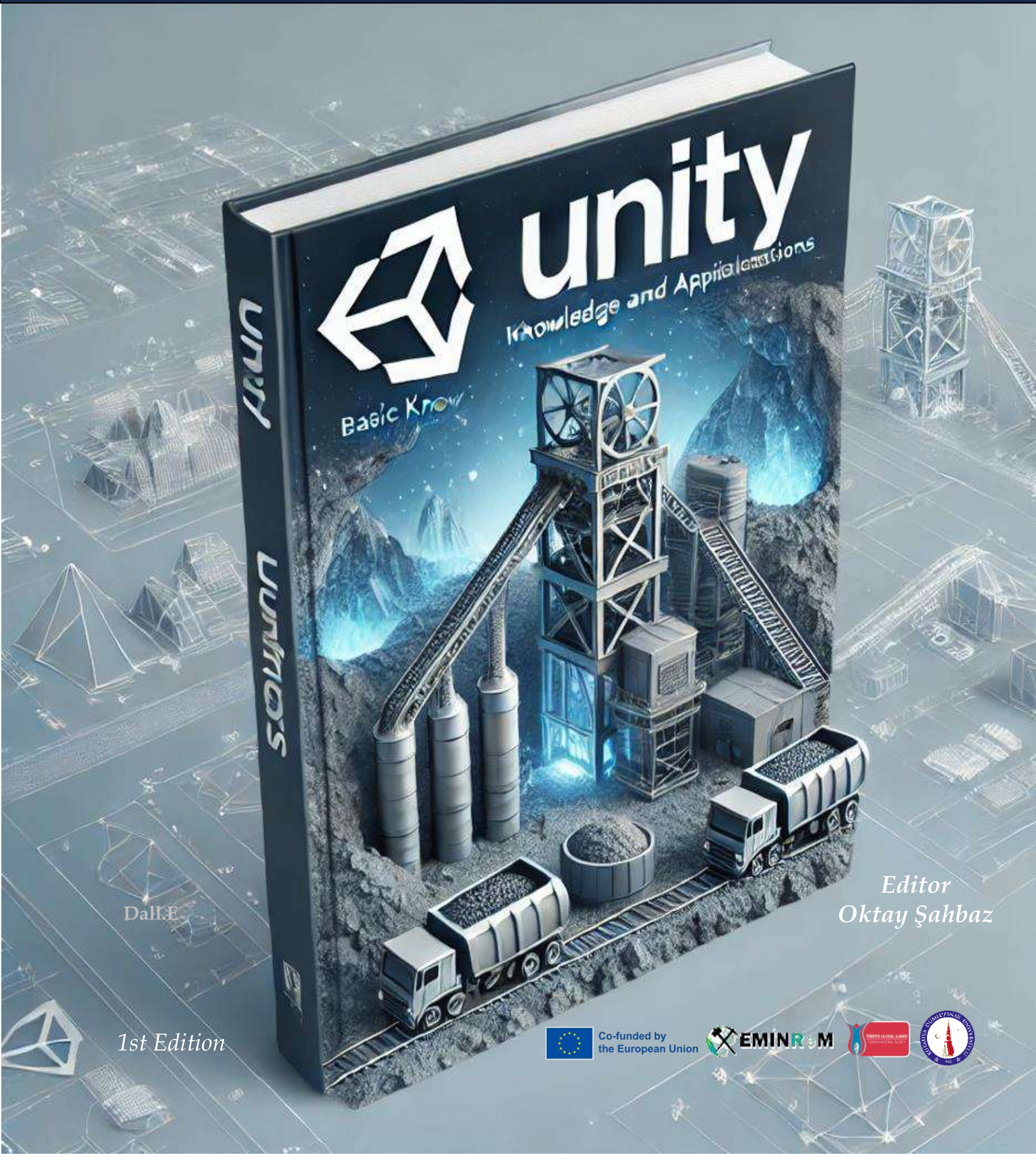
INTRODUCTION TO UNITY
A REAL-TIME DEVELOPMENT ENGINE

With Applications for Mining

Kaan Erarslan



Dall.E

Editor
Oktay Şahbaz

1st Edition

Co-funded by the European Union

INTRODUCTION TO UNITY
A REAL-TIME DEVELOPMENT ENGINE

With Applications for Mining

1st Edition

Kaan Erarslan

Editor
Oktay Şahbaz

Co-funded by the European Union

PREFACE

In the first quarter of the twenty-first century, computer electronic hardware and software are developing at a dizzying pace. A large technological matrix is emerging with artificial intelligence, deep learning, the Internet of Things (IoT), smart systems, and visual communication elements. Visualization and imaging technologies continue to develop as an independent area and a component of this large matrix.

Computer-aided modeling, visual communication tools, and classical methods provide extraordinary designs and products. In the last quarter century, one of the breakthroughs that have affected our lives has been the platforms called real-time development (game) engines, which generally appeal to every professional discipline and business area. It is a useful and empowering new tool for visual design and application developers to have in their software toolbox.

Developments in Virtual-Augmented Reality (VR-AR) and Serious Games are the subject of many innovative studies in educational curriculum and design, and their positive contributions are revealed by scientific research.

Educational materials developed in this field can be used on computers, the web, mobile devices, special headsets, and smart glasses. Therefore, all device options with their unique character offer a wide working area for application developers.

Unity is one of the most important cross-platform software for developing applications in VR-AR and Serious Games (Gamification). Unity, a real-time development engine with a history of nearly a quarter of a century, is software that can be used to develop all VR-AR and Serious Game applications. Unity offers application opportunities in all educational disciplines with these features. For this, Unity knowledge and engineering field knowledge is required.

"Introduction to Unity, a Real-Time Development Engine *with Mining Applications*" aims to provide the basic knowledge and applications in these subjects. The focus is on the visual design and application aspect rather than game development. Educational materials are mostly video-based visual materials and limited written documents due to the visual nature of the subject. In the last decade, many comprehensive and detailed video documents have been prepared in this field. It is possible to access the training records, which are paid or free, on the Internet. In fact, the most important source in this field is the constantly developing visual and written archive on the internet. Written documents may become outdated because of developments in computer software. However, these notes have been prepared to provide a written document about general information and some applications in a way that is suitable for those learning from scratch.

In the book, Unity essentials are explained, as well as general information about C# coding is also provided. Virtual and Augmented Reality topics were discussed. For the training to be successful, application studies and video materials should be considered together with the presented tutorials. In other words, practice studies, videos, and written documents are all components of a whole, parts of understanding the subject better and being successful. Apart from this, the most important key to success is individual interest, motivation, and efforts during and after the training process.

This book includes basic information about Unity, a real-time development engine, and its applications in mining engineering. The book, which consists of tutorials, also has the feature of being educational material.

Acknowledgments

Acknowledgments are extended to the **European Union**, **Brussels** for supporting the **101082621-EMINReM-ERASMUS-EDU-2022-CBHE "Master Program in Eco-Mining Engineering and Innovative Natural Resources Management (EMINReM)"** project and to the **Kütahya Dumlupınar University Scientific Research Projects Unit** for providing laboratory facilities with the **"Search and Rescue Training in Mines Using VR/AR Technologies"-DPÜ-BAP 2022-63** project.

Files in Share

The course book, asset packages, C# scripts, materials and video tutorials are presented in the folders and files.

https://drive.google.com/drive/folders/156AOkT8JP6khzlcnr02u7BzcIzeWZnjC?usp=sharing

Table of Contents

INTRODUCTION

Unity and similar real-time development engines are used to develop games, VFX movie scenes, and science and educational materials. With its cross-platform feature, it is possible to develop applications for many different device options, such as computers, mobile devices, game consoles, virtual reality (VR) headsets, and augmented reality (AR) smart glasses.

Unity, which can be used in almost every professional discipline, has applications in education, fine arts, health sciences, and engineering.

To become an educational material or field application developer in any professional field, it is necessary to learn the basic information of the Unity engine first. This book, which is prepared based on tutorials according to the subject titles for beginners, mining engineering, which includes many disciplines from earth sciences to construction machinery, from electrical systems to office management, from occupational health and safety to education, from management to labor law, has been preferred as the application area.

Book content:
Installation
Editor's introduction
Physical materials
Solid objects and gravity
Particle system
C# basics
Basic C# coding for Unity
Collision control with C#
Object control with mouse with C#
Raycast C# coding
Scene design and adding objects
Cameras
Terrain design
Output for PC (EXE), Mobile (APK), VR Headset and AR Smart-Glass
Adding audio and video
Simple animations
Working with UI (User Interface)
Scene transitions with UI
First Person (FP) and Third Person (TP) applications for PC and mobile devices
Virtual reality (VR) applications (for Cardboard)
Augmented Reality (AR) applications (with mobile devices)
Using animation-animator in Unity with Adobe Mixamo library and C# coding

Open pit, underground mine, ore preparation-enrichment facilities, ventilation, various machines sample application applications developed for the titles It is located as.

The book can be used as educational material for beginners as well as for teaching levels.

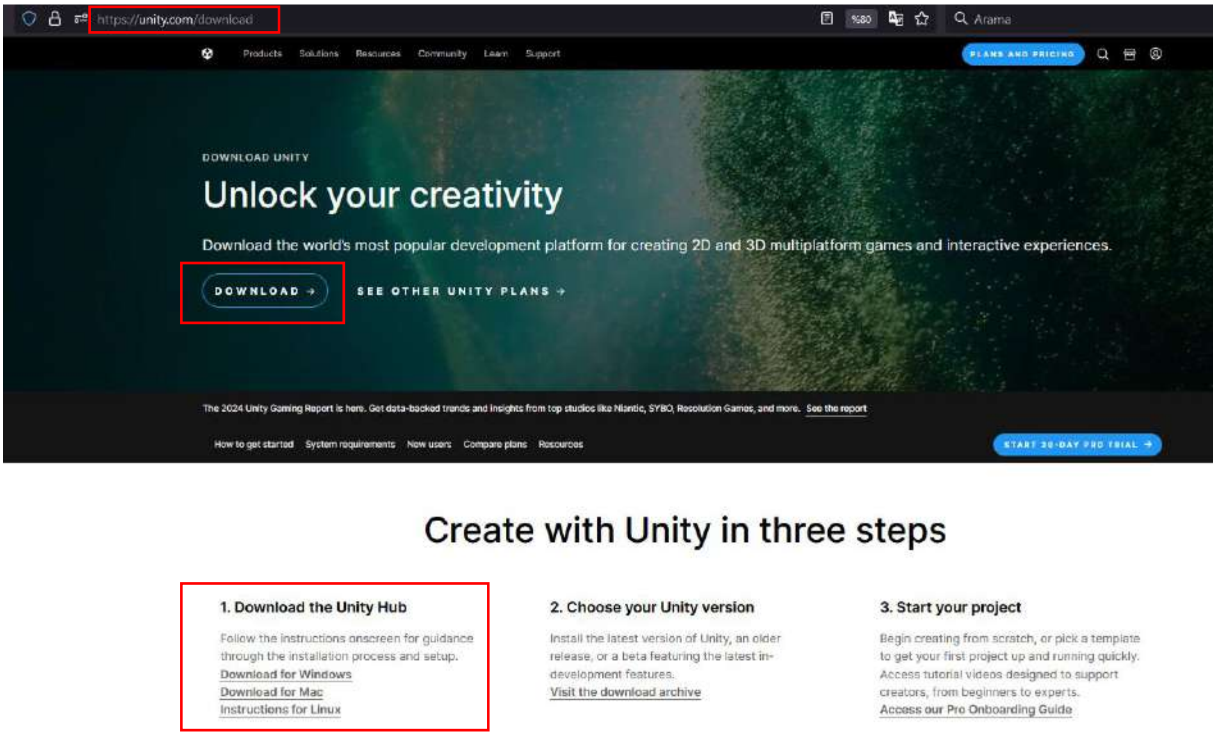
1. INSTALLATION OF NECESSARY SOFTWARE

Within the scope of the course, all the applications will be developed with **the Unity Real Time Development Engine**. On the other hand, different software will be needed for the applications to be developed as **PC-based, Web-based, mobile** and mobile-**AR/VR**.

The software that will be required for the course to be installed depending on the applications is listed below:

1.1.Unity 3D installation

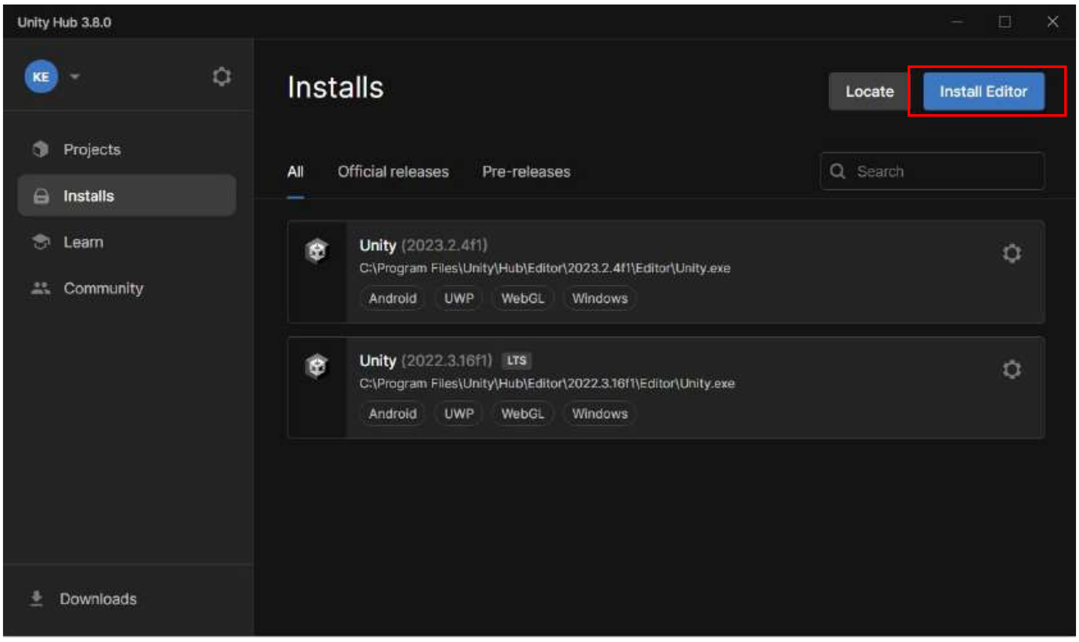
It is possible to reach the download page from the relevant page that will open after browsing with Unity on Google or from the link https://unity.com/download



First, create a Unity account with the email you will use in all your work. Since the system matches email accounts, it is important to have the same email account (i.e. Unity asset store).

Now we can download the **Unity Hub** program. This program plays a central role in all installation and project operations. It is recommended not to use the beta trial version.

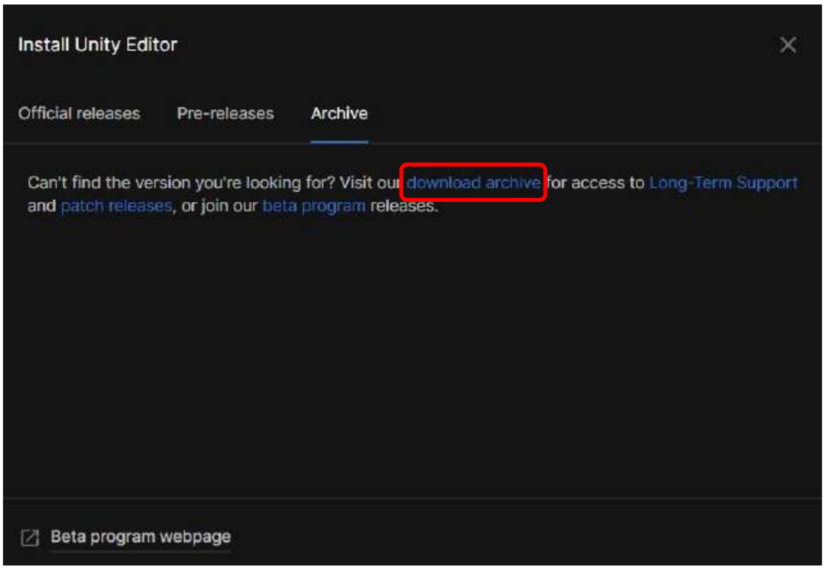
On the Hub, you can make your first download by clicking the **Add** button in the **Install** section. The system will recommend the version with **LTS** (Long-Term Support). Start installing by selecting the 2022.x LTS version.



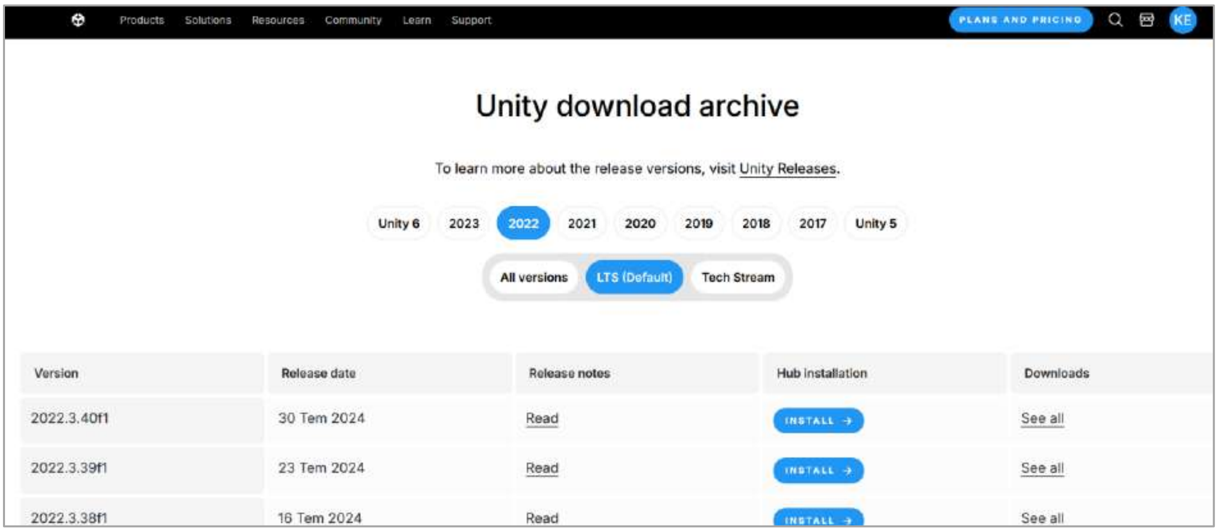
Before installing, be sure to mark the following additional subcomponents:

Visual Studio 2022 Community
Android Build Support -> Android SDK & NDK Tools (for Android devices)
-> OpenJDK
IOS Build Support (for IOS devices)
WebGL Build Support

On the Hub, it is possible to add another version by clicking **Add** from the **Install** tab. For older versions, **download archive** selection should be made.



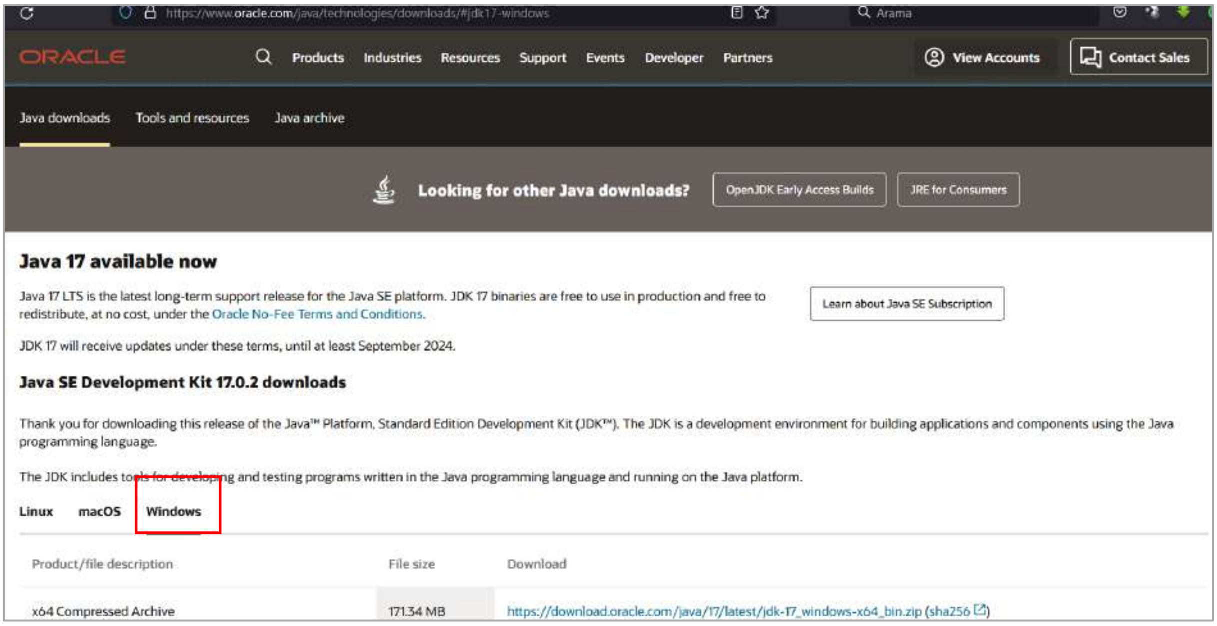
Unity's website will open, and you will be given the opportunity to choose a version.



For example, you can find and download Unity 2019.1.14. This version is still preferable for **VR** (Virtual Reality) and **AR** (Augmented Reality) applications. However, it does not support the latest updates.

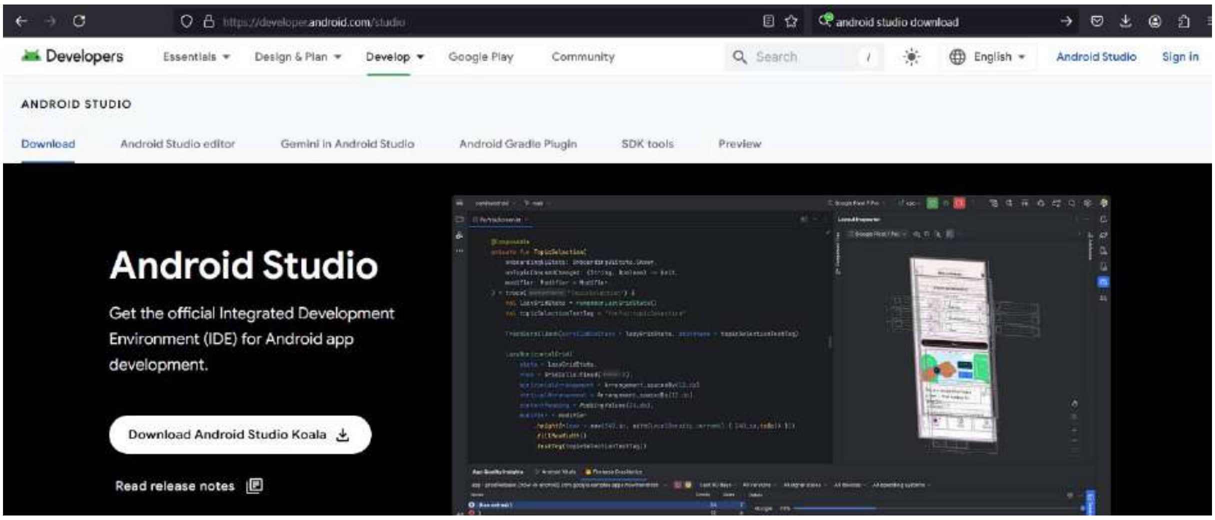
1.2.Java JDK

If not installed with Unity, **Java SE JDK** must be installed manually. To do this, a search on Google with "java sdk download" will reach the download page on **Oracle's** website. Windows users should download and install the relevant package (zip or exe) from this tab. The macOS version must be downloaded for the IOS operating system.



