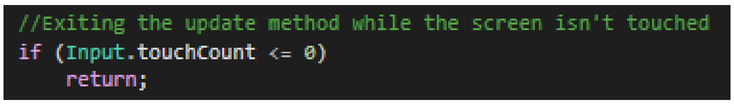
```
// particleEffect to Instantiate when a GameObject is destroyed
public GameObject particleEffect;
//The last position where we touch our screen
private Vector2 touchpos;
//Last raycast from the camera to it's environment
private RaycastHit hit;
private Camera cam;
```

Since this script is meant to be assigned to the AR Camera, we can fill the cam variable in the Start method by getting the Camera Component.



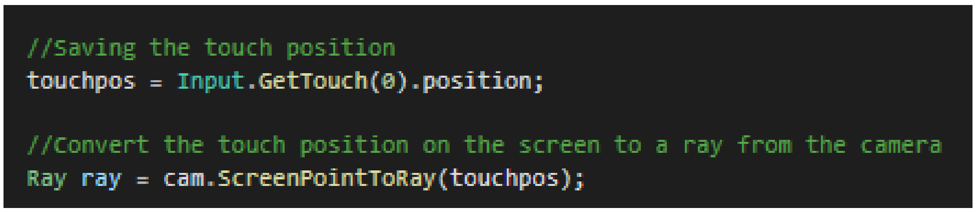
```
// Start is called before the first frame update
Unity Message | 0 references
void Start()
{
    cam = GetComponent<Camera>();
}
```

There is now only the update method left. Firstly we will use the Input class to detect when the screen is touched. Since we don't want our script to do any action at all while the screen isn't touched we will simply exit the update method when the touchCount isn't greater than 0.



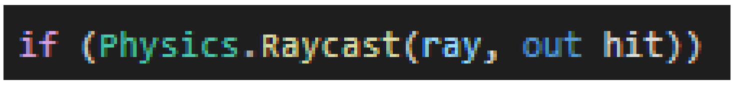
```
//Exiting the update method while the screen isn't touched
if (Input.touchCount <= 0)
    return;
```

Once our screen is touched, we want to save the touch position of the first touch on the screen using GetTouch. Using this newly obtained position we can convert our screen touch position to a ray from the camera.



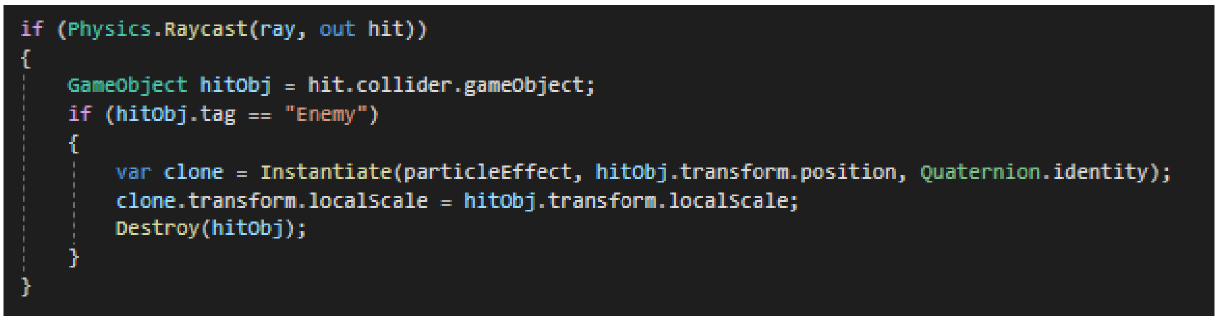
```
//Saving the touch position
touchpos = Input.GetTouch(0).position;

//Convert the touch position on the screen to a ray from the camera
Ray ray = cam.ScreenPointToRay(touchpos);
```

This time, we want Unity to help us check if our ray collided with an object. To perform that, we will use Physics.Raycast. Along with checking if our ray collided with objects it will store into "hit" extra information about the collided gameobject.



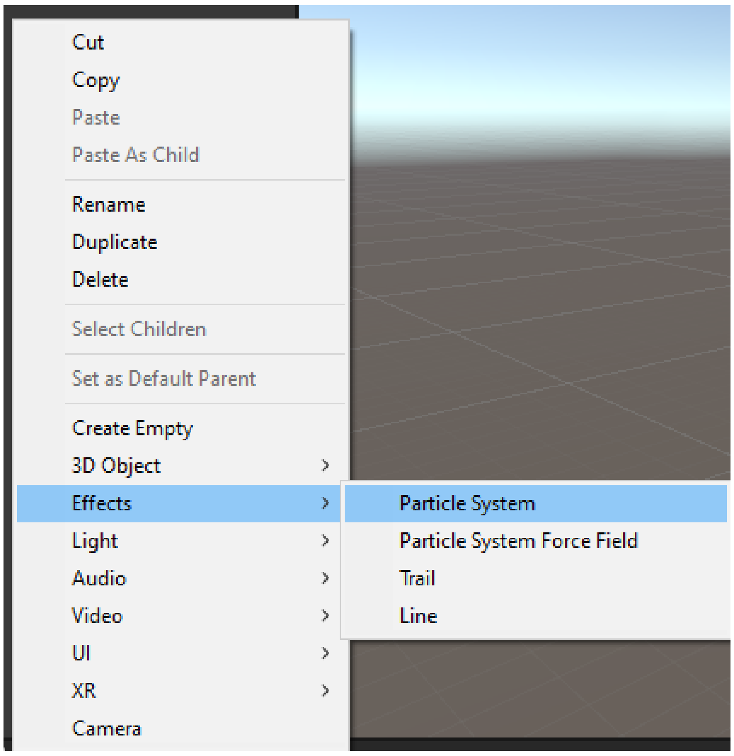
```
if (Physics.Raycast(ray, out hit))
```

From hit, we can get the GameObject of the collided gameobject. Now let's check if it's tag corresponds to our enemies category. If it does we can Instantiate our particleEffect GameObject and destroy the collided gameobject.