1.3.Android Studio

If the required components are not installed during Unity installation, Android Studio should be installed for Android mobile applications. You can access the relevant page by typing "android studio download” on Google to do this. From the table at the bottom of the open page, the latest version for Windows or Mac should be downloaded and installed.

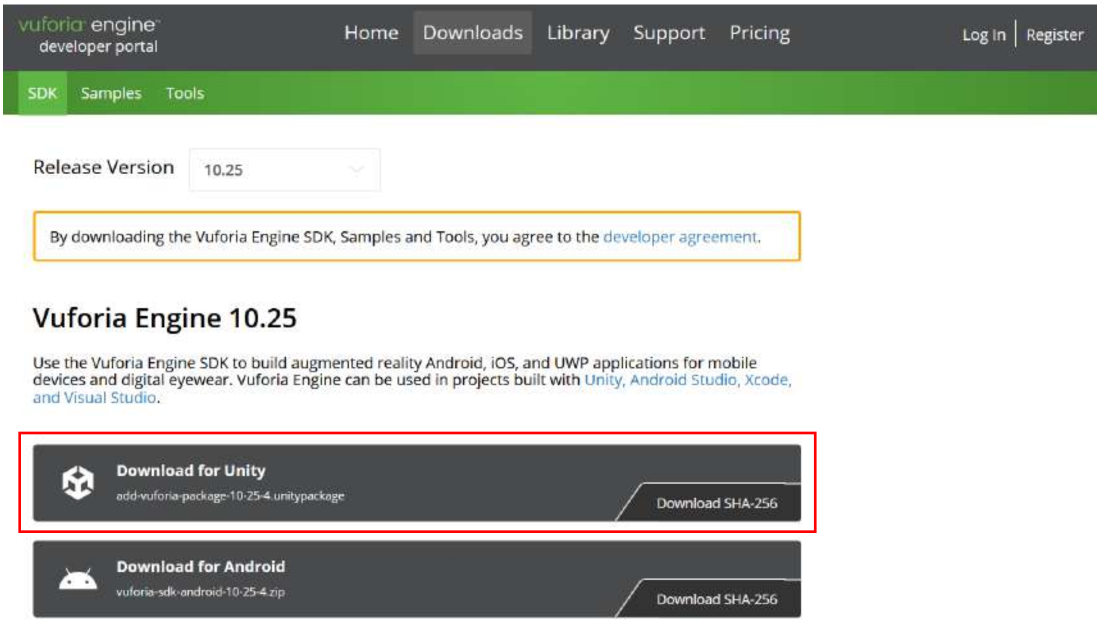


1.4.Xcode for IOS Systems

For compilation processes of IOS mobile systems, the **XCode** package should be downloaded and installed. The relevant page can be reached by typing **XCode** in Google. First, it is necessary to create a user account. At this stage, your Apple phone must be ready due to security protocols.

1.5.Vuforia

One of the pioneering companies that develops augmented reality applications is **PTC**. Mobile AR applications are developed practically using a product called **Vuforia**. We can reach the download page by typing **Vuforia SDK download** on Google. As a developer, download and install the package related to Unity.



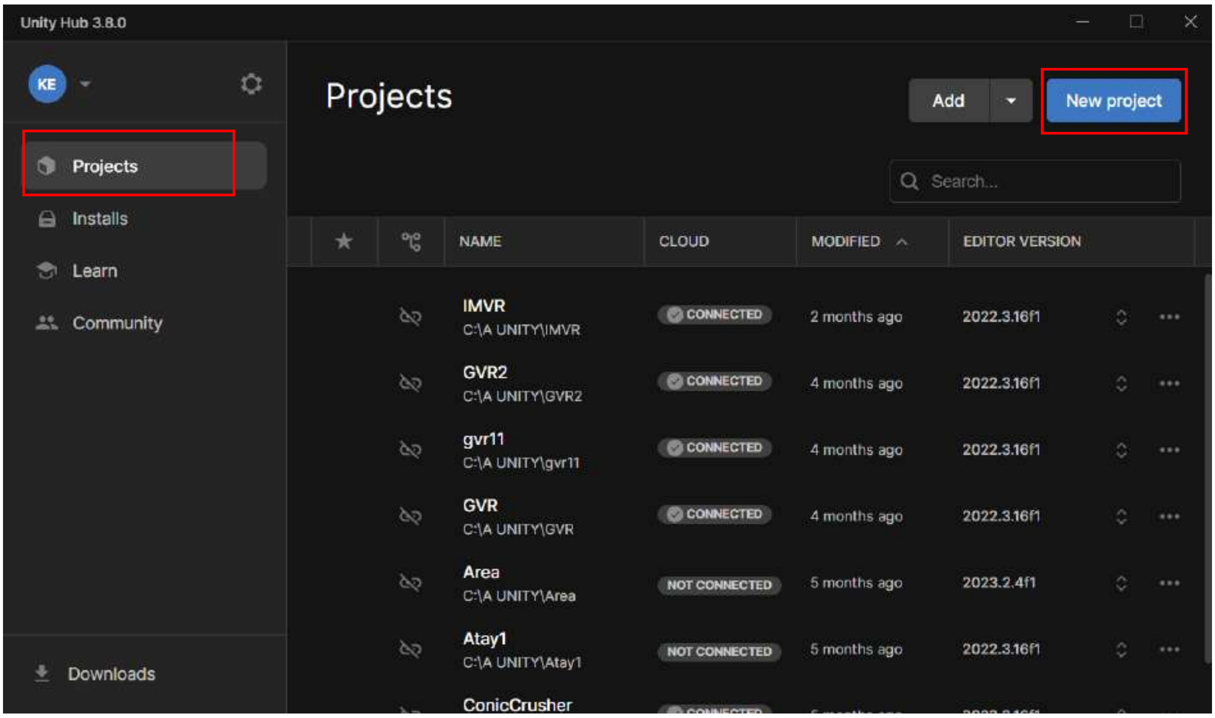
Make sure to create a user account in Vuforia. Although it is not compulsory, it is recommended to use the same email address as Unity.

Vuforia is Unity's official partner for applications in augmented reality, specifically for the **Hololens 2** headsets.

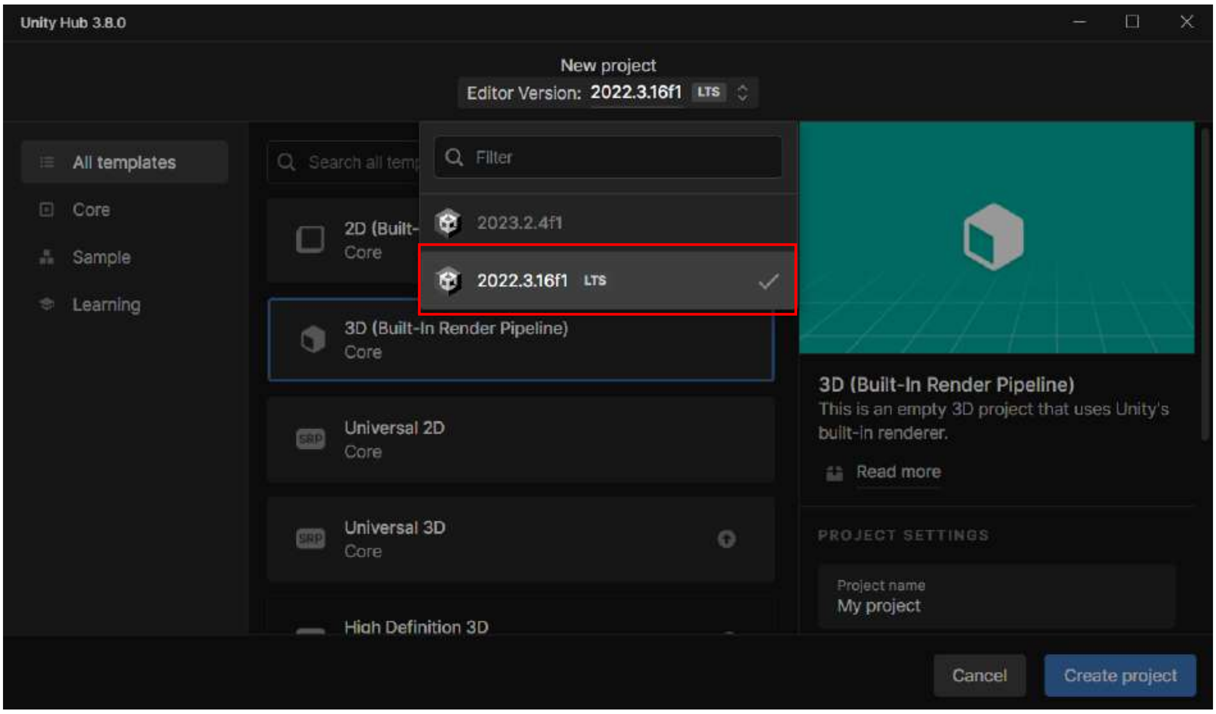
2. UNITY PANELS

2.1.Using Unity 3D Panels

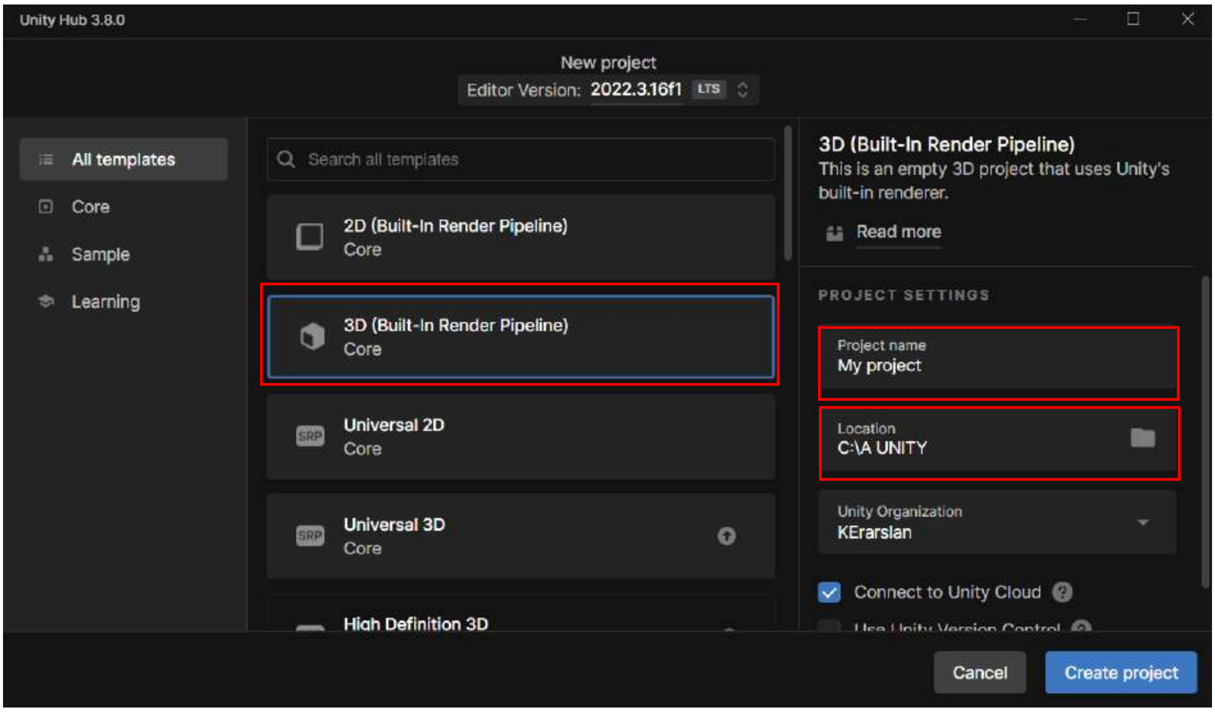
First, a project should be started. To do this, select the **Project** tab on **Unity Hub** and perform the first step of creating a new project.



If more than one version is installed, the version we want to use is selected first (currently 2022.3.1.50f).



Here, the templates for the version are listed, and the name of the project (**Project name**) and folder (**Location**) are specified. By giving the name **Project1** to the first project and specifying its folder, our editor will be opened with **Create**.



The general appearance of the empty panels is as shown in the figure. In the upper left corner is the project's name, **Project1**, and the **SampleScene**, which shows which scene it is in①.

SampleScene name is given by the system. When you look at the scene sections, there is a window at the top left that shows the scene named **SampleScene** and the **objects** (**assets**) in the scene, called Hierarchy②. Each asset (object) here can also be called a **Game Object**. When we right-click our mouse, the objects in the menu can be added.

At first, there appears to be only a **Main Camera** and a **Directional Light**. The part where the scene is located is in the window called **#Scene**③.

In the section titled **Project**④, there are **Favorites**, **Assets,** and **Packages**. Here are the files of the objects and packages that will be added to the project to be added to the scene. All the movements here are simultaneous and parallel with Windows Explorer.

The most important and frequently used subheading in the **Project** window is the section called **Assets**⑤ because it is the section where all kinds of objects to be added to the scene are located. We can think of these assets as various **scene** objects such as **2D, 3D objects, materials, sound, light, video, effects, animation, C# codes**, etc. A large window belonging to **Assets** is also positioned to show the assets inside⑥.

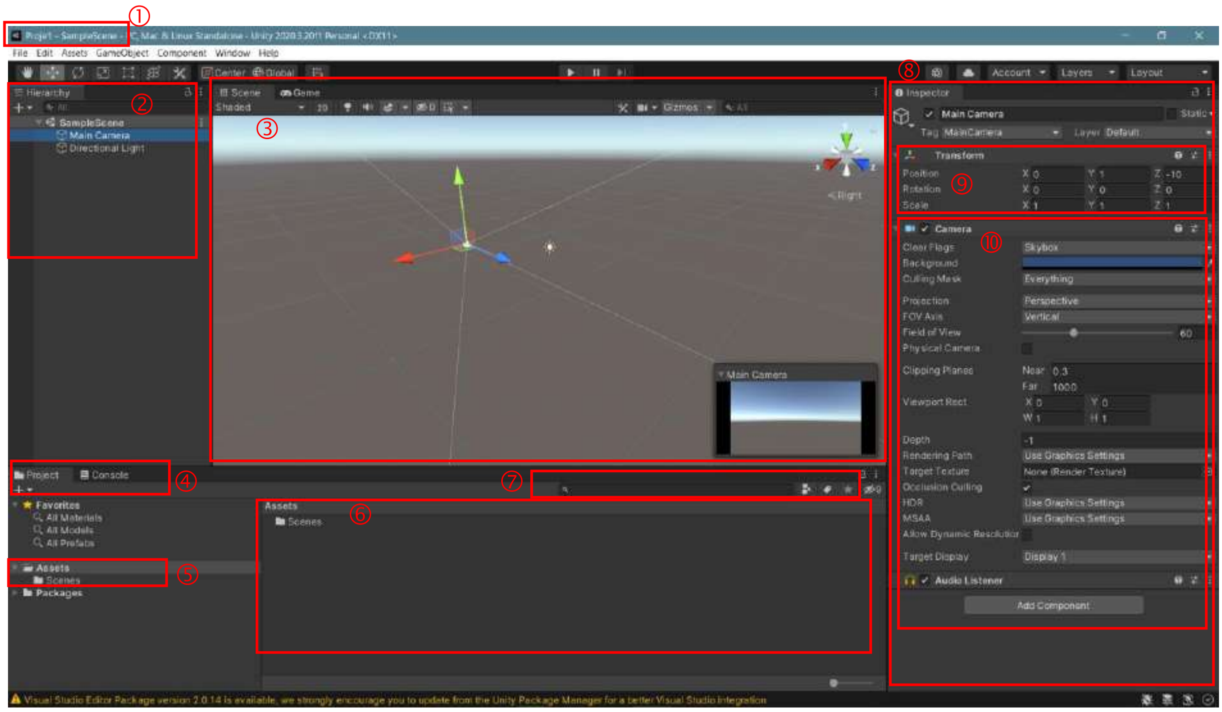
There is an asset search window just above this window⑦. As can be seen in the figure, **Scenes** are also assets.

The Inspector window is where we will make arrangements for each scene component, we select in the Hierarchy window⑧. There may be dozens of sub-parameters specific to each of the hundreds of objects in Unity, and dozens of sub-parameters belonging to them. These will be visible in the **Inspector** window.

We see the details of the **Main Camera** selected in our empty scene in the **Inspector**. Under the checked part 

that shows that the object is selected and in use, there is the **Transform** window where the objects in the scene, also known as **Game Objects**, are located, which are basically located in most of the scene's (**x, y, z**) coordinates; **Position**, rotational angles; **Rotation** and size scales; **Scale**.⑨.

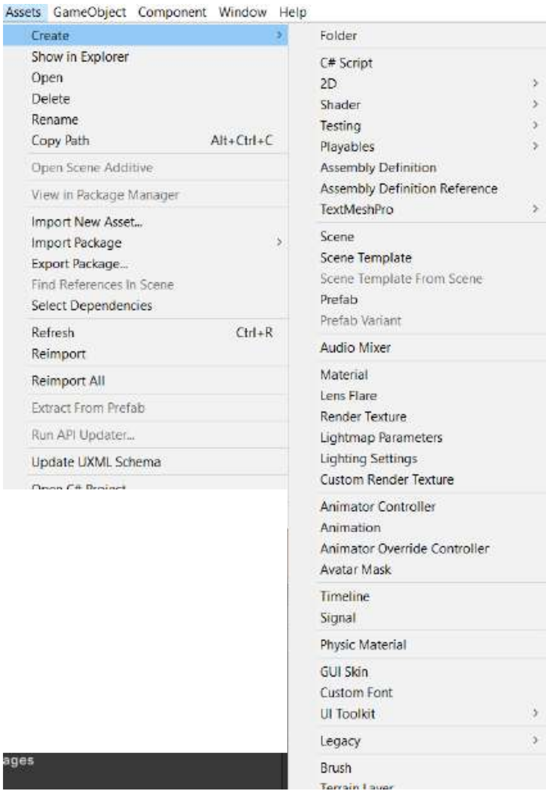
Again, in the editor of the **Main Camera**, **Inspector**, there is a section where the **Camera** properties are defined⑩. Below is the empty audio editor, **Audio Listener**. Below that is the **Add Component** button, which adds various properties to the selected object.



When you examine the project's menus, you will see File, Edit, Assets, GameObject, Component, Window, and **Help**. The **File** menu includes options for scene operations, file recording, project creation and platform operations.

There are editor operations in the **Edit** window. The main ones are copy, paste, select, play, freeze, name and similar operations. The most important operations are **Project Settings**, project-related arrangements and Preferences, pre-preferences for various settings.

The **Assets** menu is the section where transactions related to assets are located. Here, there is a very rich tab titled **Create** to create assets.

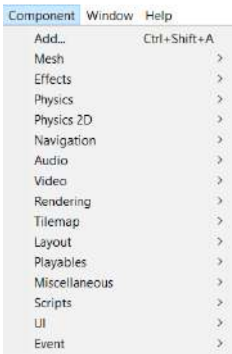


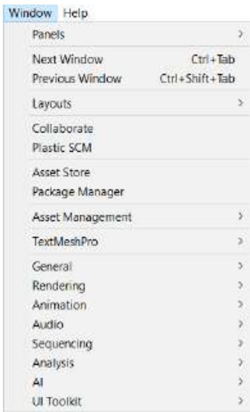
Also, the **Import New Asset** option is used to import a new asset object (file). **Packages** prepared outside Unity and in a format that Unity will accept can be imported into the project via the **Import Package** tab.

The **GameObject** submenu is a menu we can use when we want to add a new object, **GameObject**, to our scene from the objects in Unity. The same operations can be done in the **Hierarchy** window by right clicking our mouse.

The **Component** menu contains the content of the **Add Component** key that we use to assign various properties to objects.

In the **Window** submenu, the most important and frequently used sub-tabs, in addition to window arrangements, are **Asset Store** and **Package Manager**. Asset Store connects to the **Unity Asset Store** web page and allows us to include many free and paid asset packages in our project. The other option is the **Package Manager** option, which allows asset packages to be managed and added to our project. Package Manager has an important function that manages standard Unity packages and the assets we add to our assets via the Asset Store.



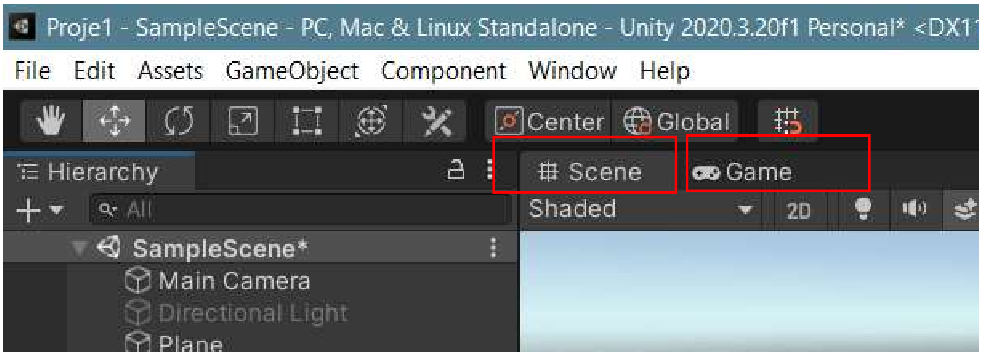


This section, which is a general introduction, can be understood by practicing while developing a project. The only way to understand and learn menus, their dozens of submenus, and hundreds, thousands of control parameters is to use them in the project.