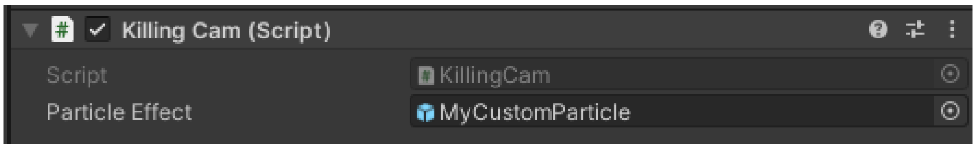


```
if (Physics.Raycast(ray, out hit))
{
    GameObject hitObj = hit.collider.gameObject;
    if (hitObj.tag == "Enemy")
    {
        var clone = Instantiate(particleEffect, hitObj.transform.position, Quaternion.identity);
        clone.transform.localScale = hitObj.transform.localScale;
        Destroy(hitObj);
    }
}
```

Let's assign our KillingCam script to our AR Camera. Once done, Our last step is to create a particle effect to play once our cubes are destroyed. To achieve that we will create a new Particle System from the Effect category on the GameObject creation menu.



Play with the particle system parameters to create a particle to your liking and assign it to your KillingCam script into the AR Camera. Don't forget to make a prefab of the particle system before assigning it. Also, remove it from the hierarchy!



Killing Cam (Script)

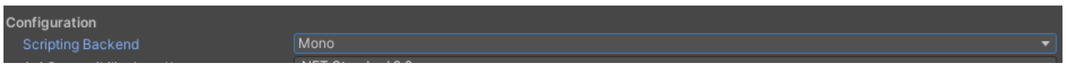
Script KillingCam

Particle Effect MyCustomParticle

Once it's done, you can verify that you have no errors by entering Play mode. If everything works correctly, you can finish this scene by building and running it on your android (File > Build Settings > Make sure everything is alright > Build And Run > Put the .apk somewhere on your COMPUTER, the build will copy it automatically to your phone).

Fixing Eventual Crashes

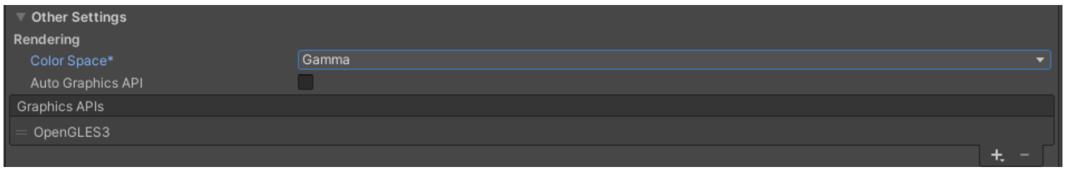
If your application crashes on start-up, here are some tips to hopefully fix the problem: First, you will need to go back to the Project Settings, and then in the Player sub-menu. Select the Android tab, and navigate to the 'Scripting Backend' option:


Configuration
Scripting Backend Mono

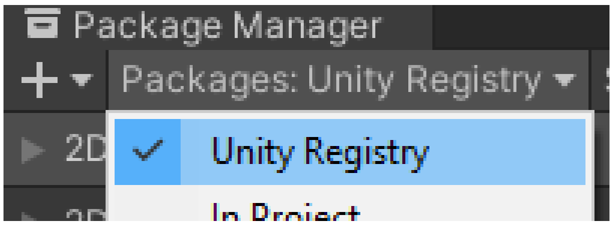
Change it from Mono to IL2CPP. Then, slightly lower there is the 'Target Architectures' option. Make sure both ARMv7 and ARM64 are checked. By default ARM64 is unchecked, but in my experience it needed to be to run on my phone.

You can now build your application again. It's going to take slightly longer than with the Mono scripting backend, but IL2CPP actually makes your application faster and is the recommended scripting backend for releases.

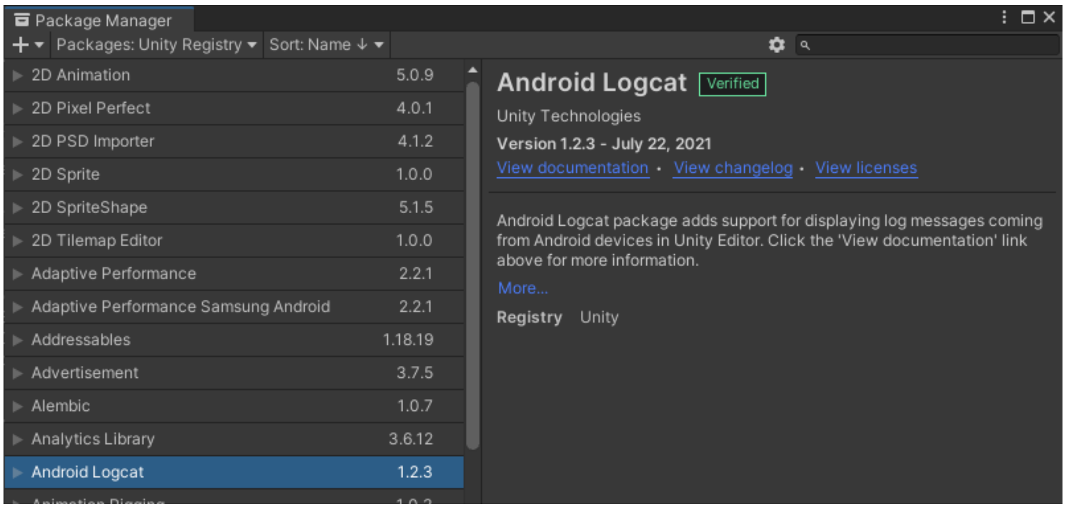
If the application still crashes, there is not much more I can help you with. Some people online have mentioned switching the 'Color Space' option, but that's about it.


Other Settings
Rendering
Color Space* Gamma
Auto Graphics API
Graphics APIs
OpenGLES3

If you are willing to, you can install an additional package that will allow you to see your app's logs and then search on internet information related to the error message; here is how to do this. Go to the Package Manager, and at the top, select Unity Registry:



Then, search for and import the Android Logcat package:



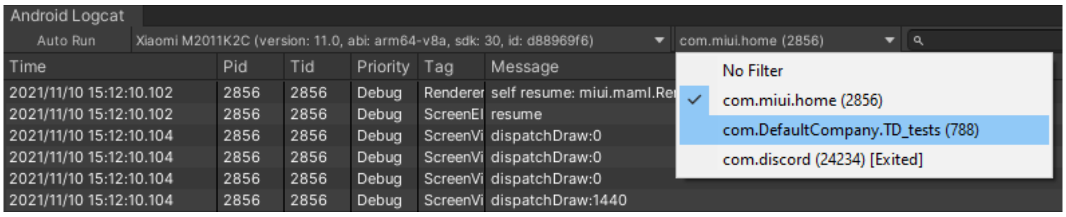
2D Animation	5.0.9
2D Pixel Perfect	4.0.1
2D PSD Importer	4.1.2
2D Sprite	1.0.0
2D SpriteShape	5.1.5
2D Tilemap Editor	1.0.0
Adaptive Performance	2.2.1
Adaptive Performance Samsung Android	2.2.1
Addressables	1.18.19
Advertisement	3.7.5
Alembic	1.0.7
Analytics Library	3.6.12
Android Logcat	1.2.3

Android Logcat Verified
Unity Technologies
Version 1.2.3 - July 22, 2021
View documentation • View changelog • View licenses

Android Logcat package adds support for displaying log messages coming from Android devices in Unity Editor. Click the 'View documentation' link above for more information.
More...
Registry Unity

This will allow you to see the crash reports of android devices plugged into your computer. In order to use that package to debug your application, refer to THIS guide.

However, simply put, you'll want to select your device, then start your application, and finally select your app from the 'Filter' drop-down:


Android Logcat
Auto Run
Xiaomi M2011K2C (version: 11.0, abi: arm64-v8a, sdk: 30, id: d88969f6)
com.miui.home (2856)
No Filter
✓ com.miui.home (2856)
com.DefaultCompany.TD_tests (788)
com.discord (24234) [Exited]

```
Time                     Pid   Tid   Priority Tag      Message
2021/11/10 15:12:10.102  2856  2856  Debug    Renderer self resume: miui.maml.Re
2021/11/10 15:12:10.102  2856  2856  Debug    ScreenEl resume
2021/11/10 15:12:10.104  2856  2856  Debug    ScreenVi dispatchDraw:0
2021/11/10 15:12:10.104  2856  2856  Debug    ScreenVi dispatchDraw:0
2021/11/10 15:12:10.104  2856  2856  Debug    ScreenVi dispatchDraw:0
2021/11/10 15:12:10.104  2856  2856  Debug    ScreenVi dispatchDraw:1440
```

Now, look for red messages, they are errors. Try to find the start of the error and then do some research on the internet. Good luck!

Scene 2

In this scene, we will aim to create a simple AR experience where you can instantiate cubes in your environment by touching your screen. It is going to use plane detection to detect your environment in a smart way. You will also be able to change their color at the push of a button.

Firstly, you need to create a new Scene. Simply head into the Scenes folder, right click and create a new scene. Rename it Scene2, open it, and add an AR Session and an AR Session Origin. You don't need to re-import the packages though, as they have been imported for the whole project.

Planes Detection

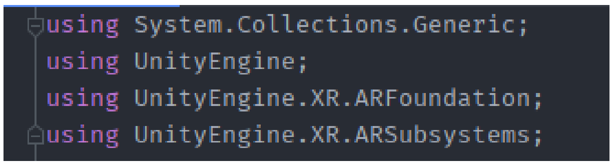
- In order to detect planes, add an AR Plane Manager component to the AR Session Origin.
- Right click in the hierarchy, then click on AR Default Plane under the XR sub-menu.
- Drag the AR Default Plane in the Prefabs folder, then delete it from the hierarchy.
- Finally, drag the new prefab into the Plane Prefab field of the AR Plane Manager component.
- In the build settings, click on Add Open Scenes, then make sure Scene2 is the ONLY scene TICKED. This will build only our current scene and will ignore our previous one.
- Like earlier, make sure everything is fine and then click on Build or Build And Run.