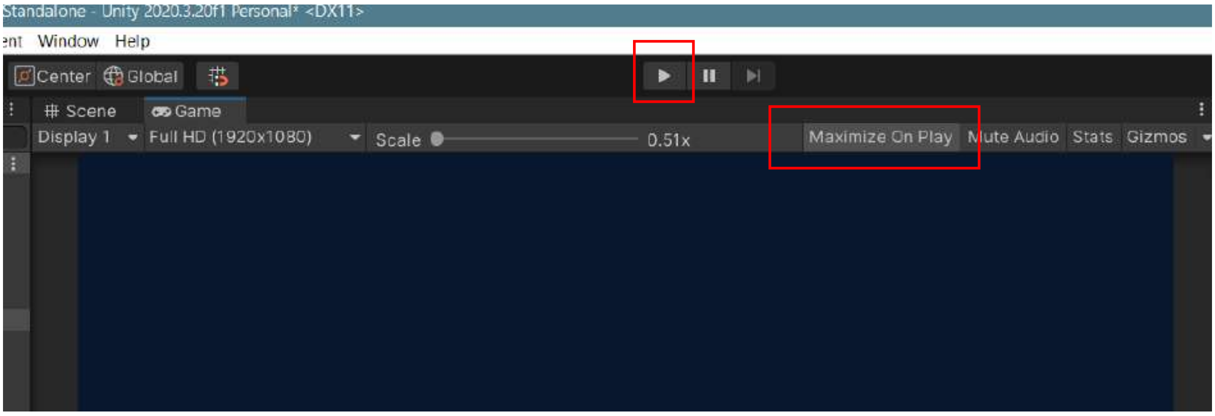
In conclusion, Understanding and learning Unity 3D will be the best possible by following the course, practicing the visual video materials, and applying them in person due to the visual character of this very comprehensive program.

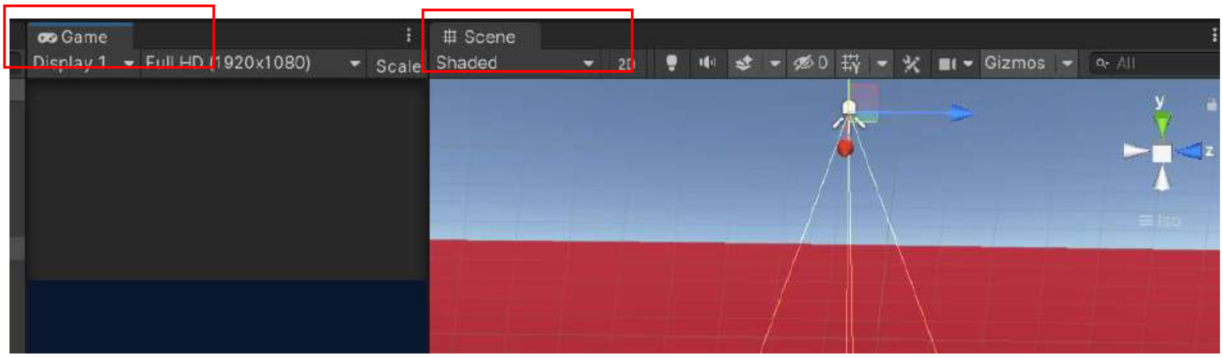
2.2.Game Tab/Mod

In addition to the **#Scene** window where stage design is done in Unity, there is also a **Game** window to see what will be encountered when the game is started.



When the play button for this window is clicked, the display window belonging to the **Game** becomes active. If the **Play Maximized** option is active, the game scene will cover the entire screen. The game is closed by pressing the same button. The design page is activated by selecting the **#Scene** tab. It is also possible for both windows to be active at the same time. For this, the **Game** tab is fixed as a window in the desired position by dragging it. It can also be brought back to its previous position in the same way.





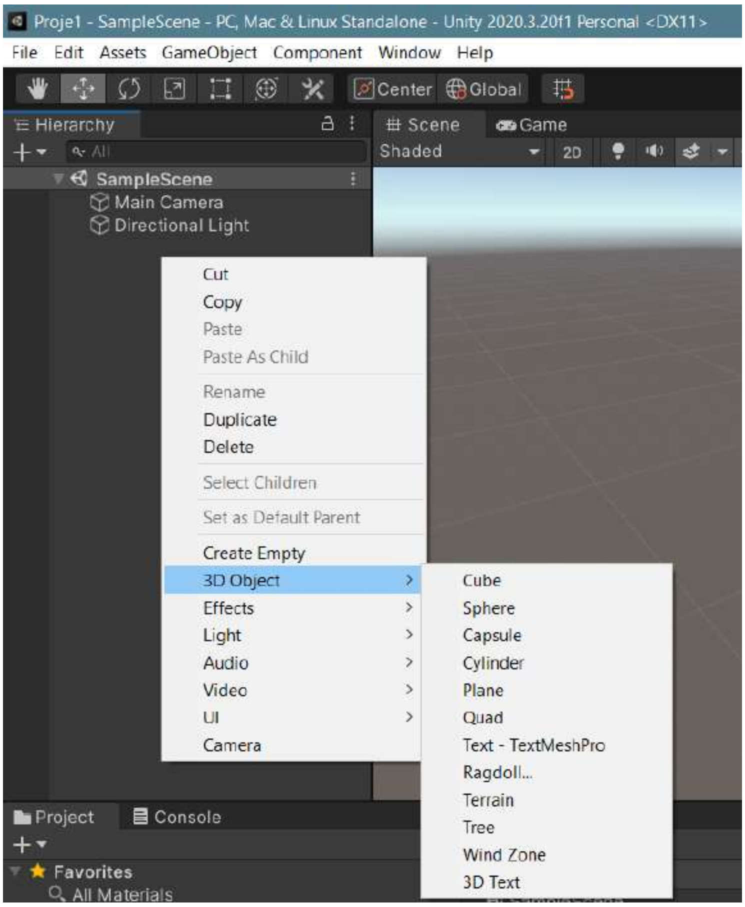
The **Game** mode feature will be used continuously in future scenes, and its function will be better understood.

3. SCENE DESIGN

3.1.Adding a GameObject

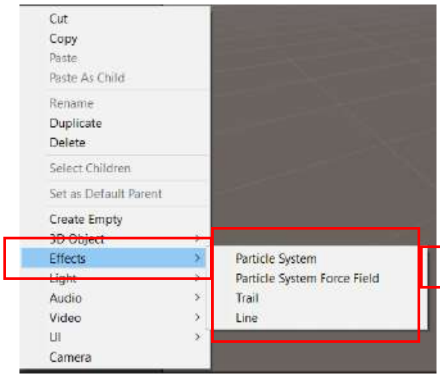
To add an object (**GameObject/Asset**) to the scene, internal or external sources can be used. First, in our project named Project1, let's create some game objects from Unity's standard libraries in the scene named **SampleScene**.

In the **Hierarchy** section, we can see the titles of the objects that we can add by right clicking our mouse and the objects under these titles. **Create Empty** is used to add an empty **GameObject** that will be used for different purposes. The most used object group is **3D Object** - a 3-dimensional object.

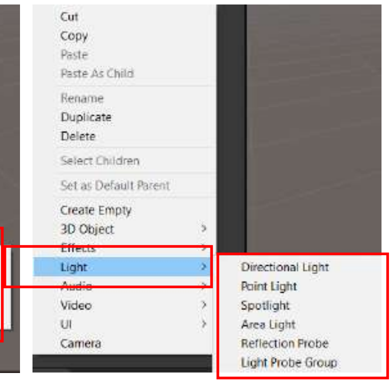


As can be seen, under the **3D Object** heading; Cube, Sphere, Capsule, Cylinder, Surface/Plane, Quad, **TextMeshPro, Rich Text Adding**, and **Ragdoll,** creating a character that is designed with a joint structure that we can move, Terrain, Terrain/Field Design, Tree Adding, Wind Zone, Wind Effect Adding, 3D Text – 3D Text Adding subheadings are included.

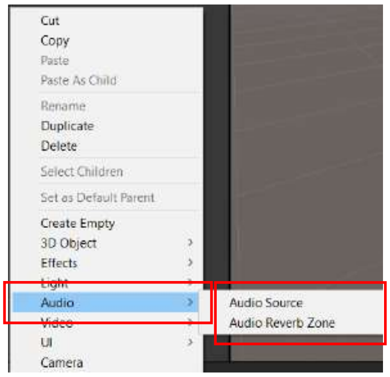
The menu and submenu components that we will see in detail while creating a project are in the figures below. These are, respectively, Effects, Light, Audio, Video, UI and Camera.



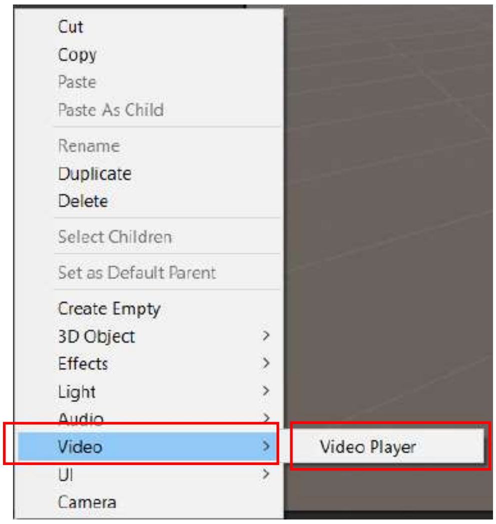
-Effects-



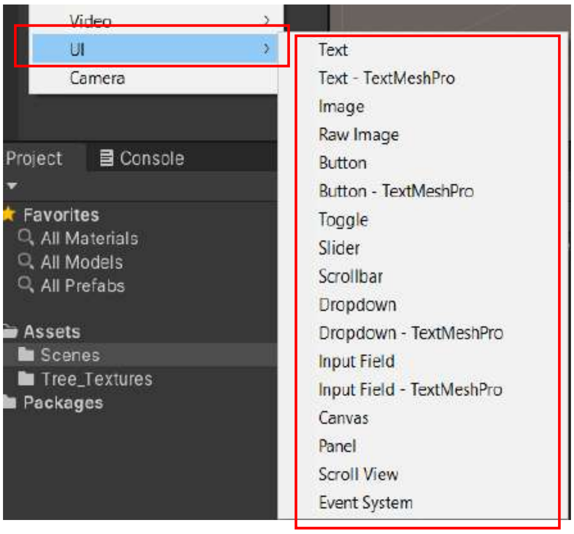
-Light-



-Audio-



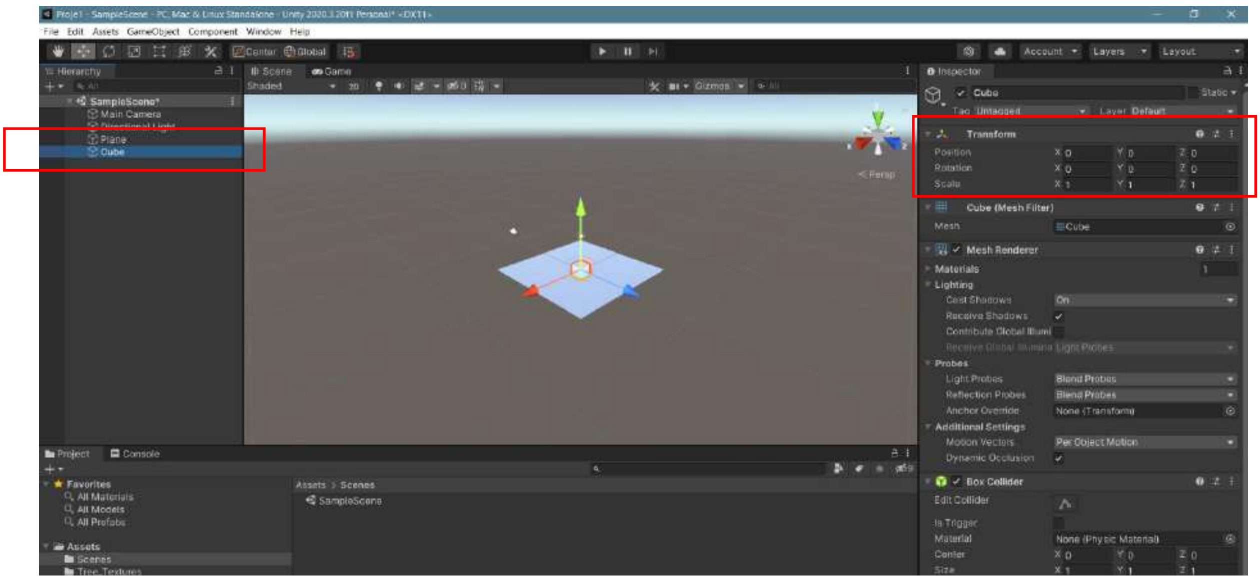
-Video-

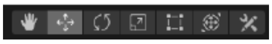


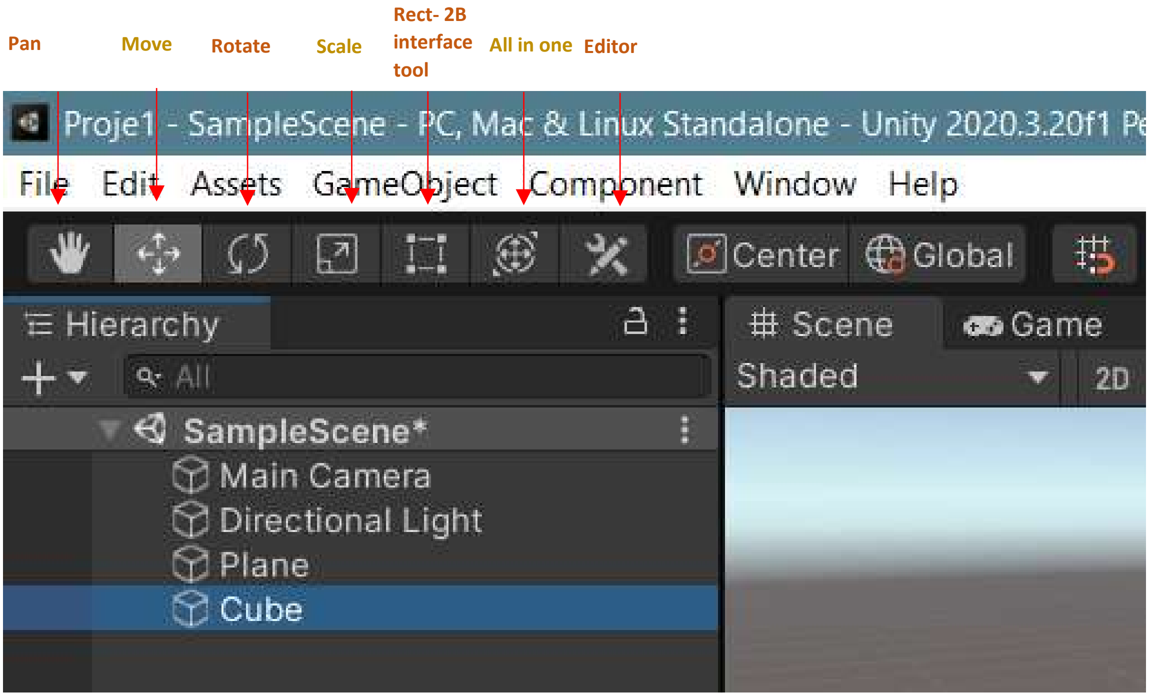
-UI-User Interface -

3.2.Object Properties

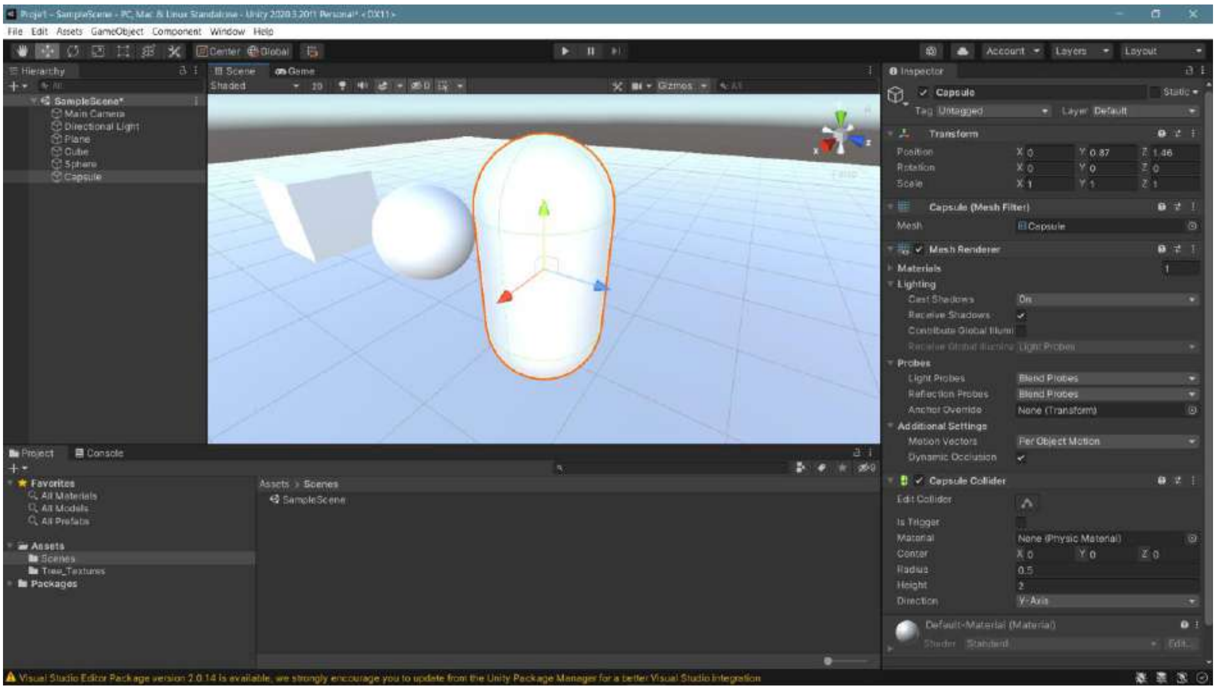
Let's add a few objects under **3D Object** to our scene. These will be a plane, a cube, a sphere and a capsule, respectively. But first, let's add a **Plane** and a **Cube** one after the other.



When the **Transform** window is examined, it will be seen that both Plane and Cube objects come with standard coordinates-**Position** of (0,0,0), angular rotation values-**Rotation** of (0,0,0) and scale settings-**Scale** of (1,1,1). To change these values, the numerical values in the window can be changed. **Transform** properties of objects can also be adjusted by selecting the shortcut keys located in the upper left corner of the project window. 

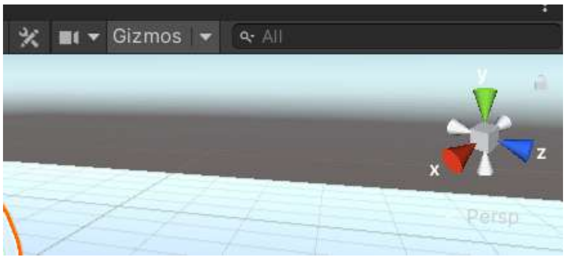


After selecting an object added to the list or an existing object with the mouse, if we hover over **#Scene** with the mouse and press the **F key**, it will be possible to focus and approach the object.



In practice, the rotation of the scene is done with the right click of the mouse, the zoom in and out with the rollers, and the **Pan** operation is done when the rollers are pressed.

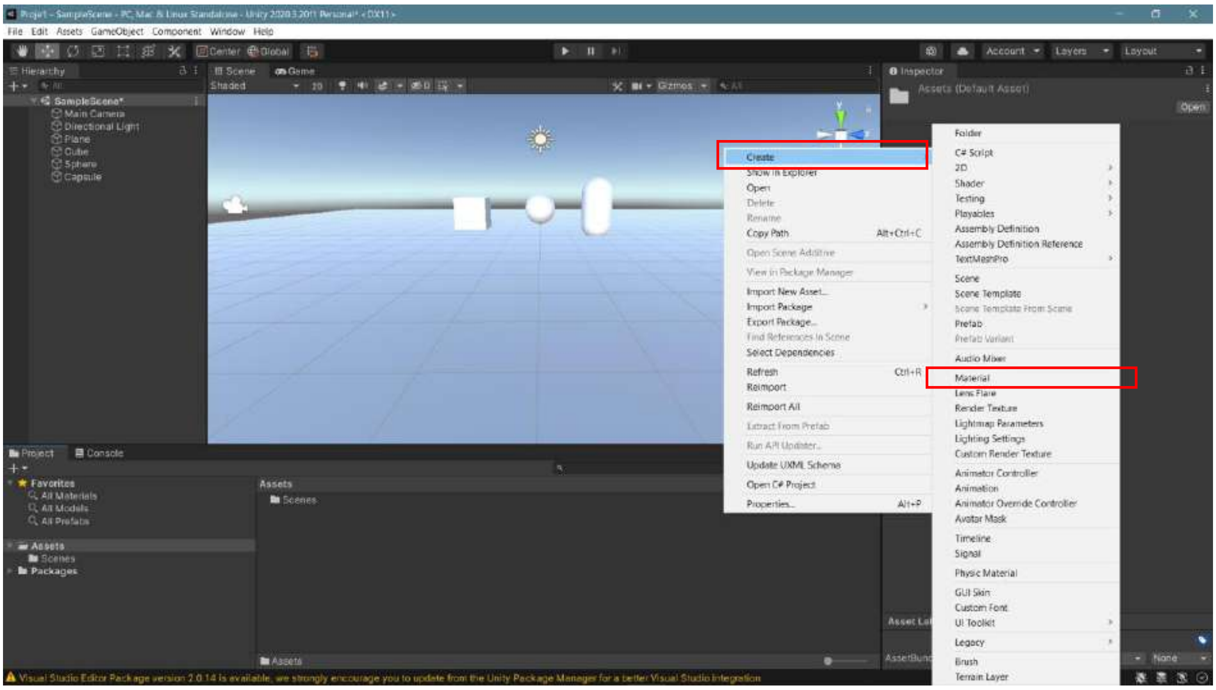
Another issue is to clearly determine the positions of objects in the scene. For this, it is useful to use direction arrows called **Gizmo**. As can be seen in the figure below, x-z is used to define the horizontal and y-vertical directions.