You should see the planes being detected (it might take a couple of seconds, and/or be completely wonky; it just works.). What is happening is the application is detecting planes and creating GameObjects to represent them.

Instantiating cubes

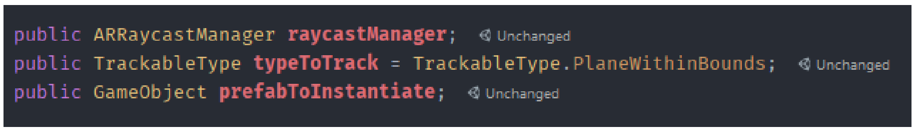
- Create a cube and save it as a prefab. You should know how to do this already, but if you don't, read earlier steps. This is something you have already done multiple times. Don't forget to remove the original from the hierarchy, we won't need it.
- Also create a new material for the cube, and set its color to whatever you want.
- Open the prefab, and set its scale to 0.2 in X, Y and Z.
- Add an AR Raycast Manager component to the AR Session Origin.
- Create an empty GameObject in the hierarchy, name it Manager.
- Create a new Script named Scene2Manager, attach it to the empty GameObject.

Open the script. We are going to edit it. What we want is to detect when the user touches the screen, and if the user has hit one of the detected planes, we want to instantiate our prefab there. Let's start with the "using" statements:



```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;
```

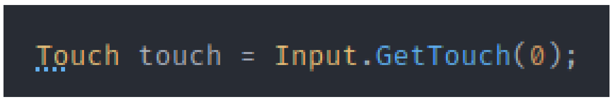
These are nothing special, the first one is to be able to use common data structures, and the others are to use common Unity features and more advanced ones relative to the AR Foundation.
Now, let's edit our class. We are going to need three public variables.



```
public ARRaycastManager raycastManager;   Unchanged
public TrackableType typeToTrack = TrackableType.PlaneWithinBounds;   Unchanged
public GameObject prefabToInstantiate;   Unchanged
```

The first two will be used to know where in the scene the user is currently touching, and the last one is going to be our prefab. We will give the three public variables values later in the Unity Editor.

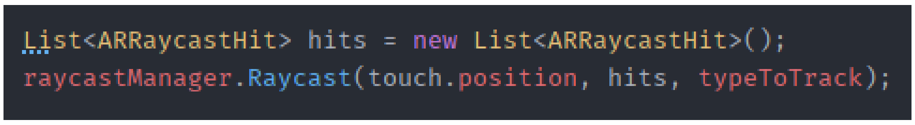
Next, we will create the private OnTouch method. It is going to contain what we want to execute whenever we detect touch. It returns void, and this time there is no screenshot to guide you. In order to get information about the user's touch, here is what you have to do:



```
Touch touch = Input.GetTouch(0);
```

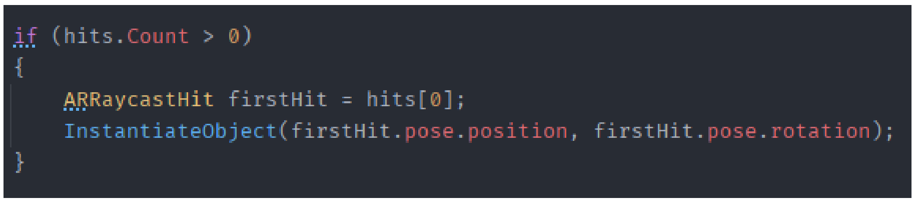
This will allow us to access where on the screen the touch happened. Here, we only want to get the first touch, hence the 0 as parameter.

Then, we need to project the touch on the screen into the scene. For that, we are going to use raycasts. We will use an AR Raycast, which is a special type of raycast.



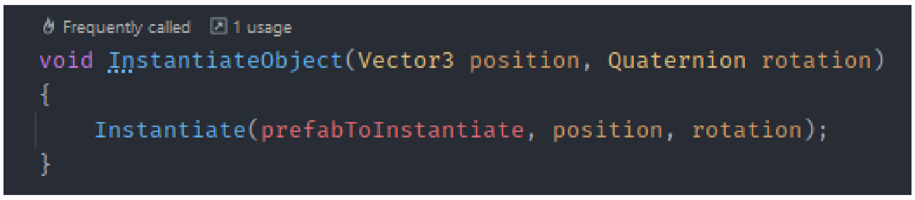
```
List<ARRaycastHit> hits = new List<ARRaycastHit>();
raycastManager.Raycast(touch.position, hits, typeToTrack);
```

First, we create an empty list of ARRaycastHit. This list is populated by the raycast if it hits objects of the type typeToTrack. Every ARRaycastHit then contains information about one of these objects. In our case, we only care about the first hit, so if there is any hit in the list, we want to instantiate our prefab on the position of the first object hit in the list:



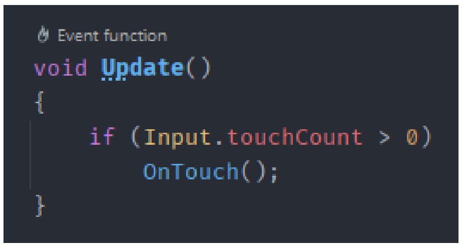
```
if (hits.Count > 0)
{
    ARRaycastHit firstHit = hits[0];
    InstantiateObject(firstHit.pose.position, firstHit.pose.rotation);
}
```

We have not created the InstantiateObject method, so do not worry if it gives you an error; we are going to create it.
It's a fairly simple method as it mostly acts as a wrapper around the method classically used to instantiate an object. Using such a wrapper will allow us to add more behavior to it later down the road. Here is the method:



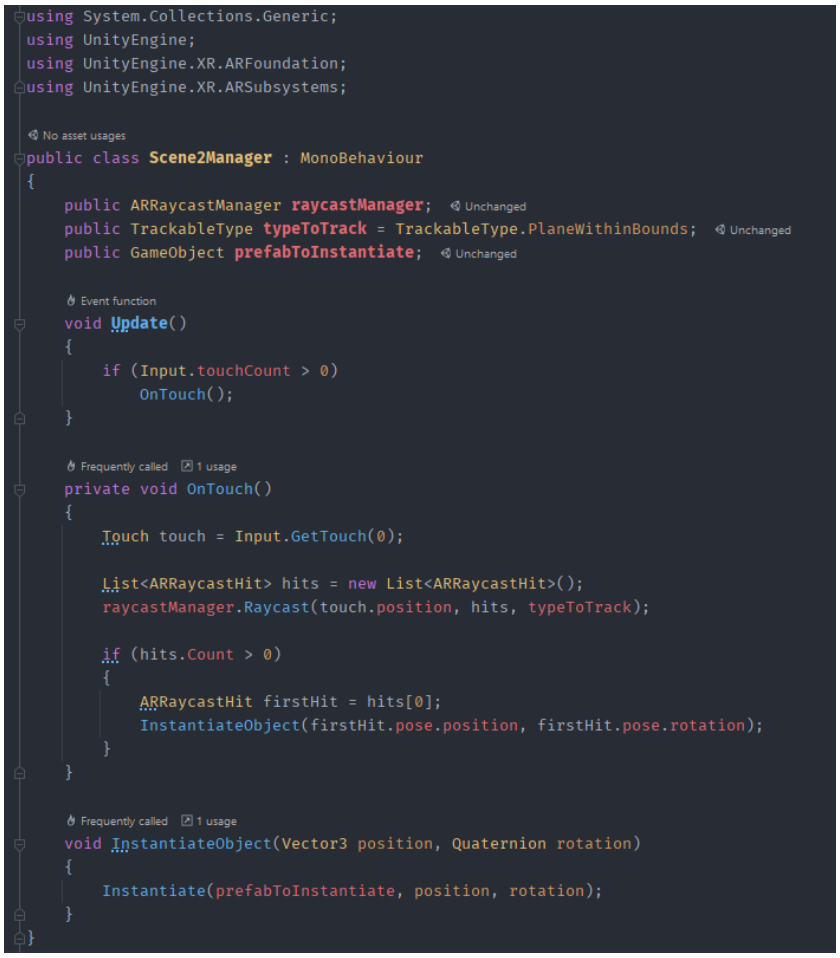
```
Frequently called  1 usage
void InstantiateObject(Vector3 position, Quaternion rotation)
{
    Instantiate(prefabToInstantiate, position, rotation);
}
```

Finally, we only need to call OnTouch whenever we detect touch. To do that, write these two lines in your Update method:



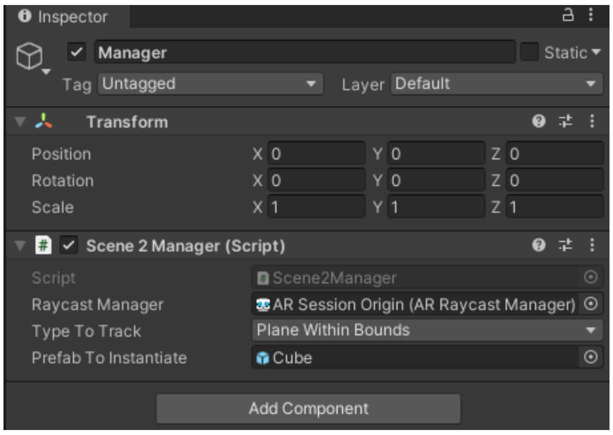
```
Event function
void Update()
{
    if (Input.touchCount > 0)
        OnTouch();
}
```

Don't forget to save the script! It should look something like this:



```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.XR.ARFoundation;
using UnityEngine.XR.ARSubsystems;

public class Scene2Manager : MonoBehaviour
{
    public ARRaycastManager raycastManager;
    public TrackableType typeToTrack = TrackableType.PlaneWithinBounds;
    public GameObject prefabToInstantiate;

    void Update()
    {
        if (Input.touchCount > 0)
            OnTouch();
    }

    private void OnTouch()
    {
        Touch touch = Input.GetTouch(0);

        List<ARRaycastHit> hits = new List<ARRaycastHit>();
        raycastManager.Raycast(touch.position, hits, typeToTrack);

        if (hits.Count > 0)
        {
            ARRaycastHit firstHit = hits[0];
            InstantiateObject(firstHit.pose.position, firstHit.pose.rotation);
        }
    }

    void InstantiateObject(Vector3 position, Quaternion rotation)
    {
        Instantiate(prefabToInstantiate, position, rotation);
    }
}
```

Now, head back into Unity.
Select the Manager GameObject in the hierarchy. As you can see, there are empty fields under your script in the Inspector.
Drag and drop the AR Session Origin into the "Raycast Manager" field. It's going to automatically assign its AR Raycast Manager component.
Drag and drop your cube prefab into the "Prefab To Instantiate" field. Your Manager's components should look like this:



Inspector
Manager — Static
Tag: Untagged — Layer: Default

Transform
Position: X 0, Y 0, Z 0
Rotation: X 0, Y 0, Z 0
Scale: X 1, Y 1, Z 1

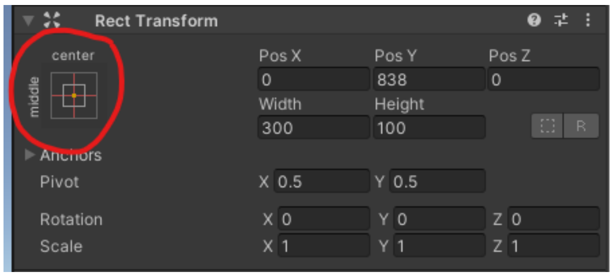
Scene 2 Manager (Script)
Script: Scene2Manager
Raycast Manager: AR Session Origin (AR Raycast Manager)
Type To Track: Plane Within Bounds
Prefab To Instantiate: Cube

You can now Build and Run your application. Whenever you touch one of the detected planes, your cube prefab should be instantiated on it. If you feel the cube is too big, open the prefab and change its scale just like you did before.

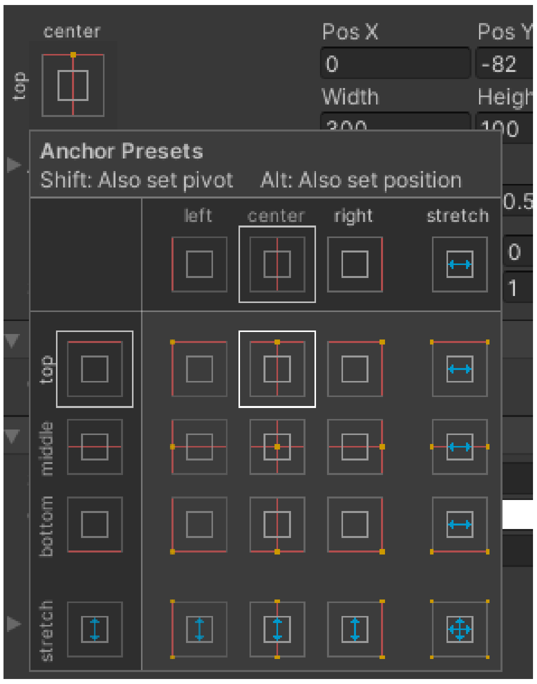
Creating an UI

- Right click in the Hierarchy window and create a Canvas (under the UI sub-menu).
- Right click on the Canvas and create a Button (under the same sub-menu as before).
- To see the UI correctly in the Scene view, make sure to select it in the hierarchy, put your mouse in the Scene window and press F once or twice to center the view on the UI.
- Notice the Canvas Scaler component in the Canvas' components. Change its UI Scale Mode to "Scale With Screen Size".
- Change the Reference Resolution to the resolution of your phone's screen. If you don't know it or don't want to bother checking, set it to X: 1080 Y: 1920.
- There is a slider right under the Reference Resolution. Slide it to the right, so that it is set on "Height".
- Move the new Button wherever you see fit. You can also change its size by modifying its Width and Height values. It contains a Text child, which's text you can change to whatever you want. Keep in mind that this button is supposed to change the color of the instantiated objects.

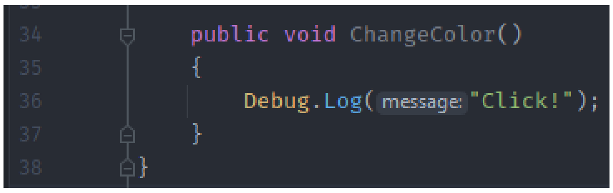
Select the button, and in the inspector, click on the part circled in red in the following screenshot:



This will allow you to set the anchor for the button, that is which side of the canvas it should try to stick to when it gets automatically resized. For instance, if you wanted to set it to the top, here is how you would do it:

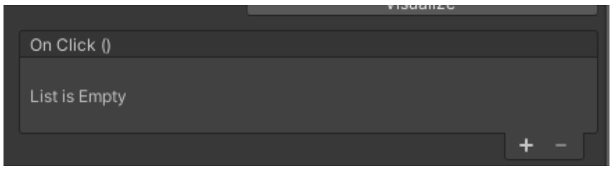


As you can see at the top of this picture, there are plenty of other controls you can use, to stretch the button or set its position on its anchor, etc.
At this point, if you enter Play-mode, you should be able to click on the button and see it react to your cursor. However, nothing happens yet when you press it. Let's change that!
Go back to the script we edited earlier. We are going to add a small function:



```
    public void ChangeColor()
    {
        Debug.Log( message: "Click!");
    }
}
```

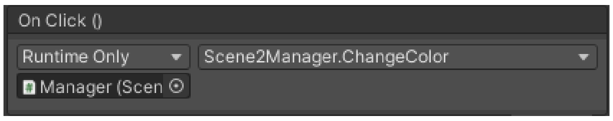
This will only print "Click!" to the Unity Console; we will come back later to add some more code to it. Make sure to add it at the bottom of the Scene2Manager class, not at the bottom of the file. Don't forget to save it.
Head back to Unity. Select the button, and you should see the following at the bottom of its Button component:



On Click ()

List is Empty

Click on the + at the bottom right.
Drag and drop the Manager GameObject onto the "None (Object)" field. Now, you should be able to select our "ChangeColor" function on the top-right field, like so:



On Click ()

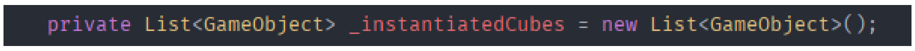
Runtime Only | Scene2Manager.ChangeColor

Manager (Scen

Enter Play-mode, and press the button. You should see "Click!" appear in the console window!