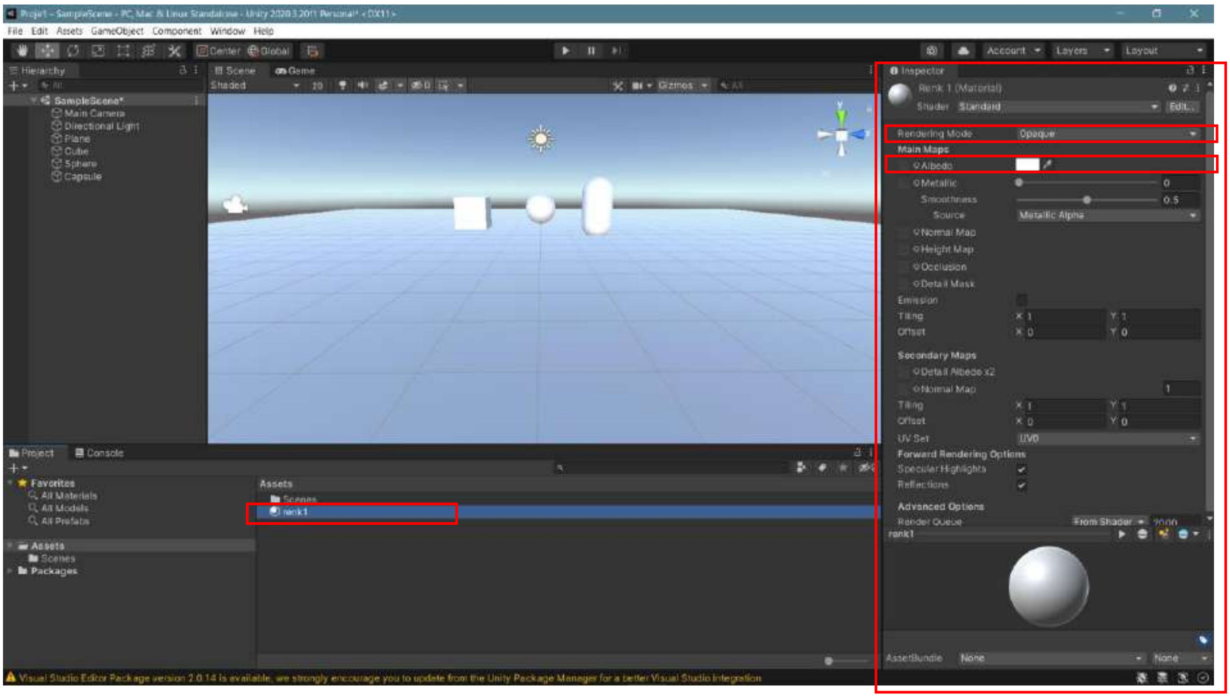
By clicking on the green **y-key**, we can look at our scene vertically from the top and horizontally from the sides with the red **x-key** and blue **z-key**. As a standard, a 3D image is given by taking perspective into account. **Pers** (perspective) expression under the **Gizmo** shows this. By clicking on the **Pers**, the image will be converted to **Iso**, or isometric format.

3.3.Materials

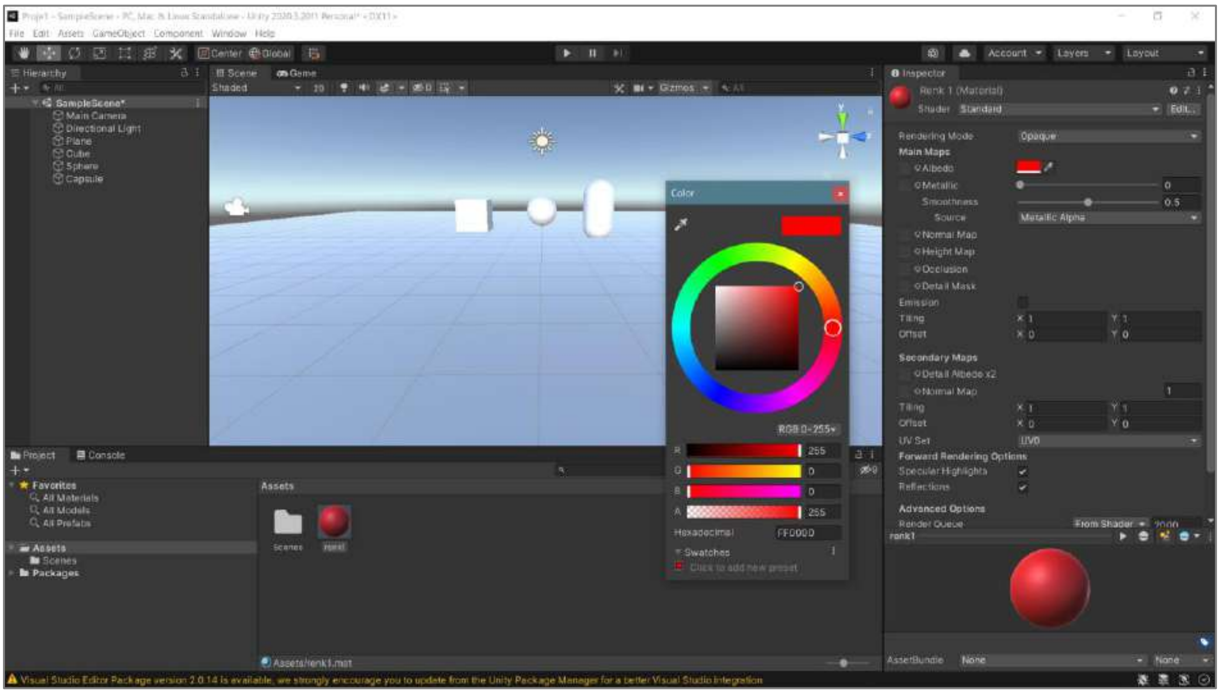
After creating objects, material is usually assigned to them. This can be a solid color, an image or a texture file. For this purpose, let's right click on the Assets window.



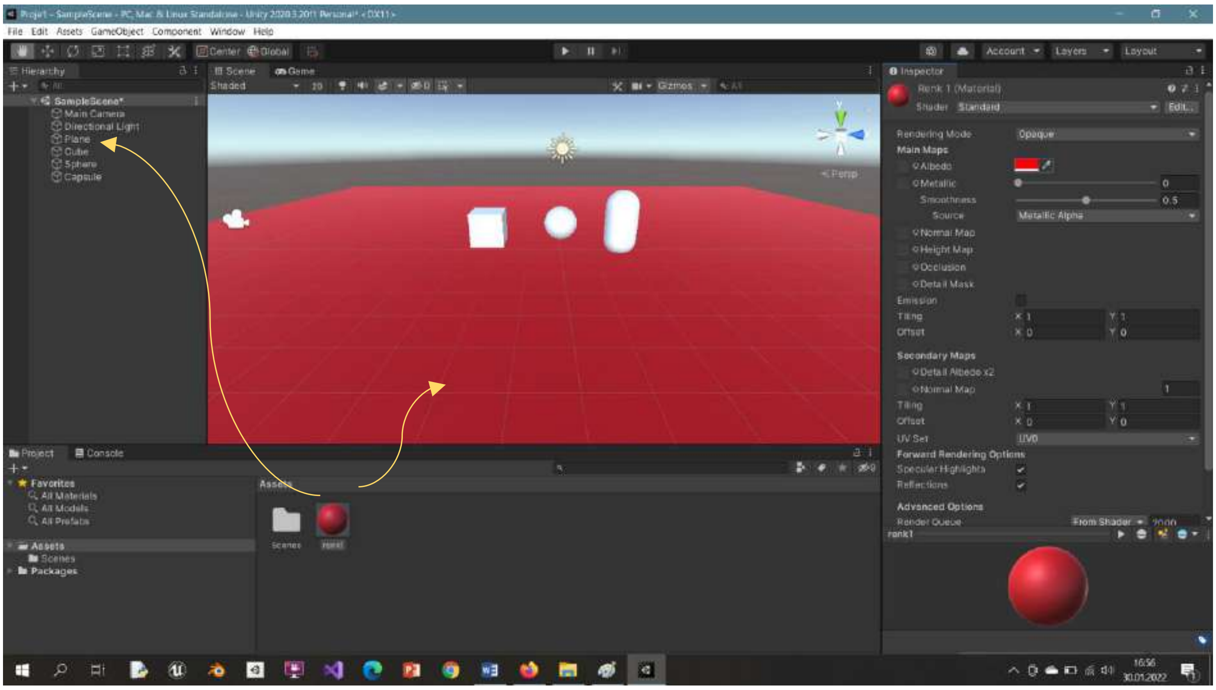
A window will open with a click. When you select **Create** here, a sub-window opens, and a comprehensive asset list appears. Since our goal is to create material, the **Material** option is clicked from the list. After giving a name to our new material/material, we can move on to editing information about this asset in the Inspector. Our material is opaque and white by default. Let's check the white box in the **Main Maps** window to determine the new colon.



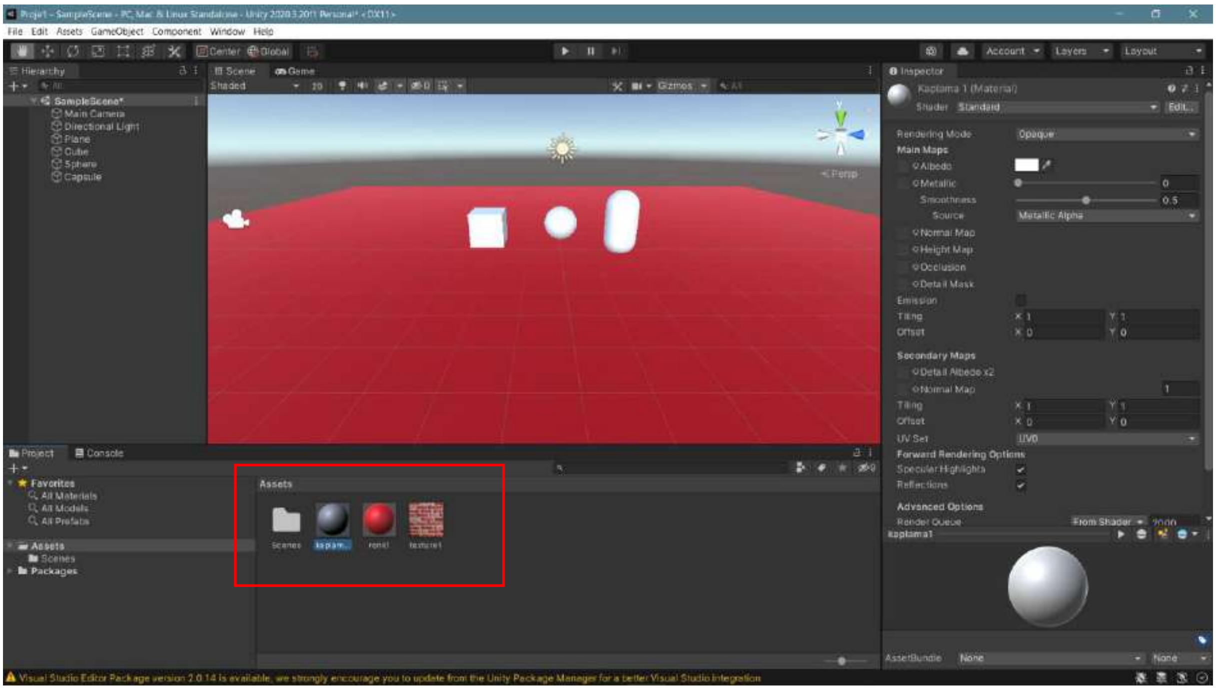
Let's specify the color we want from the **Color** window that opens.



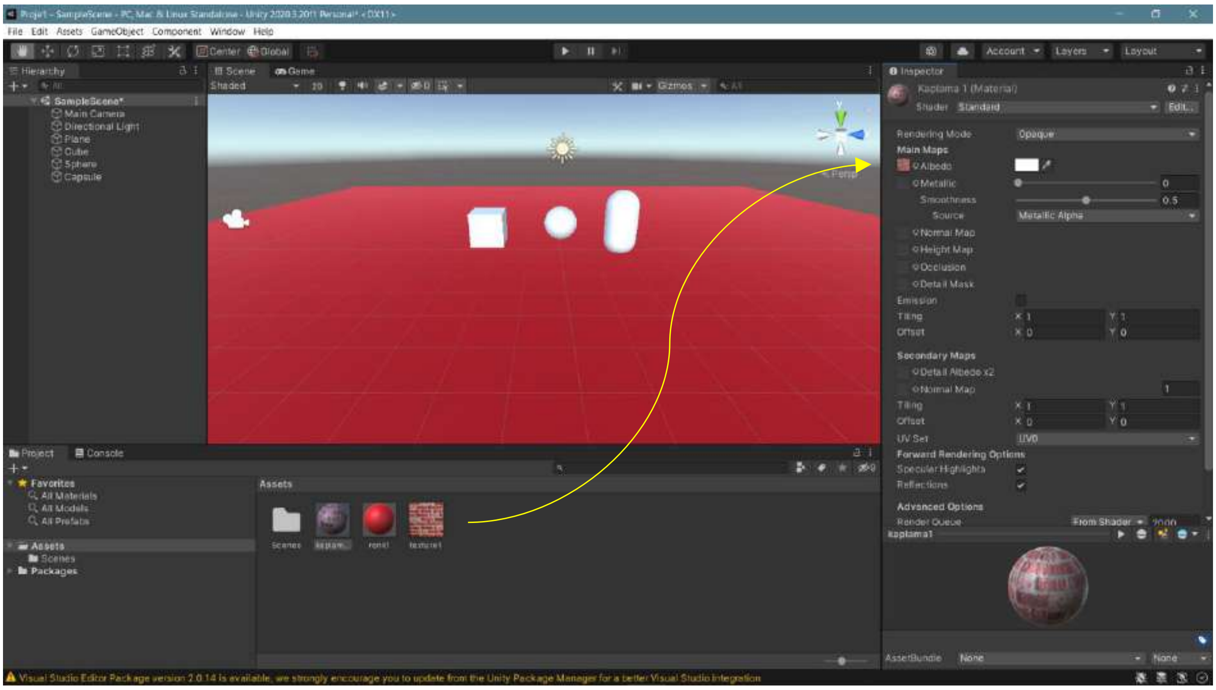
Now, it is possible to see the assigned color of the material that we created in the asset window. It should not be forgotten that there will be no change in the scene unless our objects in the **Assets** window are applied to the **Scene**. For this, we assign our color object in the scene by dragging it to the **Plane**, for example, or by dragging it to the **Plane** in the **Hierarchy**.



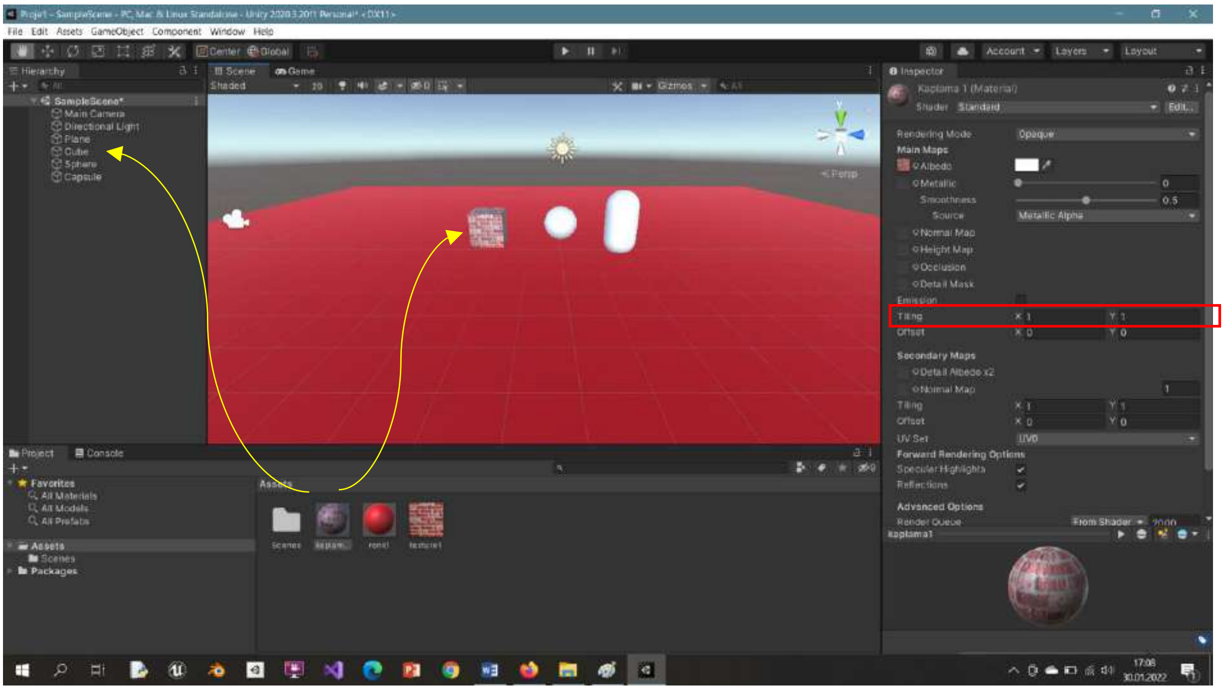
It is possible to do the same process on other objects and with different colors if desired. Another way of covering is to use a ready texture or image file. As can be seen, we added the **Texture1.JPG** file to the **Assets** folder by dragging it to the folder on Windows Explorer or directly to this window. To cover this material, we created a **Material** named **kaplama1**.



Now, let's drag our **Texture1.JPG** file to the box named **Albedo** in the **Main Maps** section in the Inspector window of the **kaplama1** material.



It is now possible to connect our material covered with **Texture1.JPG** to an object in the scene. To do this, we simply drag our k**aplama1** material onto our cube or the **Cube** on the **Hierarchy**.



The coating frequency can be changed by determining how many times the selected texture will be repeated in the **x** and **y** directions on the **Tiling**.

3.4.Lights

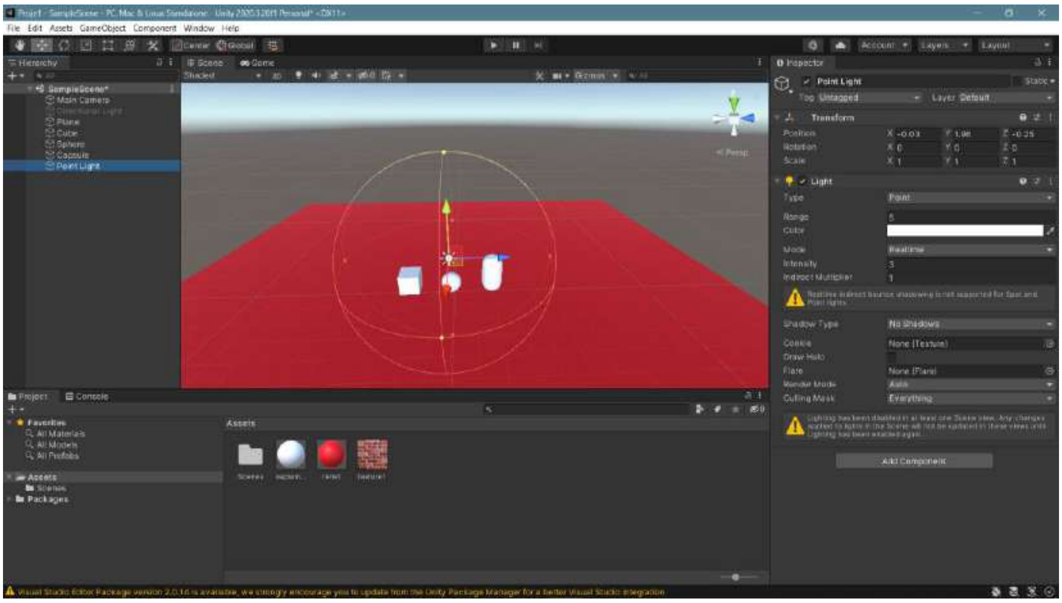
The subject of **light** is a very comprehensive title in game engines and has a direct impact on quality. Here, the addition of light types will be examined first. **Standard** light types are in the Light submenu. Here, **Directional Light** was already the current light type. It will be possible to see other light types more clearly by making the box for this inactive position in the **Inspector** or by deleting it.

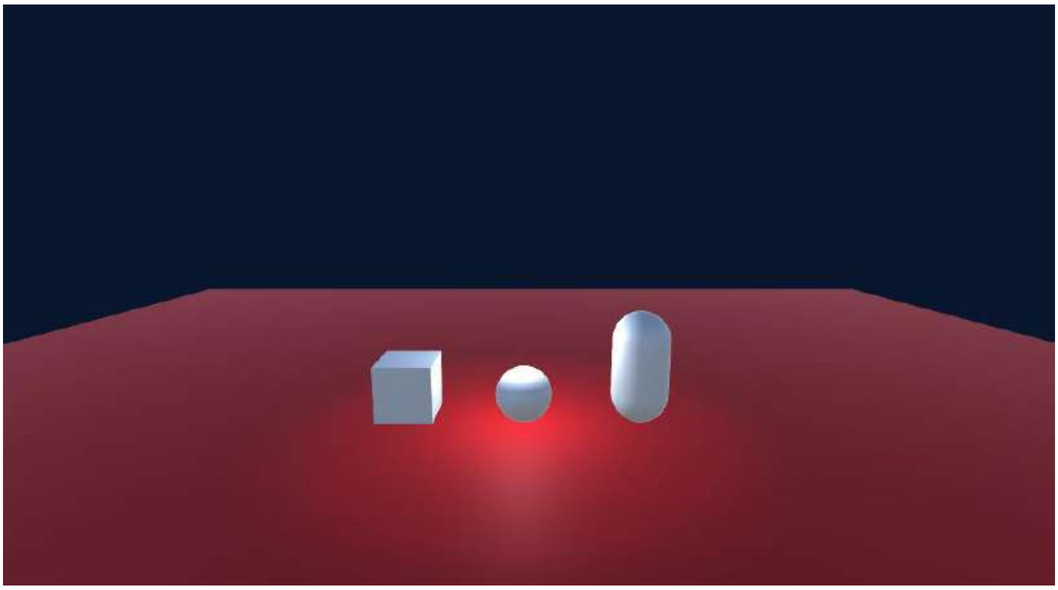
Let's try **Point Light**, **Spotlight**, **Area Light** types in that order. **Probe light** types can be examined at a later stage.

In this section, the information is given in a limited way due to its visual character and many light settings are demonstrated in practice during the lesson.

3.4.1 Point Light

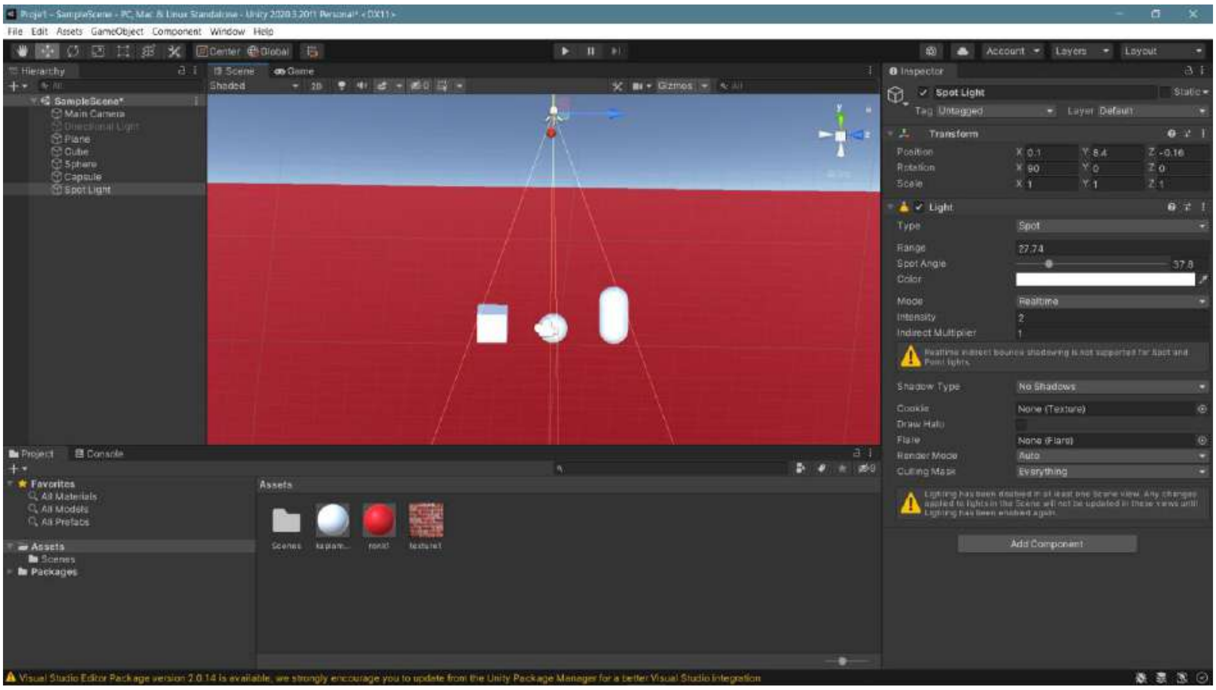
When **Point Light** is used, point light is applied to objects in a certain position, light color and intensity, and the following instant images are obtained. The images are shown first in the scene and then in **Game Mode**.

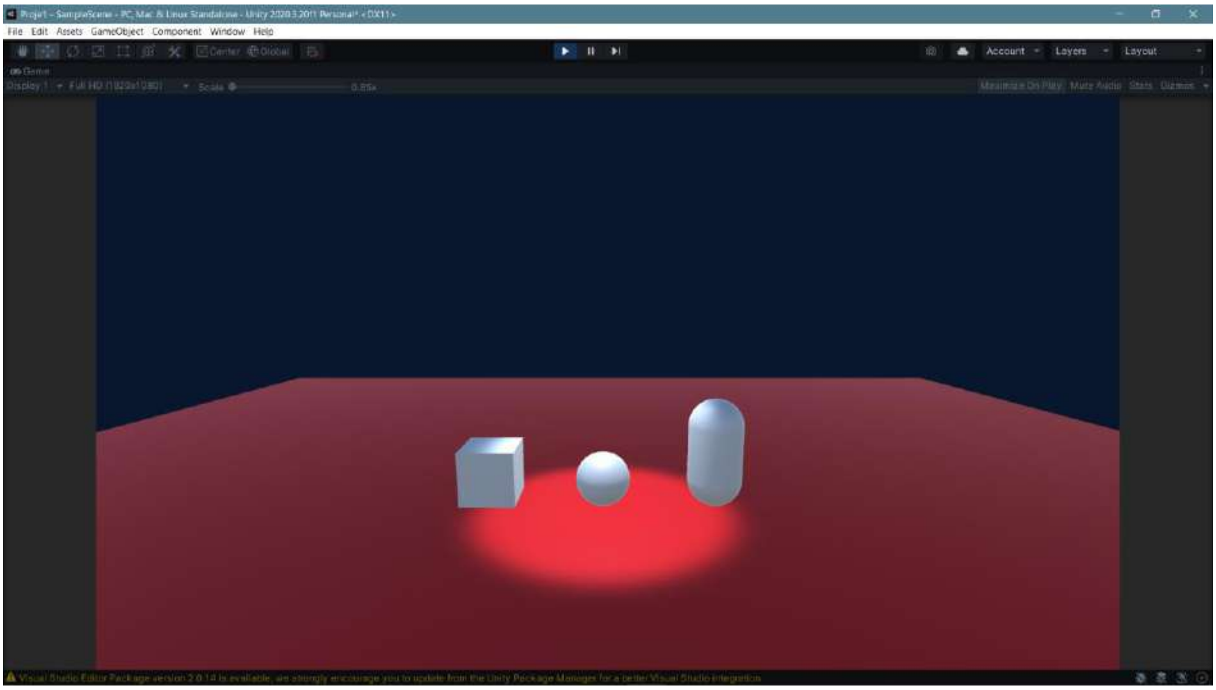




3.4.2 Spotlight

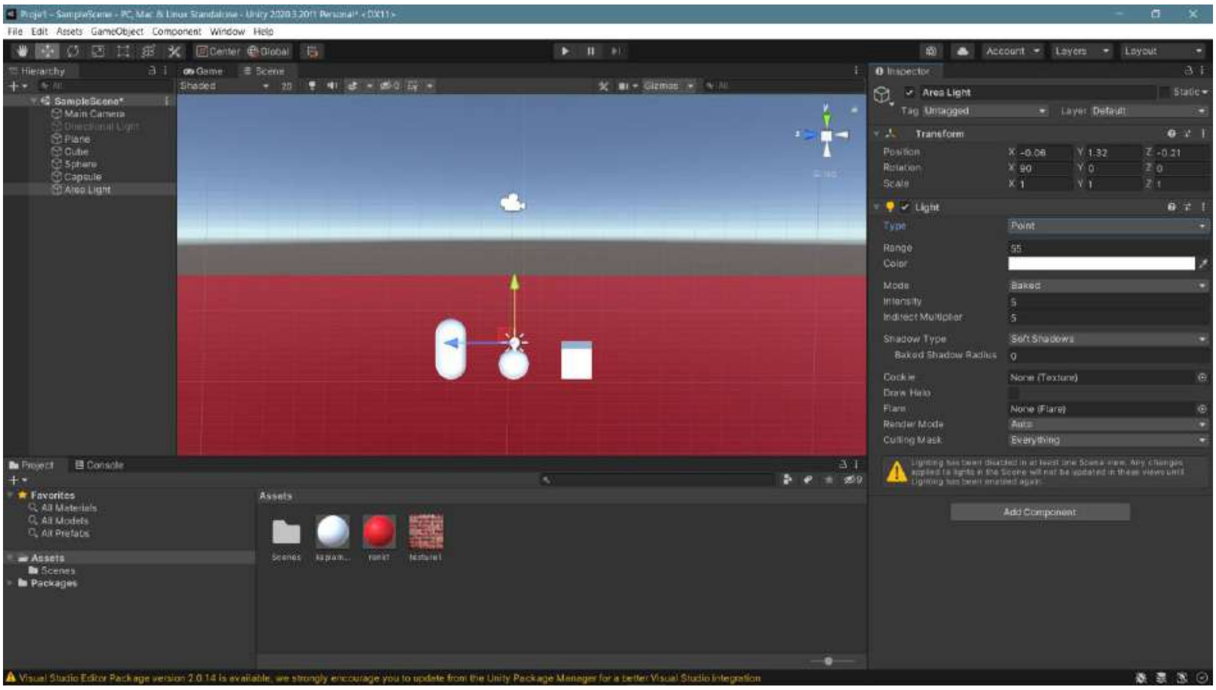
In this type of light, light application is achieved in a way that creates a cone.

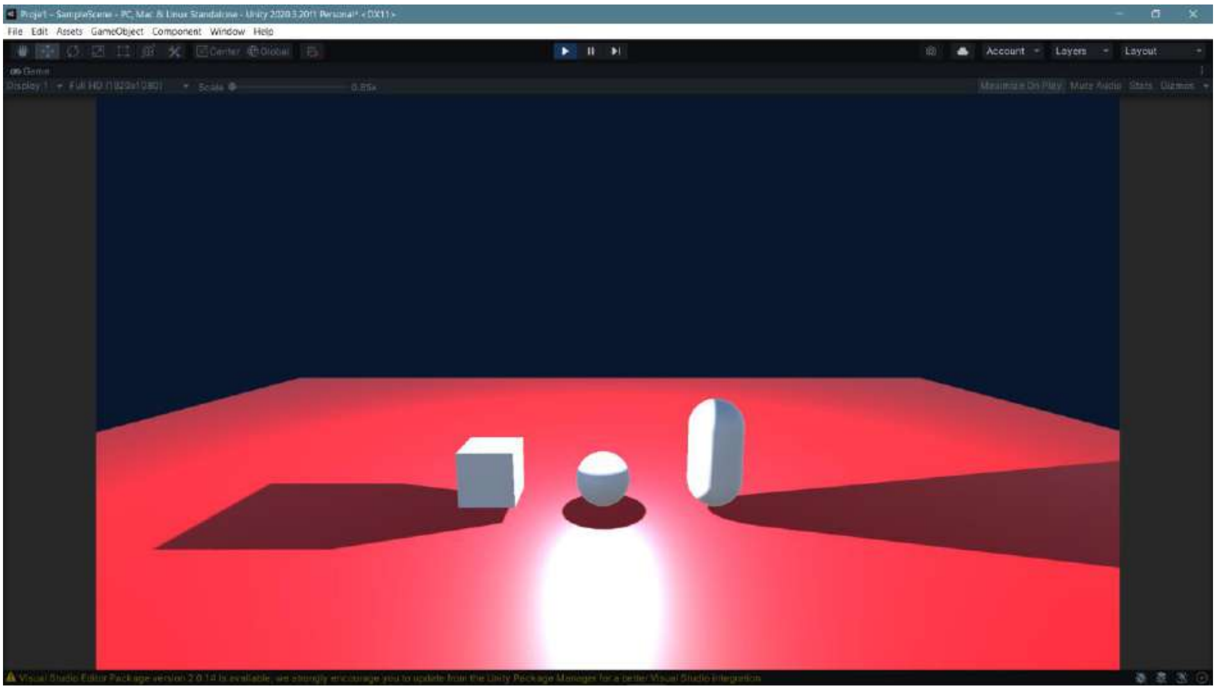




3.4.3 Area Light

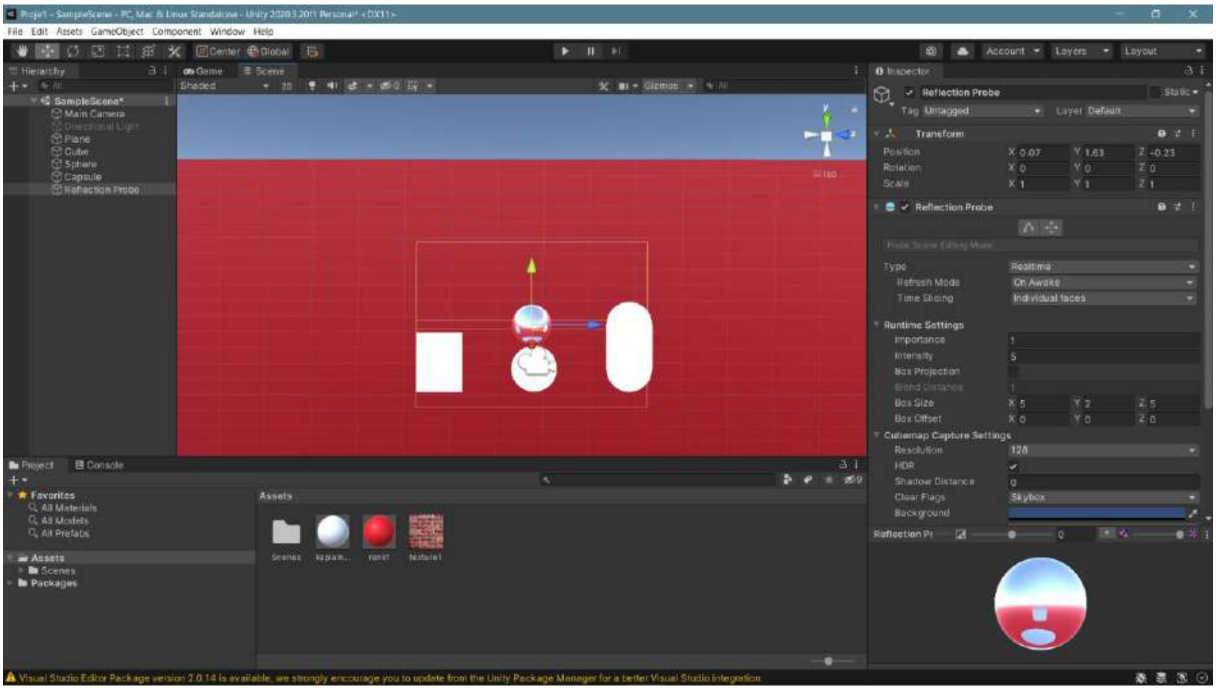
This type of light serves the purpose of illuminating a certain area. **Area Light** has **Spot**, **Directional**, **Point,** and **Baked** options as **Type** and produces different results.

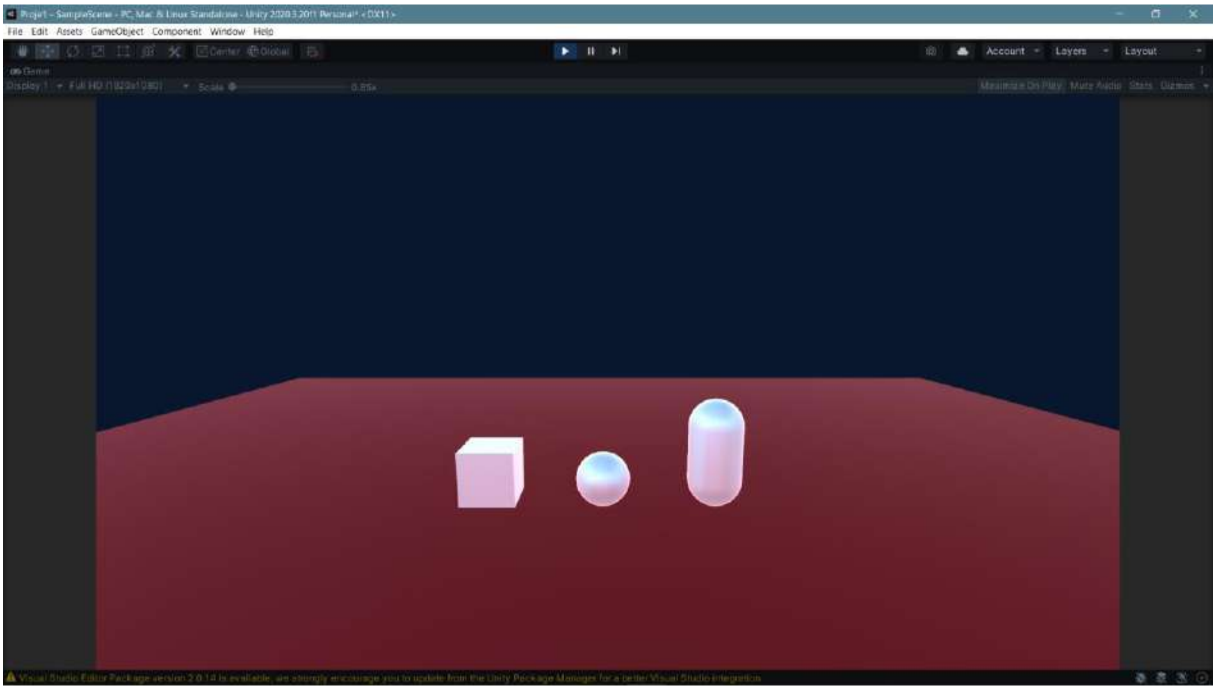




3.4.4 Reflection Probe

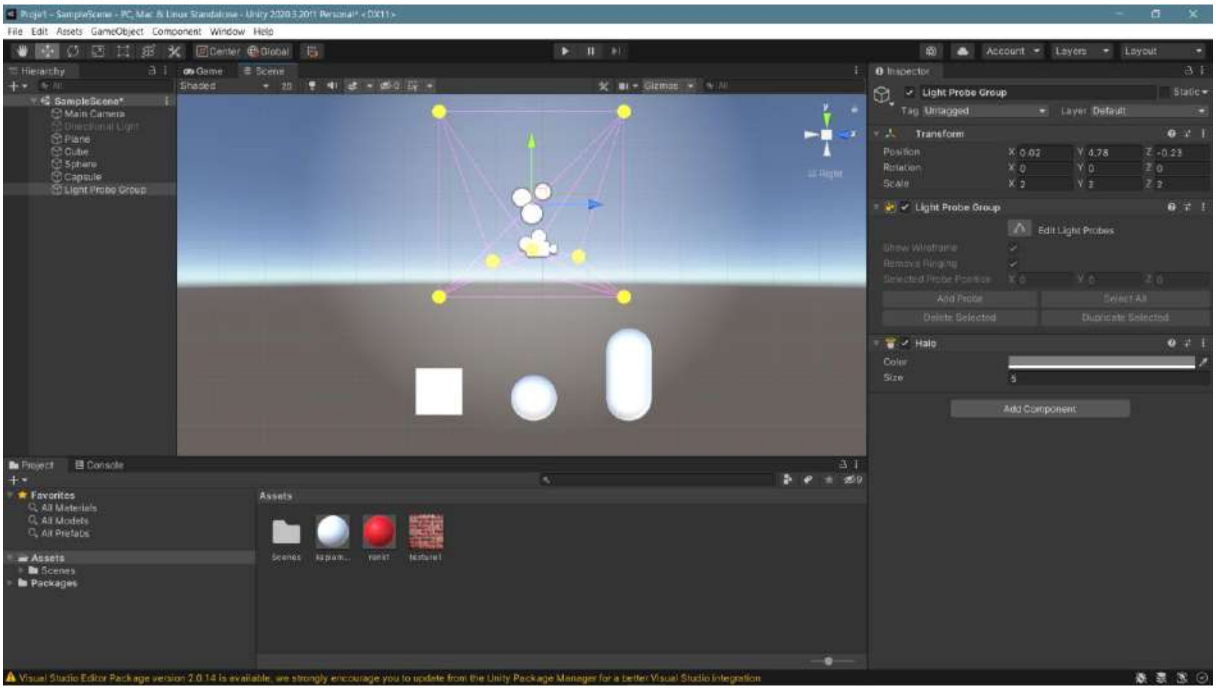
In this type of light, illumination and reflections are calculated via a probe.

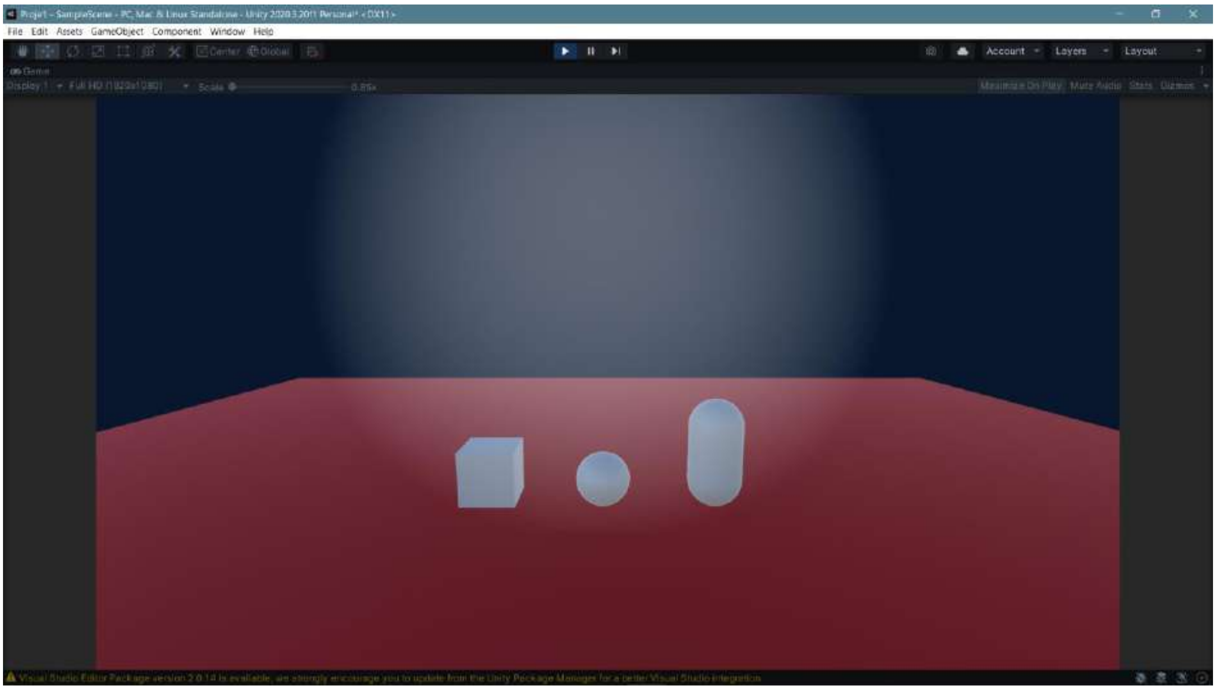




3.4.5 Light Probe Group

Grouped probes can be used in stage lighting. Here, **Halo** from **Effects** is also added with **Add Component**.



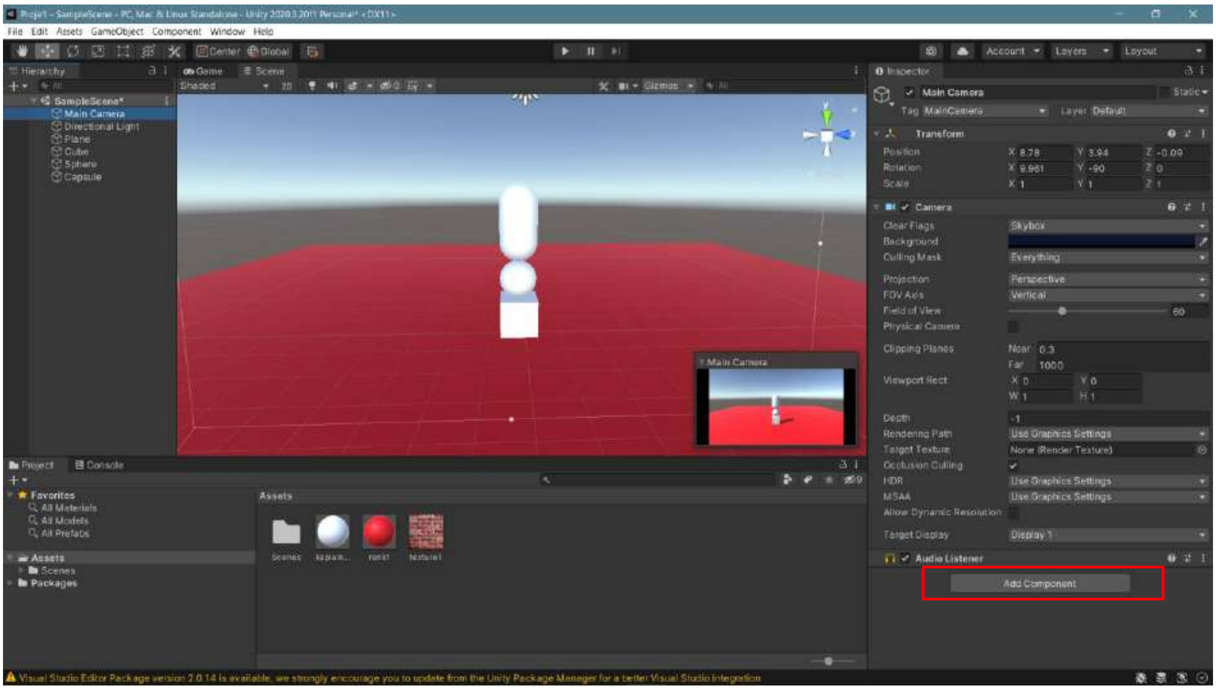


Lighting in Unity has very rich content and detail. For example, while creating a project, scenario-based lighting can be designed by using special lighting mode templates such as **Universal Render Pipeline** (**URP**), using **Post Processing** components, or by adding many special scene lighting features via **Package Manager** within the project. In future topics, these issues can be included according to the lighting needs of the project.