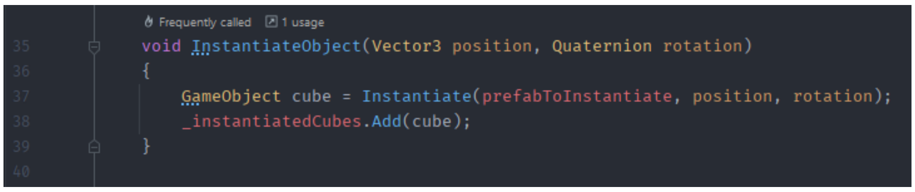
Now that we know the button is working, we can add more behavior to it. Let's go back to the script. We are going to add some code, but also change some of our old.

Let's add a new private variable to the class. It is going to store a list of all our instantiated cubes. In this scene, we won't destroy them, so we won't need to implement a way to remove them from this list.



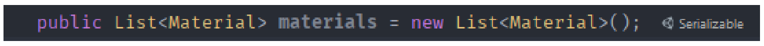
```
private List<GameObject> _instantiatedCubes = new List<GameObject>();
```

Now, we need to change our InstantiateObject function to add every instantiated object to our list.



```
Frequently called  1 usage
void InstantiateObject(Vector3 position, Quaternion rotation)
{
    GameObject cube = Instantiate(prefabToInstantiate, position, rotation);
    _instantiatedCubes.Add(cube);
}
```

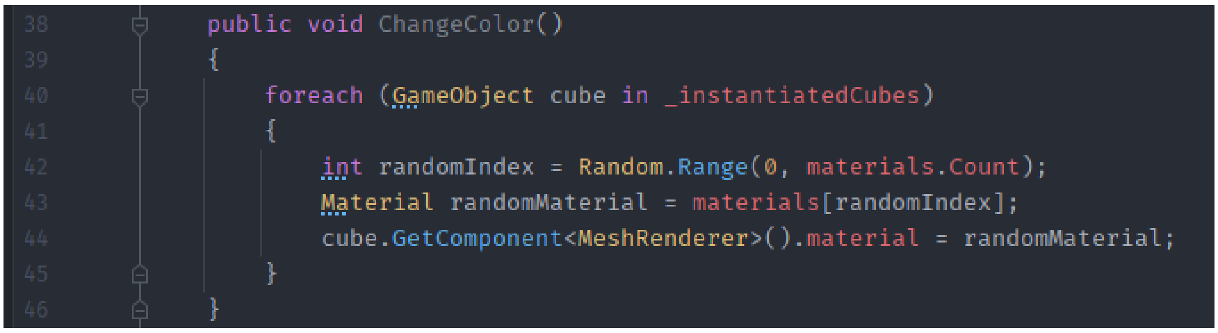
Let's also add a new public variable. We will need it to store materials, so let's make it a List of Material.



```
public List<Material> materials = new List<Material>(); Serializable
```

Now let's get to writing the new ChangeColor function! You can remove the Debug.Log we put in there earlier.

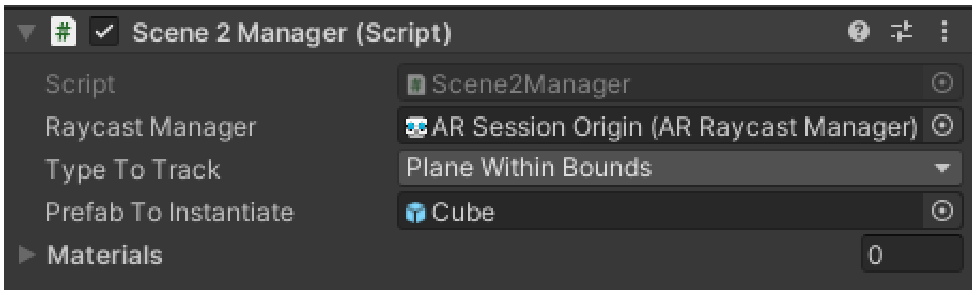
What we want to do is, for each cube in the list, we want to get a random material from our material list and assign it to its MeshRenderer component. Here is how we do it:



```
public void ChangeColor()
{
    foreach (GameObject cube in _instantiatedCubes)
    {
        int randomIndex = Random.Range(0, materials.Count);
        Material randomMaterial = materials[randomIndex];
        cube.GetComponent<MeshRenderer>().material = randomMaterial;
    }
}
```

For now, our materials list is empty. We could write a function that fills it with new materials with a random color, but we will go the easy route and create them by hand, and drag and drop them in the correct field. So, head back to Unity, and create a bunch of materials with different colors.

Now, drag and drop them on the “Materials” field of our script component on the Manager GameObject. If you cannot find it, it’s at the bottom of the component:



Scene 2 Manager (Script)

Script: Scene2Manager

Raycast Manager: AR Session Origin (AR Raycast Manager)

Type To Track: Plane Within Bounds

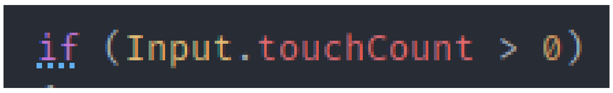
Prefab To Instantiate: Cube

Materials: 0

Unfortunately, if we wanna test our code, we will have to Build and Run our application. However, enter Play-mode first to make sure there are no errors.