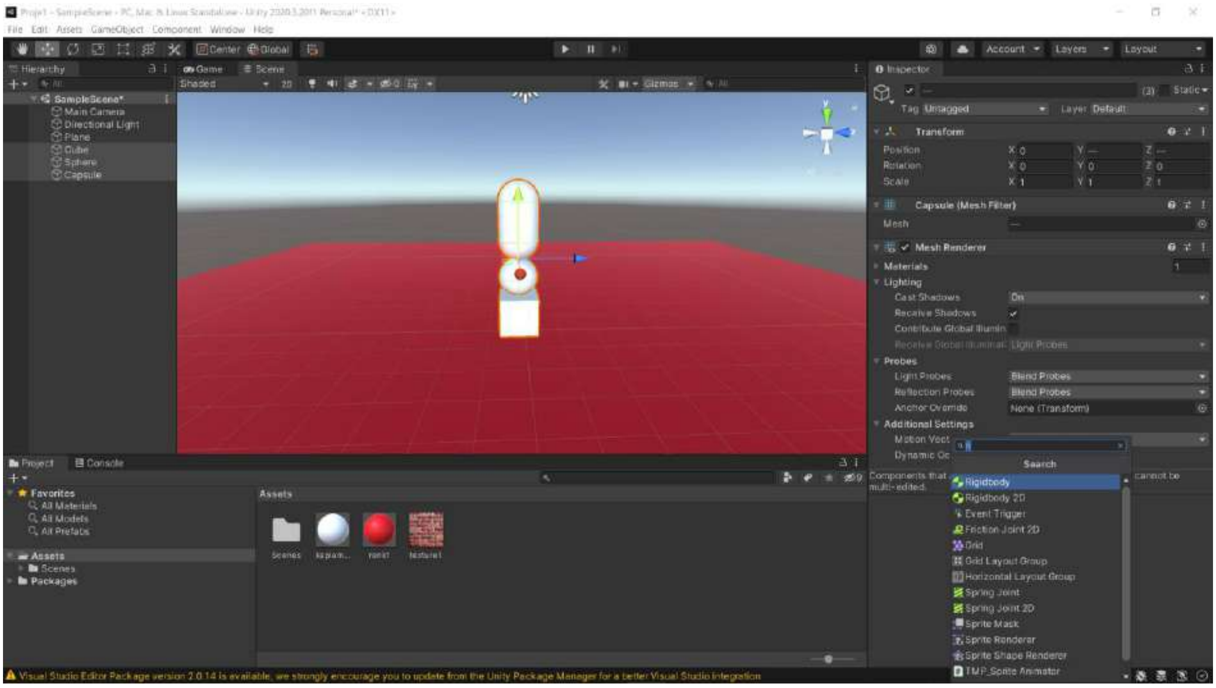
3.5.Rigidbody

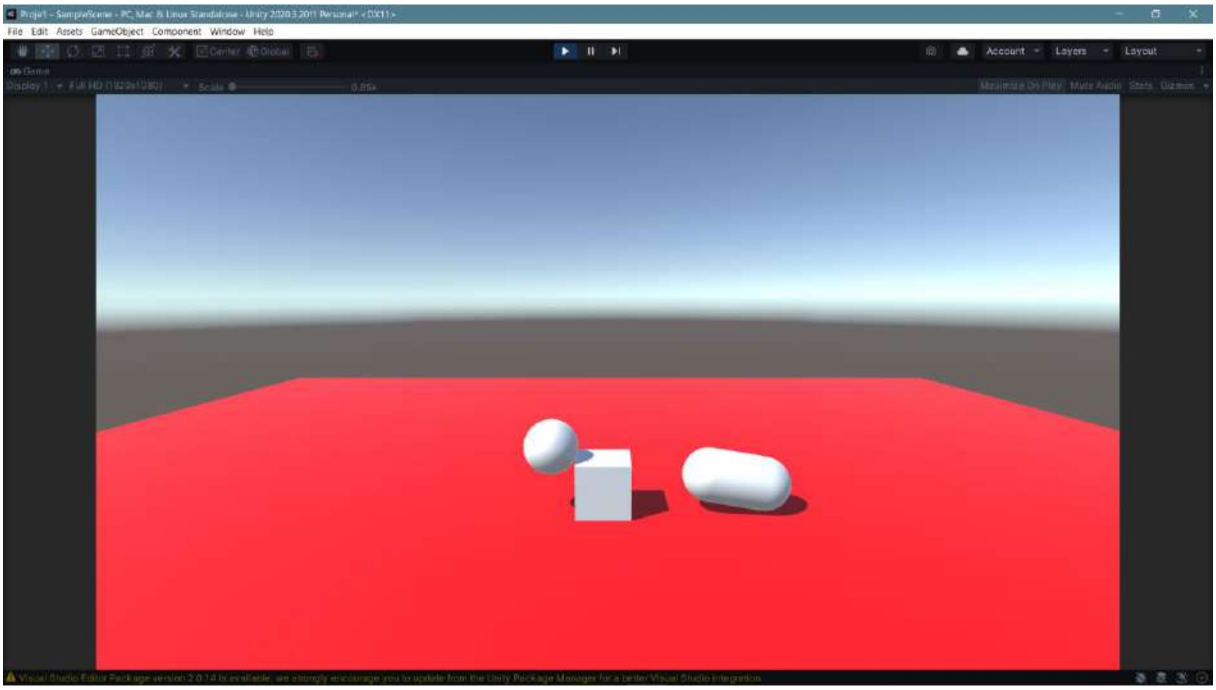
By giving physical properties to objects, they can be made to move in game mode. When an object has a mass like a solid object, it is expected to be affected by gravity and fall towards the ground accordingly or rise upwards in the opposite case. **Rigidbody** is added to the object's properties by clicking **Add Component** in the **Inspector** section of the object.

In the figure, the objects in the project are placed on top of each other. This image does not change when **Game Mode** is on.



However, when a solid physical body property is added to the cube, sphere and capsule with the **Rigidbody**, it will be seen that the objects move with the acceleration of gravity.

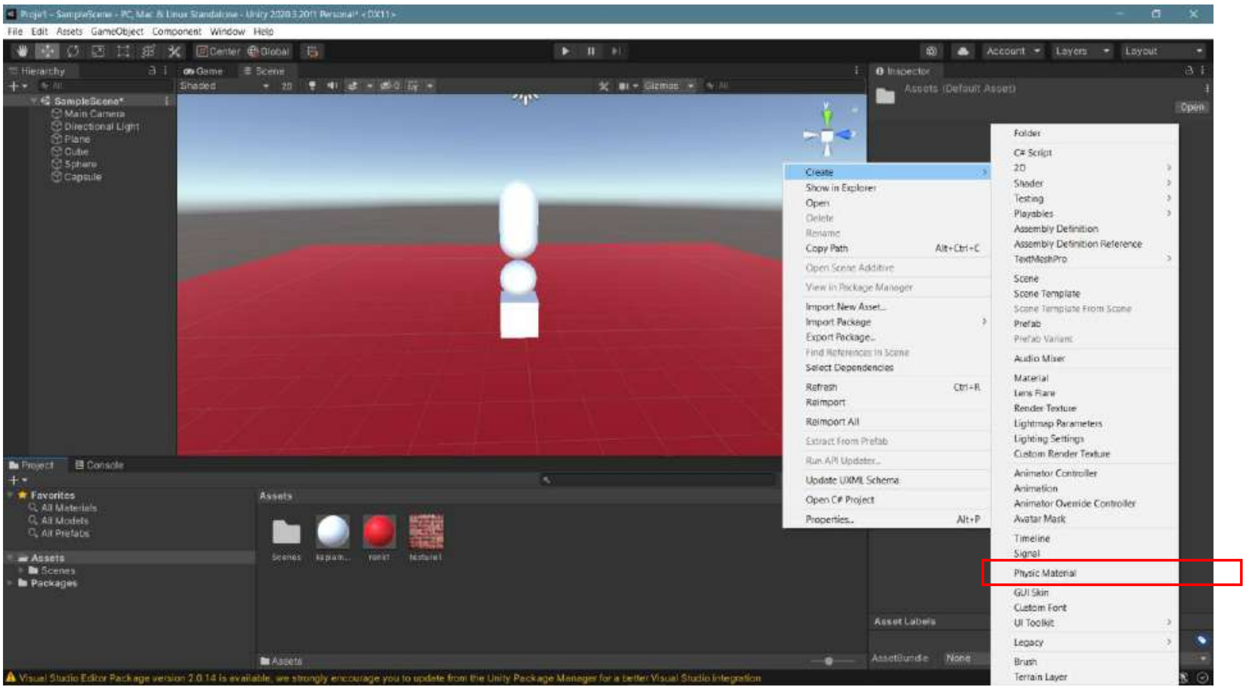




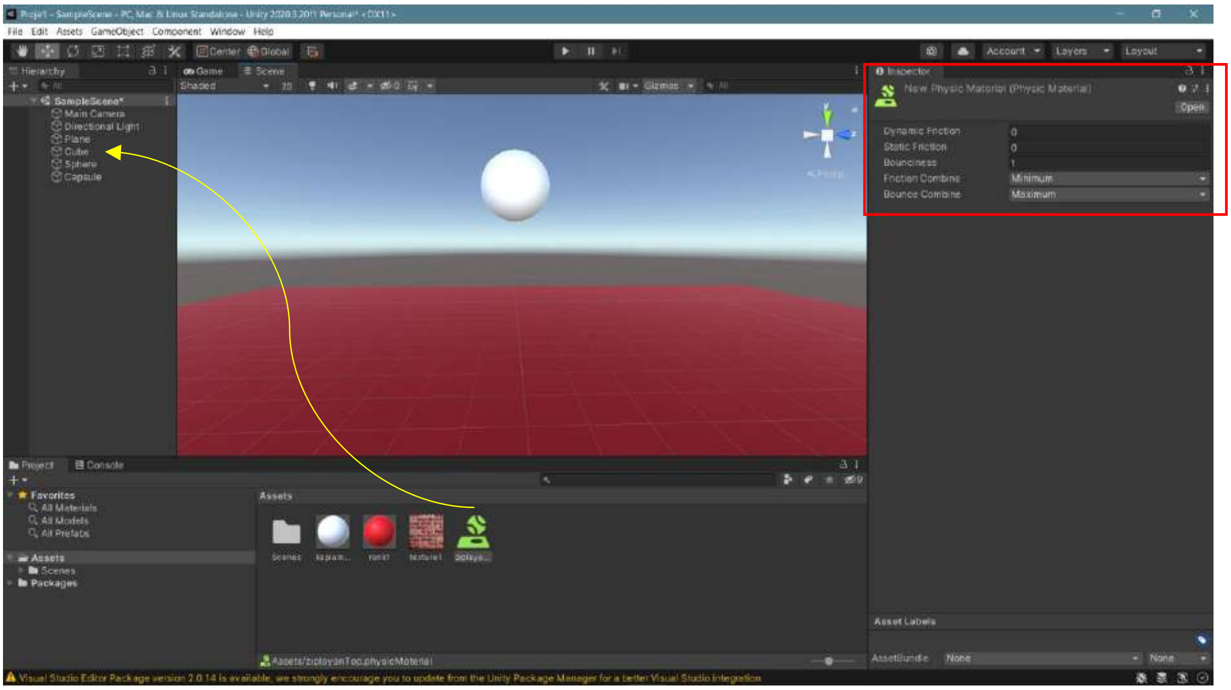
Here the capsule falls next to the cube, and the sphere also falls and continues to roll towards the end of the floor.

3.6.Physic Material

In order to move the physics properties to a higher level, it is necessary to define a special material type, **Physic Material**, and assign/bind it to the object. To do this, open the menu with the right mouse button in the **Assets** window. Select **Physic Material** from **Create**.



After naming the **Physic Material** as **ZiplayanTop**, let's edit its properties in the **Inspector**.

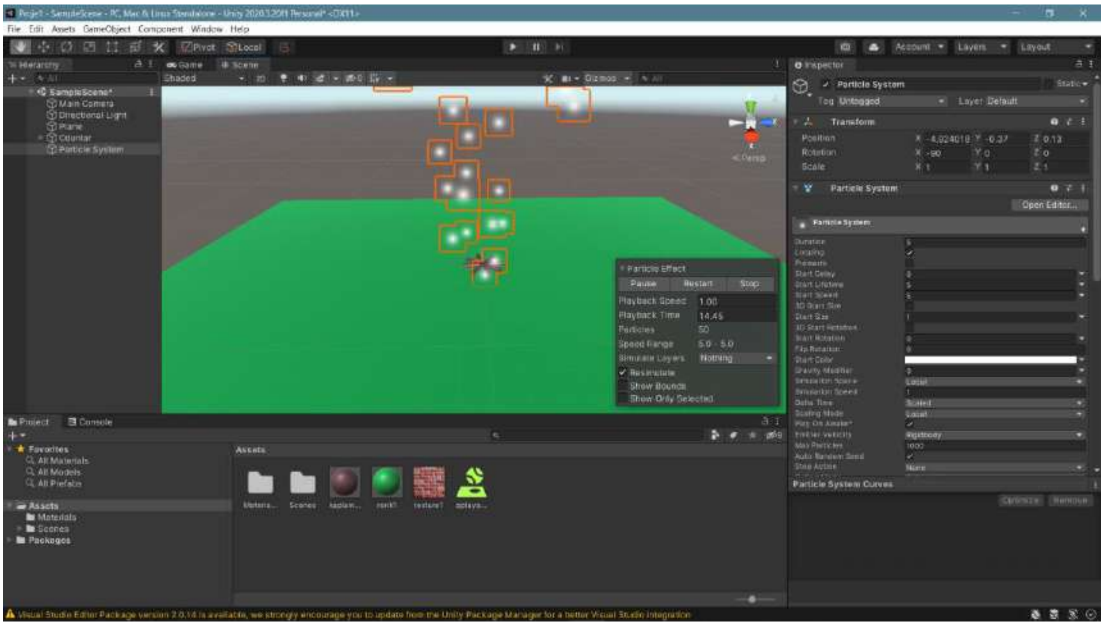


If we want the sphere to **bounce** continuously, we must change the **friction** settings of the physics material and drag this material to the **Sphere**, that is, to our sphere. In **game** mode, we can see that our sphere (ball) is constantly **bouncing**.

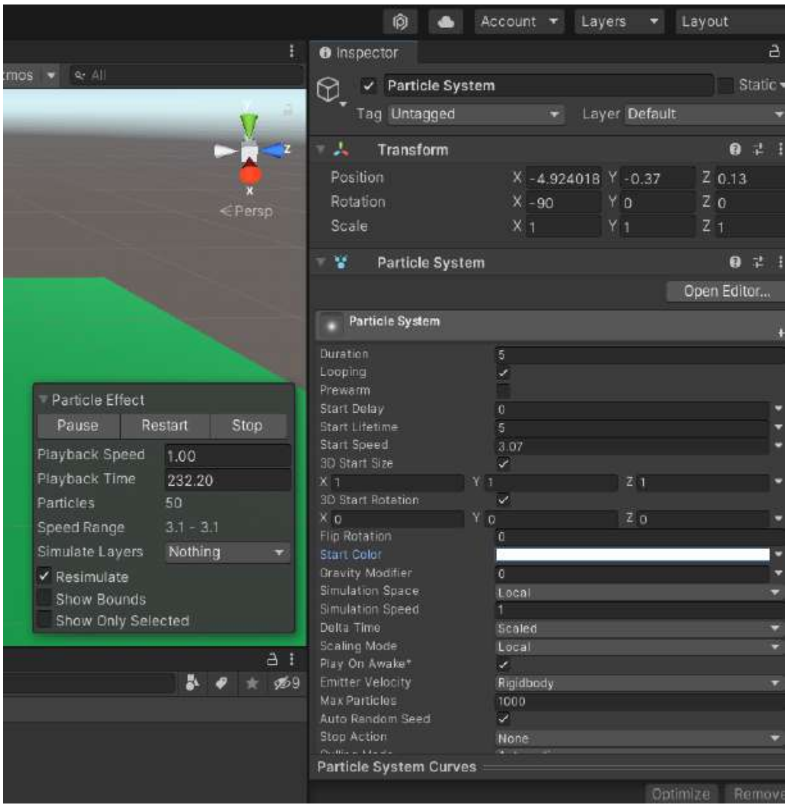
3.7.Particle System

One of the most functional objects is the **Particle System** – particle/particle system. In order to better understand the subject, it is possible to proceed with an example study in the form of fire burning. For this purpose, a few cylinders were laid on their sides and painted with brown material to make wood appear.

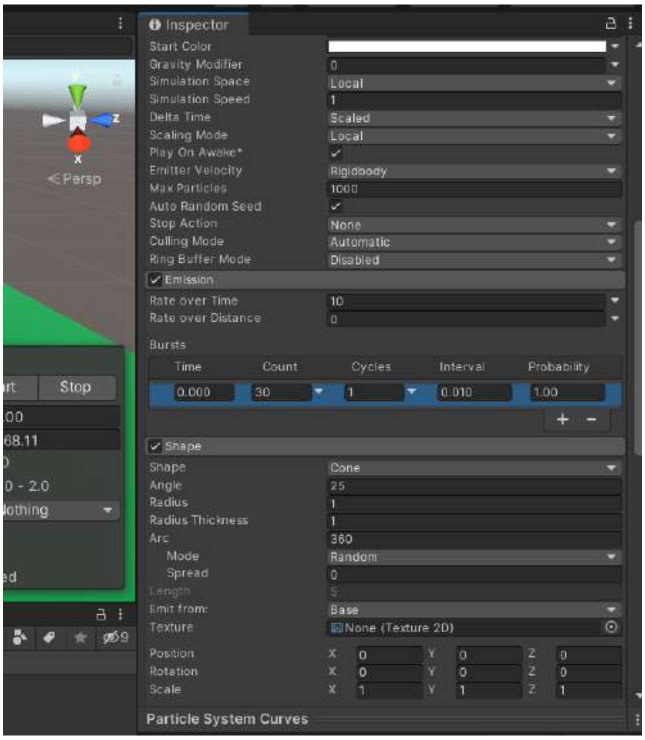
Right-click on our mouse in Hierarchy and select **Effect -> Particle System**. This object is designed to automatically scatter particles around.



There are many settings in the Inspector for **Particle System**. Let's examine the most important ones here. It is possible to go into more detail and apply them during the course.



Duration determines the time the particle will appear. **Looping** ensures that particle production continues in a cycle structure. **Start-Lifetime** regulates the particle life, **Start Speed** regulates the particle speed. **Start Color** determines the initial color of the particle. This color can be changed to a light red or orange color for a flame image. **Gravity Modifier** regulates gravity. A value of zero ensures that the particle rises as standard. Negative values ensure faster rise. However, positive values mean that the particles return to the ground. In some applications, if the particles are desired to fall, this value should be made positive.

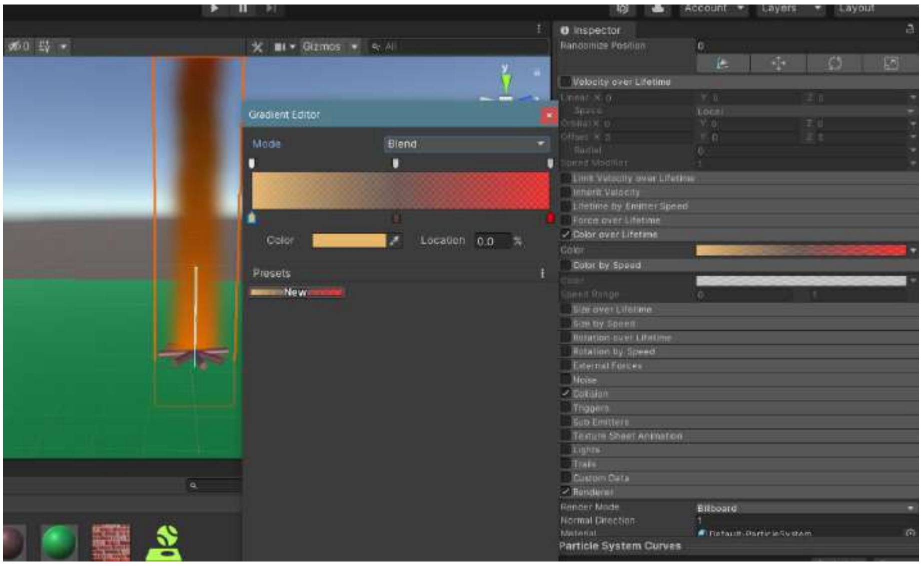


When we scroll down in the **Inspector**, we will see the **Emission** settings. R**ate over Time** determines the number of particles to be produced per second. In the **Bursts** section, it is possible to obtain an explosion effect by determining which number of particles will be released at which time with the **+** button.

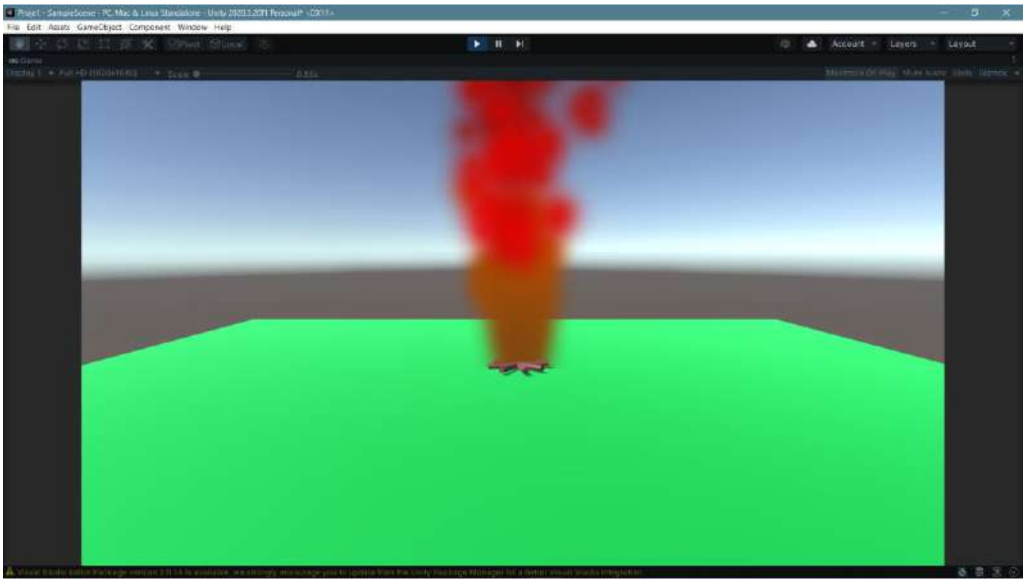
Shape determines the volumetric shape in which the particles will spread.

Color over Lifetime, **Renderer** and dozens of settings and their sub-settings can be changed to observe the results. Such changes can be made until the desired image is obtained.

In the **Color over Lifetime** setting, the color change and transparency level can be determined from the beginning to the end moment.

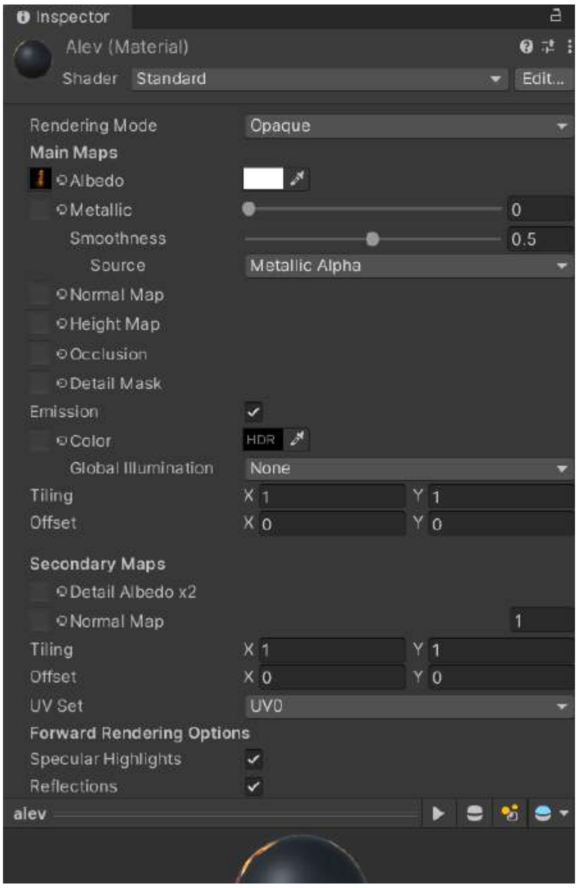


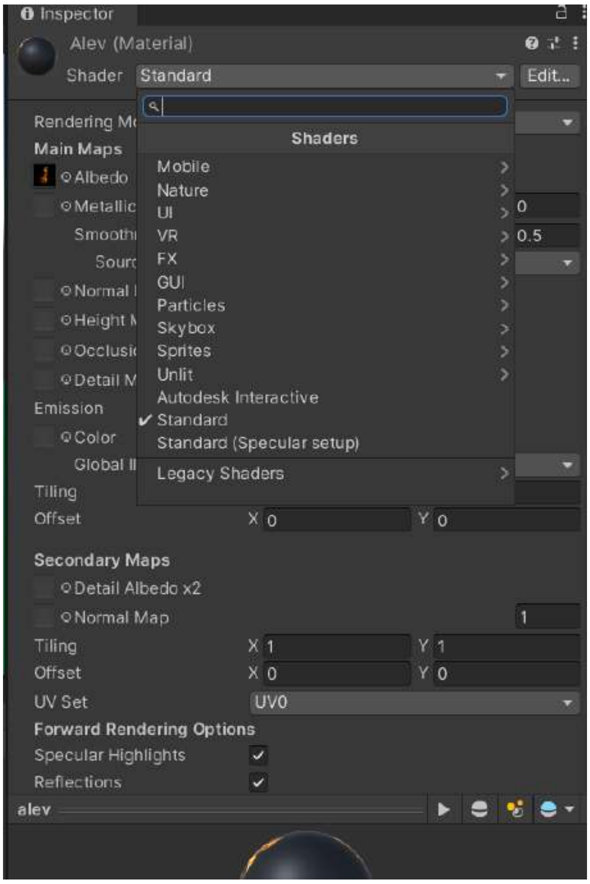
When specifying color, if **Gradient** - gradual change is selected, there are two settings in the window that opens: **Color and Mode**. The first and last color of the particles are determined by **Color**, and the **Alpha** percentage in Mode determines the transparency levels. If desired, the intermediate color and transparency level can also be assigned.

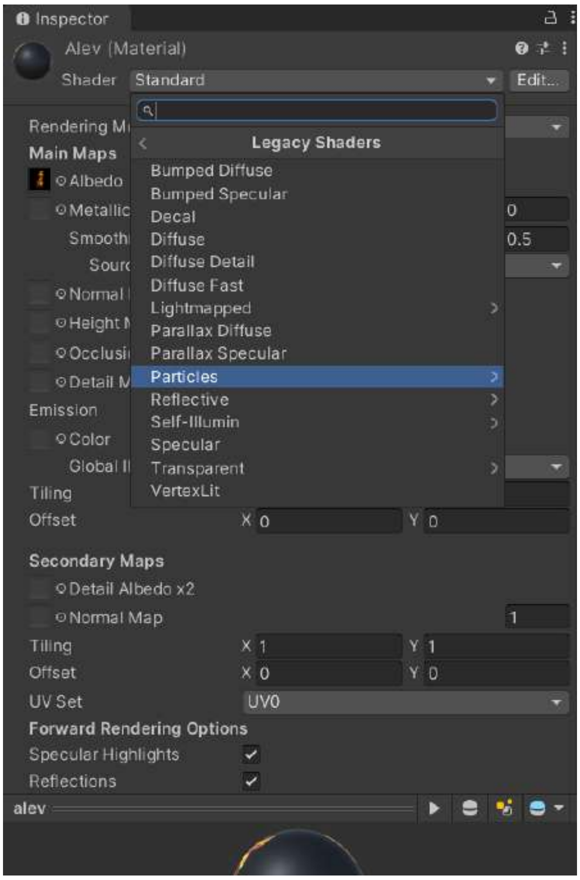


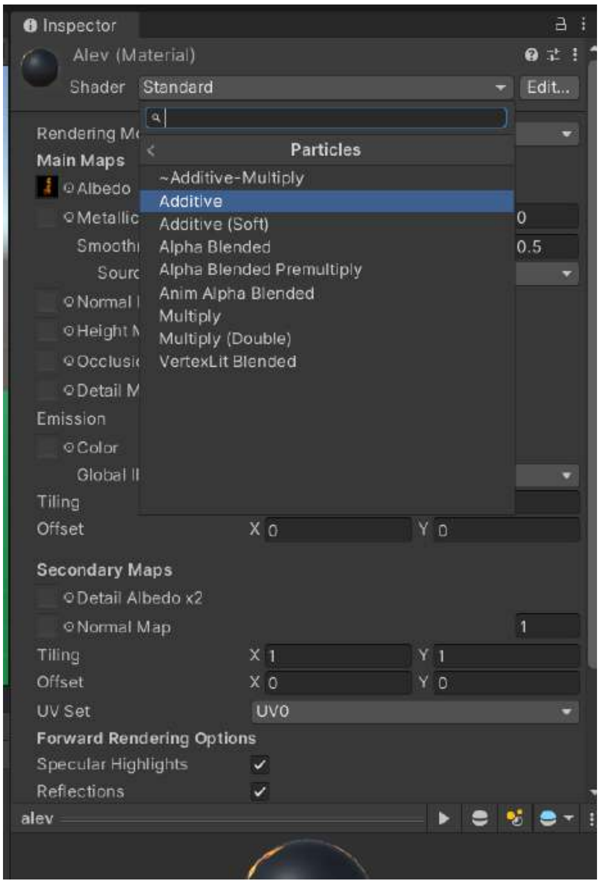
A flame image should be used to capture a more realistic image in the flame image. A flame image can be downloaded to the **Assets** section by searching for flame texture on **Freepik** and similar sites. Some presets are required to use this image.

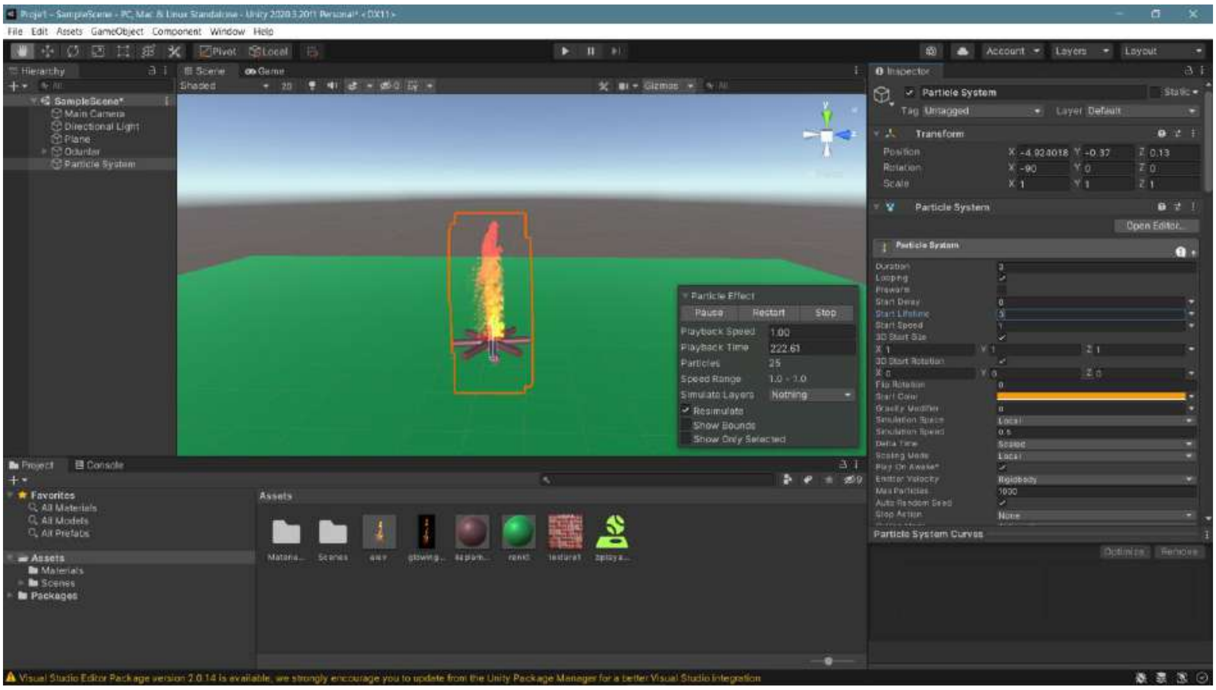
First, a flame image is placed in the **Albedo** of a **Material** in the **Assets** section. However, this image is set to **Shader -> Standard**. This needs to be made compatible with the **Particle System**. **Shader > Legacy Shaders>Particles>Additive** selections should be made for this. The result is more realistic.



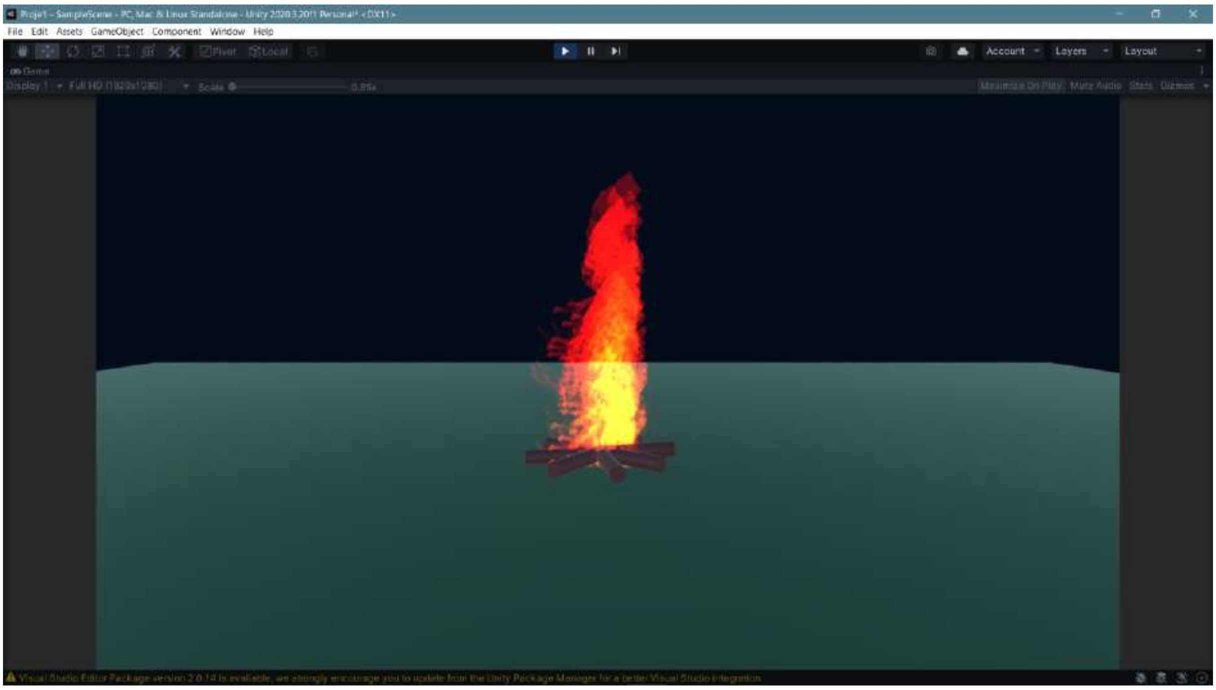








Or, from **Main Camera -> Clear Flags -> Solid Color** option, a solid color can be selected instead of the **Skybox** in the background. This will give us a different image.



With the **Particle System,** many different effects can be developed that produce particles downward, upward. **Lateral** directions depending on the scenario and need.

Another example is the scene of a truck that crashed into a mine, and its engine caught fire.

4. C# SCRIPT CODES FOR UNITY

This section contains introductory information for those interested in coding. Unity is a real-time development engine based on the **C#** programming language. This language is used in coding (script) in the developed computer, mobile device, VR/AR applications. This section covers the most basic C# coding that may be required within the course scope.

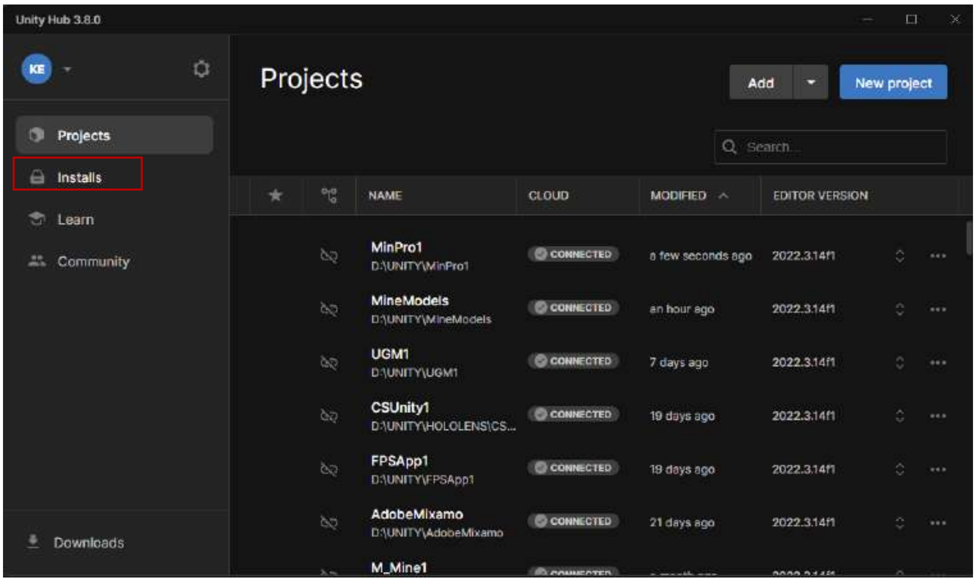
Let us state that our aim will be more about Unity-specific structures and the developed command library over the years rather than teaching the C# language. In addition, even if it is ready, code files will be used within the scope of the course. Therefore, Unity C# basics are briefly discussed to give an idea. Other coding information within Unity applications will be briefly emphasized.

4.1.Basic C# Information

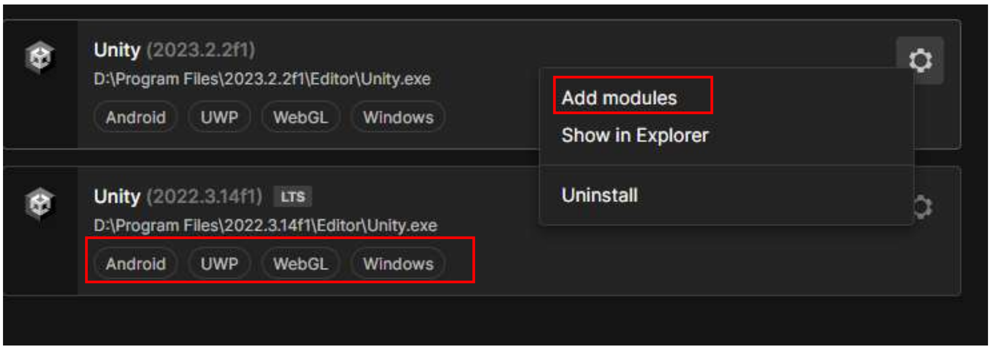
After the scene preparations, doing C# coding to develop various events will be necessary. If the codes are **not connected** to the scene and the **objects** that make up the scene, the game will **not work**. For this, both coding and connections must be known.

C# gives console and visual outputs. There is a window for **Console** outputs (DOS screen) in Unity. Visual outputs are already observed movements and actions in the game. With coding, we can access every object and its sub-areas and settings on the Inspector screen.

Visual Studio 2022 had already been installed in Unity for C# coding. However, if Visual Studio was not added while installing Unity, open the **Unity Hub** and go to the **Installs** section.

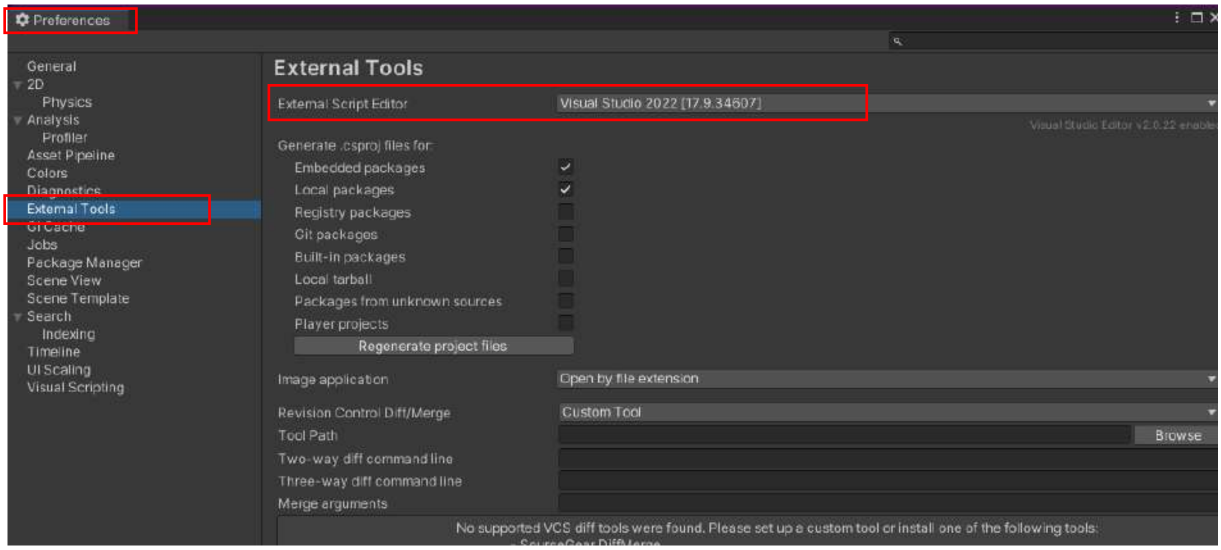


Select the **settings** button in the upper right corner of the Unity 2022.3 version window. From the window that opens, select the **Add Modules** tab.

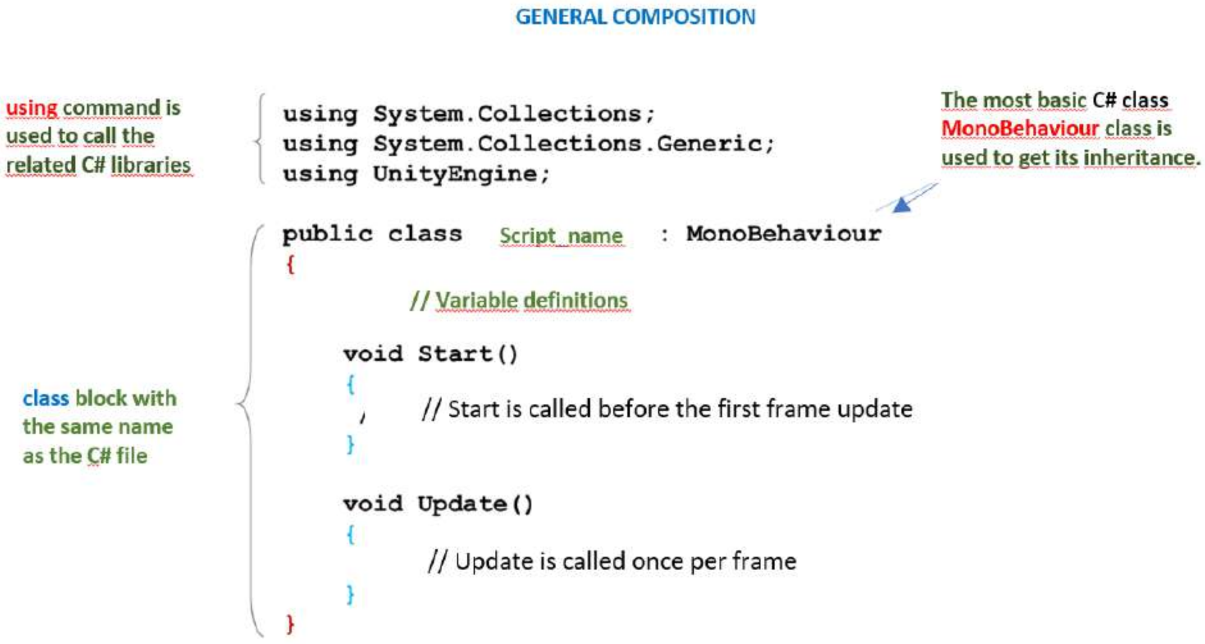


Here, Visual Studio will automatically be selected if it has yet to be chosen. After that, the installation process continues. Also, **Add Modules** is used when there is a module that you want to add that has not been installed before, such as the IOS module.