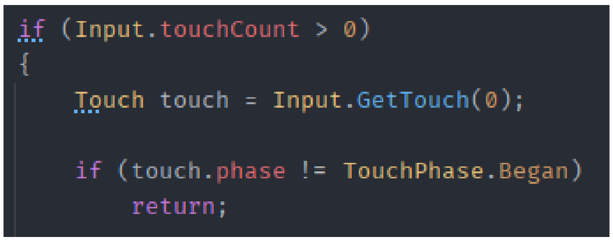
Cube counter

You might have noticed that cubes are created every frame as long as the user holds his touch. This is not the behavior we were expecting. The problem resides in that line of our code:



```
if (Input.touchCount > 0)
```

Every frame, so long as the screen is touched, we are instantiating an object at the touch's location. To change that, we just need to add a bit of code inside the body of the if statement:



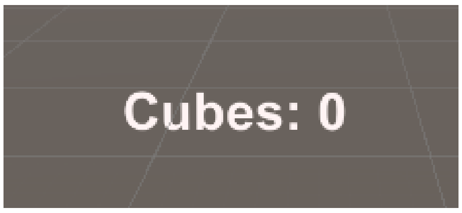
```
if (Input.touchCount > 0)
{
    Touch touch = Input.GetTouch(0);

    if (touch.phase != TouchPhase.Began)
        return;
```

The touch's phase will be set to TouchPhase.Began if this is the first frame the touch is detected; so if that's not the case, we don't want to do anything, hence the return statement.
What we just added could have also been a simple additional condition of the if statement, but we would have had to access Input.GetTouch(0) twice. It is cleaner that way. Feel free to save and test your application like this, however because this is such a small fix that we know is almost guaranteed to work, you might not want to waste some time building and then running the app, as we are about to add some more functionalities. It's up to you.

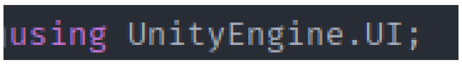
Anyway, now that our problem is fixed, let's add a new element to our UI, to make sure only one cube is instantiated at a time.

Head back into Unity, and add a new Text element to the Canvas. Make sure not to select TextMeshPro, as it is gonna cause additional imports to the project, and even though it would be more flexible than some straightforward text, we don't need such flexibility in our small application.

Just like the button, you can modify it however you want. Just make sure it is visible: remember that the background is going to be your camera feed! Also, keep in mind that it is supposed to keep track of the number of cubes in the scene.
Here is mine, for reference:

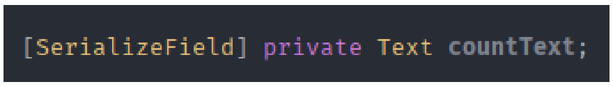


I put the text in Bold so that it stands out more. Note that it does not matter what you type into it for now, as we will be changing it via code later; however it is an accurate representation of how it is going to look.
Now, let's actually add the functionality to our code. Go back to the script.
Before diving into the code, let's import what we need:



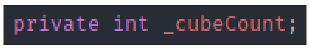
```
using UnityEngine.UI;
```

Firstly, we are going to need a new private reference to reference the text we just created. Just a side note though: public variables are accessible by any other script that can reference this one; but private variables don't appear in the Unity editor for us to give them a value. There is a solution, and it's the [SerializeField] attribute. Since the start of this exercise, none of the variables we created really needed to be public, we could have just made them private and used the [SerializeField] attribute if needed. Therefore, for the sake of exercise, we are going to use that attribute right now. This is how your variable should look like:



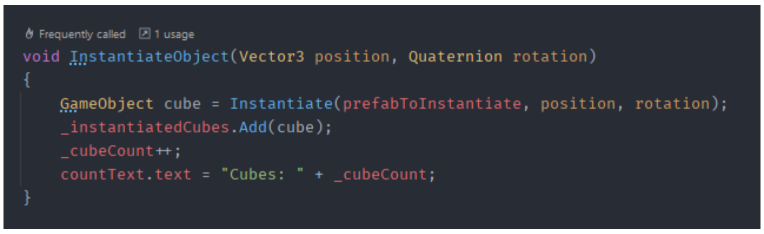
```
[SerializeField] private Text countText;
```

The attribute doesn't need to be aside the variable. It could also be on the line directly above it. Secondly, let's create another private variable to keep track of the count:



```
private int _cubeCount;
```

Thirdly, head to the InstantiateObject method. We will be adding some new lines of code:



```
Frequently called  1 usage
void InstantiateObject(Vector3 position, Quaternion rotation)
{
    GameObject cube = Instantiate(prefabToInstantiate, position, rotation);
    _instantiatedCubes.Add(cube);
    _cubeCount++;
    countText.text = "Cubes: " + _cubeCount;
}
```

You can replace the string of characters on the last line with whatever you want, just don't forget to increment the count AND to add it to the text.

Finally, go back to the Unity Editor. Drag and drop the text object into the script's field (just like you did previously) and everything should just work!

Before building and running the application, remember to enter Play-mode at least once to see if there are any errors.
Don't forget to save your scene.

Exercise

Add an UI to the first scene, as you did for that one. It must track how many cubes you have destroyed, in real-time. Remember to select the correct scene in the build settings before building! If you want you can improve the Application Manager script by allowing our cubes to reappear once the player kills them all.

The end

Congratulations on reaching the end of this practical! If there is still some time left, you can add other functionalities to the scenes. Feel free to experiment with Unity!