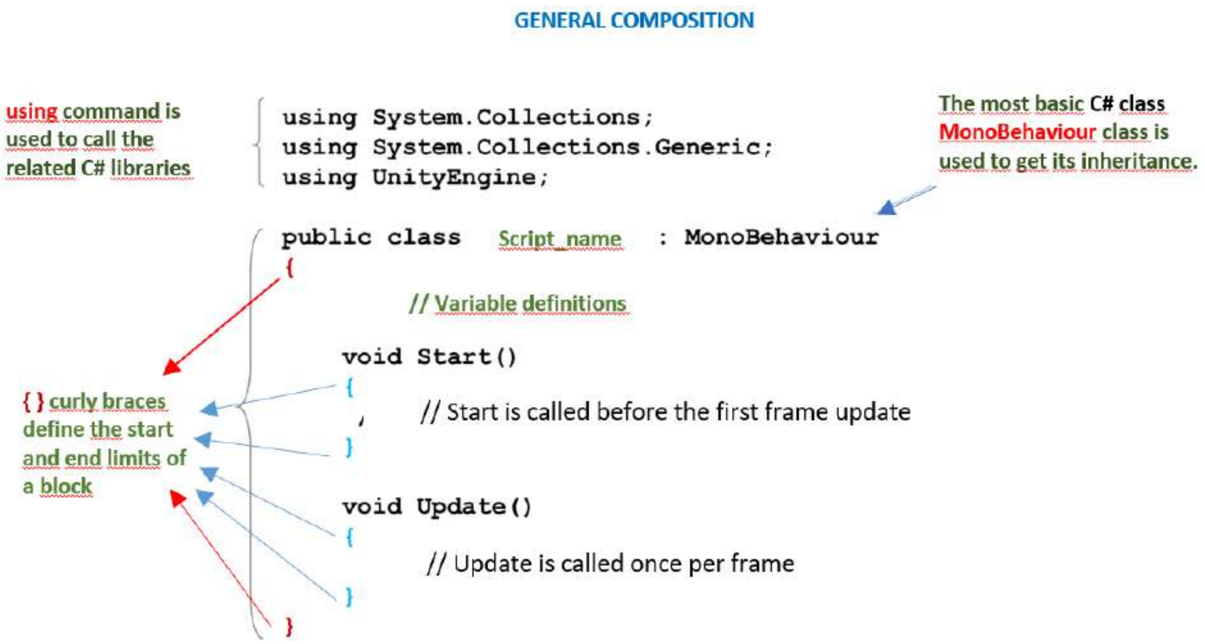
To access the code files from Unity, double-click on the script file is enough. However, a definition is required to do this if Visual Studio was added later. The connection is established via **Edit>Preferences>External Tools**.



The C# program opens with a certain **template** and is ready for the codes we will add. The script file's name is also the class structure's name. **Class** is the most basic building block of **object-oriented programming** and is an object that contains variables, methods and functions and is designed to be used and accessed repeatedly. Adding **MonoBehaviour** at the entrance inherits the most basic C# class structure found in the **C# library** with the opportunities it provides. Let's look at the general view and template.



C# codes-scripts consist of **blocks**. Commands other than library and variable definitions **must be written** in a **block** whose boundaries are defined by **{ }** braces.



Variables are used to carry parametric values inside or outside the block. Space is allocated in **memory** according to their types. These memory areas are assigned values according to their **numerical**, **verbal** or **type**.

The code developer determines variable names. While making a definition, many special characters (*i.e.*, *ö, ş, ü, ğ, ı, ç, à, ä, å, ã, ñ, ß, ë, ÿ*), spaces, and dashes, **cannot** be used. One of the most important elements besides variables is **functions-methods**. Commands are written into existing or created methods according to the character of the software to be developed.

Variables

In the operation z=x+y, x, y,and z are variables.

Variables are valued carriers allocated space in computer memory. They can be for any purpose, such as verbal, numerical, vectorial, etc.

```
int a=10, b, c;                  // integer
int[] dizi;                      // integer array
dizi=new int[20];                // allocation for 20 elements in the memory
dizi[0]=27;                      // assignment to the first element of the array
tamsayi=new List<int>();         // definition of list-type variable
tamsayi.Add(10);                 // assignment of 10 to the first element of the array
tamsayi.RemoveAt(0);             // removing the first element of the array
float x, y, z;                   // float number variables
string ad[30];                   // character type variables
bool basla=true;                 // variable that can be true or false
Vector3 koordinat;               // 3 dimensional vectorial coordinate
⋮
```

and many similar variable definitions special to Unity

C/C++/C# **statements** are **terminated** with **;**. Missing this character is an **error** source that will **prevent** the program from running.

Inclusion of library

class having the same name as the script

externally accessible variable

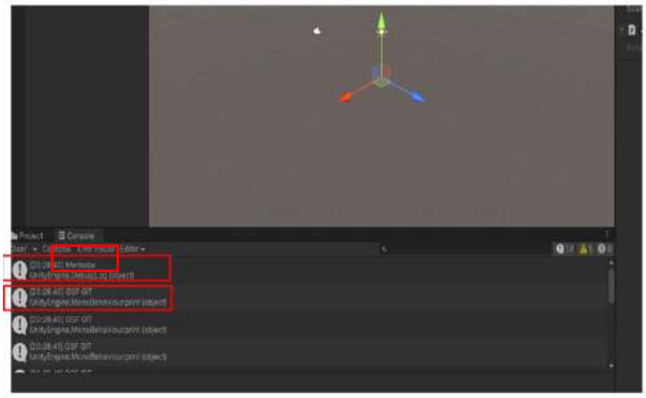
externally inaccessible variable

Method that will run once at the beginning

Method that will work during run time

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

public class KameraKontrol : MonoBehaviour
{
    public OyunKontrol oyunK;
    private float hassasiyet = 5f;
    private float yumusaklik = 2f;
    Vector2 gecisPos;
    Vector2 camPos;
    GameObject oyuncu;
    // Start is called before the first frame update
    void Start()
    {
        oyuncu = transform.parent.gameObject;
        //camPos.x = 12f;
        //camPos.y = 12f;
    }

    // Update is called once per frame
    void Update()
    {
        if (oyunK.oyunAktif)
        {
            Vector2 farePos = new Vector2(Input.GetAxis("Mouse X"), Input.GetAxis("Mouse Y"));
            farePos = Vector2.Scale(farePos, new Vector2(yumusaklik * hassasiyet, yumusaklik * hassasiyet));

            gecisPos.x = Mathf.Lerp(gecisPos.x, farePos.x, 1f / yumusaklik);
            gecisPos.y = Mathf.Lerp(gecisPos.y, farePos.y, 1f / yumusaklik);
            camPos += gecisPos;
            transform.localRotation = Quaternion.AngleAxis(-camPos.y, Vector3.right);
            oyuncu.transform.localRotation = Quaternion.AngleAxis(camPos.x, oyuncu.transform.up);
        }

        if (Input.GetKey("escape"))
        {
            Application.Quit();
        }
    }
}
```

The output of a script that gives **Console** output will be seen in the **Console** window in Unity.



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

public class Kodlar : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        Debug.Log("Merhaba");

    }

    // Update is called once per frame
    void Update()
    {
        print("GSF GİT");
    }
}
```

Message to the Console with Debug.Log()

Message to the Console with print()

Since the subject of coding is very comprehensive, existing coding examples will be repeated and better understood over time. Since this is not the main goal of this course, only quick and superficial information is included. However, it is useful to briefly explain two very common patterns such as **if()** and **for()**.

In the program flow, some operations will sometimes develop conditionally. **If**, **else** and **if-else** conditional statements are created to express this. General template:

if(condition)

{

command(s)

}

Here, the part inside the if block will be executed if the condition expression is true. Example:

int a, b=5;

if(b>0) { a=b*b; }

else pattern is used to code situations that only have two possibilities, while **if else** conditional sentences are used when there are more than two situations.

If the same process is repeated many times in coding, they are included in repeating patterns called cycles. Although there are different methods for these, the most commonly used is **for** block patterns.

for(initial value; final value; increment/decrement amount)

{

command(s)

}

Example:

```
x=0;
for (i=1; i<=100; i++)
{ x=x+i;
  print(x);
}
```

Here, the value of the integer variable **x**, which is initially *zero*, is increased *one by one* until it reaches *100*, and each value is printed in the **Console** window. If you notice, the variable **i** is used as a **counter** and also as an **increment** element in the command. In the block, **i=1;** is assigned as the *first* value, it is increased *one by one* with the command **i++;** and if it is *less than or equal* to *100* with the expression **i<=100**, the command(s) in the block are **executed** again. The increment and print command lines are written once in the **block** and executed *100* times.

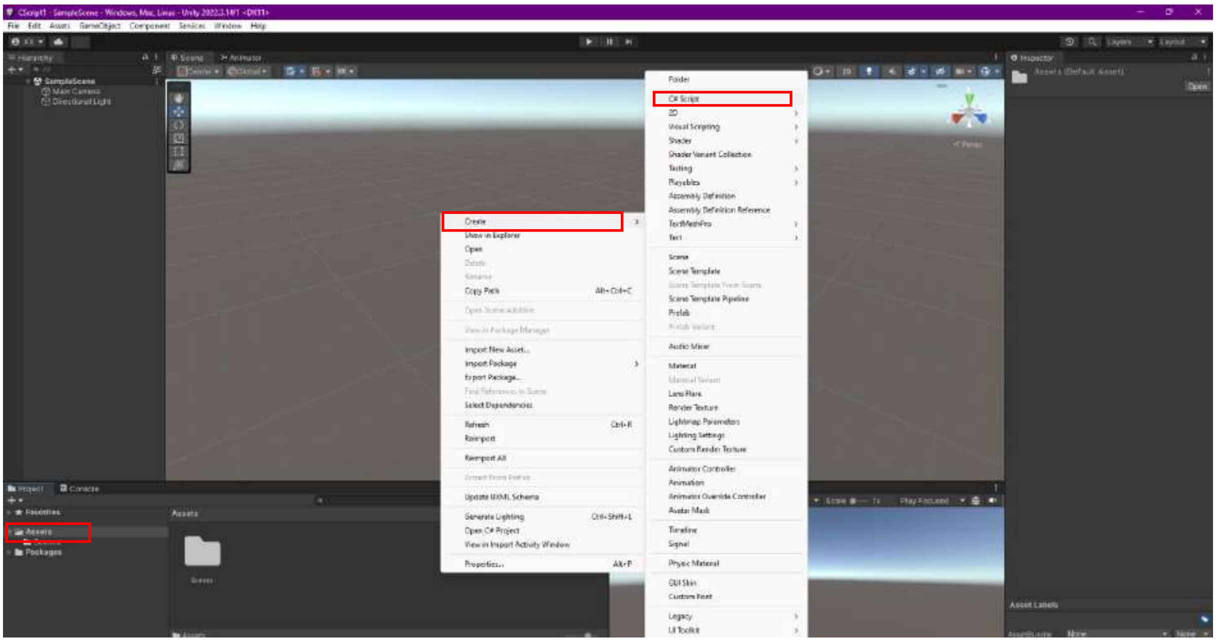
It will be possible to see C# codes in various trainings. One of the most important learning methods is to repeat these codes first and gradually enter the process of understanding, comprehending, interpreting and producing by developing basic knowledge.

After the basic part and code composition information, let's see the use of **C# Script** in our Unity project.

Since the C# Unity library has a large volume, let's start from the simplest level of codes and progress by adding new information according to specific needs.

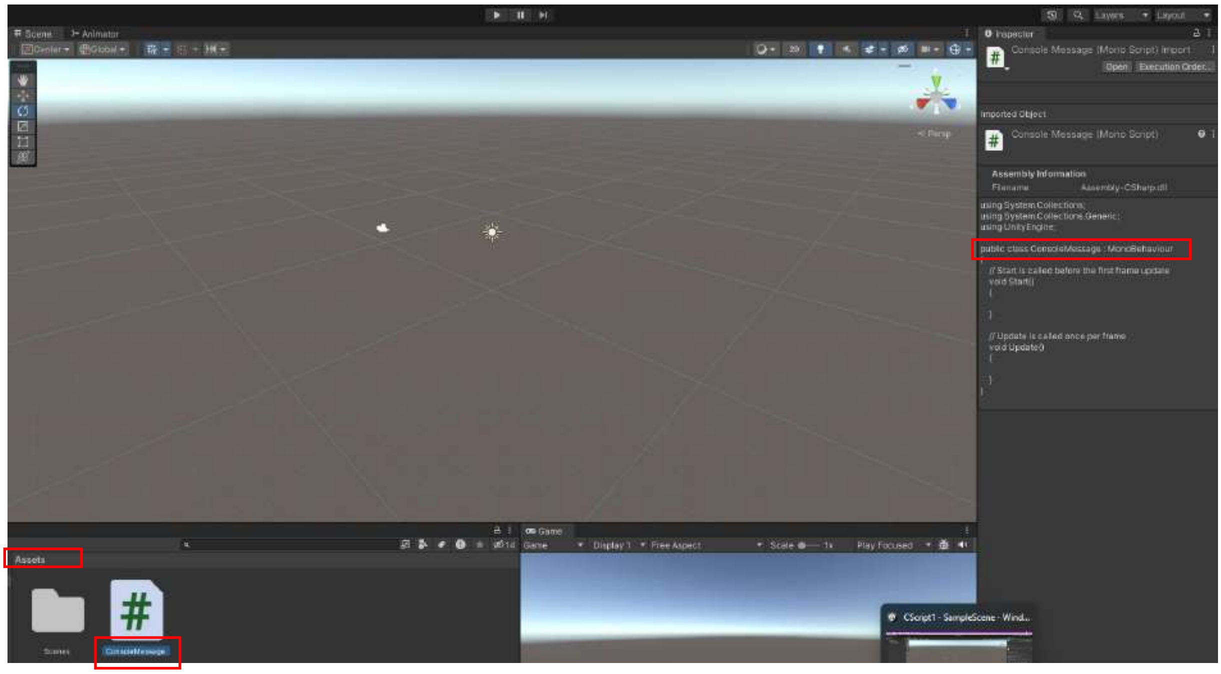
4.2.C# SCRIPTS IN UNITY

Let's start seeing how C# information can be used in Unity. For this, we will create a new project. Create a code file with **Create>C# Script** by clicking the right button of the mouse in the **Assets** section.

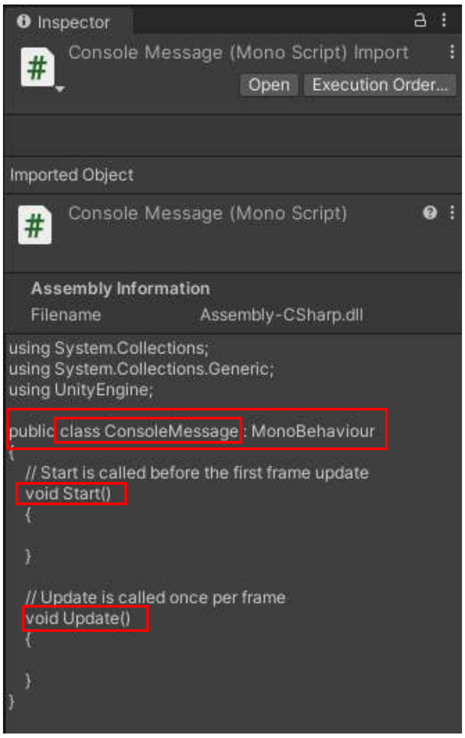


The point to be noted here is that special characters should not be used in file naming, similar to naming web pages. In our study, the **C# file** was named **ConsoleMessage.cs**. The file will be opened in the **Assets** window with the standard **C# template**, and accordingly, the **class** in the program will have the same name as the file name.

It is very important that the file naming process is synchronized with **Create>C# Script** and that the **class** name in the created file has the same name. If the file name and class name are not the same, the program will be **incorrect**. If, by mistake, the C# file name and the class name are different, the class name must be <u>changed</u> and **matched** according to the file, or the file name must be changed according to the **class** name.



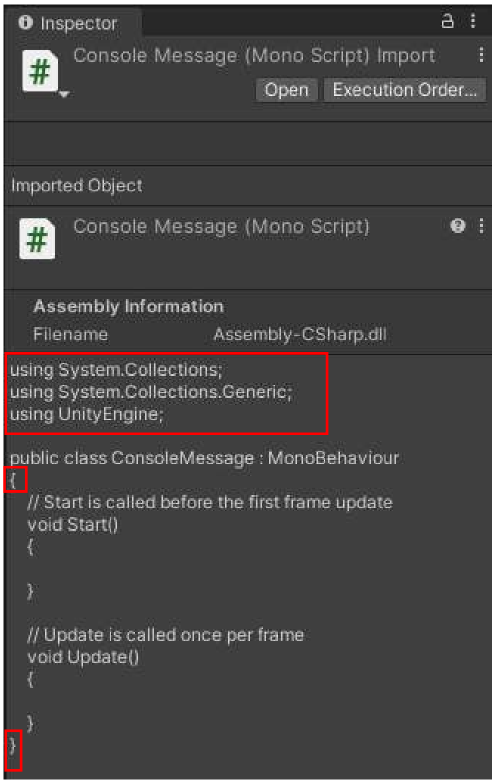
The file content is displayed in the **Inspector** section. This template code consists of a **class ConsoleMessage** block and two methods (functions) named **Start()** and **Update()**.



The C# code sentences to be used in the program are written inside the **class**, and method (function) blocks are created below it.

As seen here, the **methods** (functions) are automatically named within the framework of this template in which the program will be written, and their boundaries are determined with **{…}**.

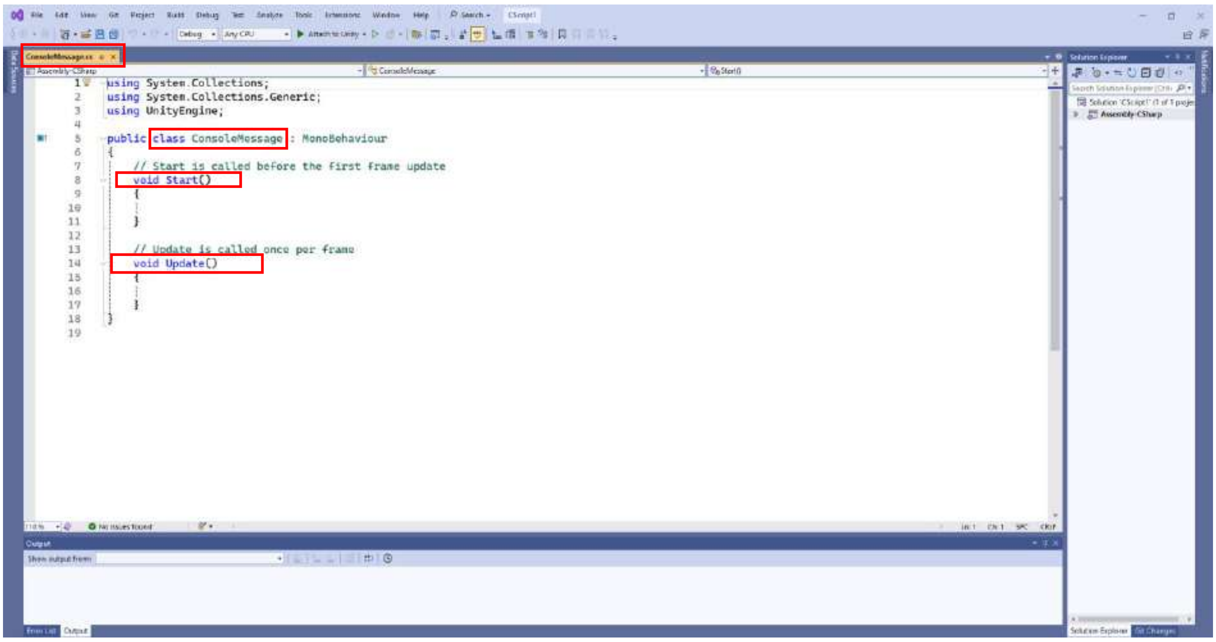
In order to access the commands in the library, the program requires using the files that contain them. Therefore, our template starts with **using …** statements and access is provided to the library commands.



Now let's start developing simple programs on this template.

4.3.Console Messages

As the name of the project suggests, our program aims to send messages to the console window of the Unity editor. In other words, messages will be sent to the screen in the codes, and these will be displayed in the Console window. Let's double-click on the **ConsoleMessage.cs** file and open it in Visual Studio 2022.



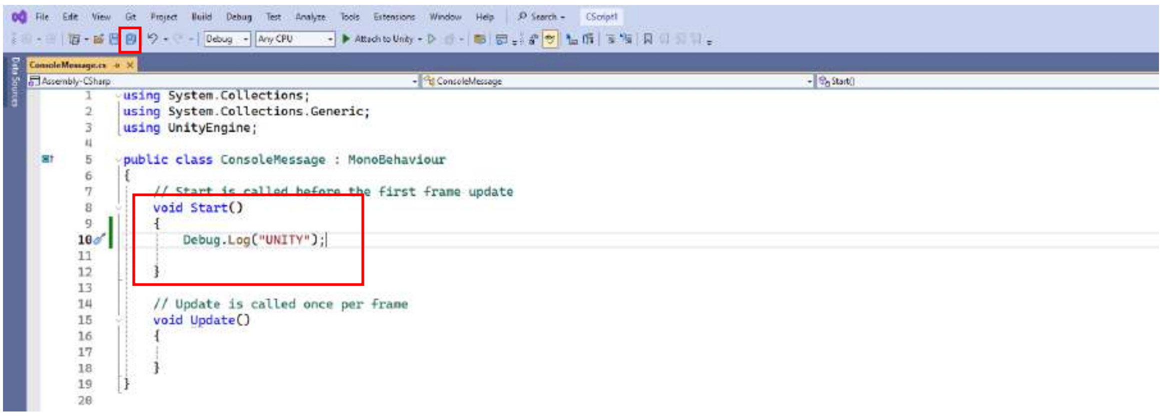
The content we see in the **Inspector** is seen in the compiler editor for processing. **Start()** method becomes active when the program runs and runs once. **Update()** runs 24/60 times per second, depending on the screen frequency. Here, it will be enough to write it in **Start()**.

Debug.Log() command is used to send a message to the screen (**Console**). **print()** command can also be used for the same purpose.

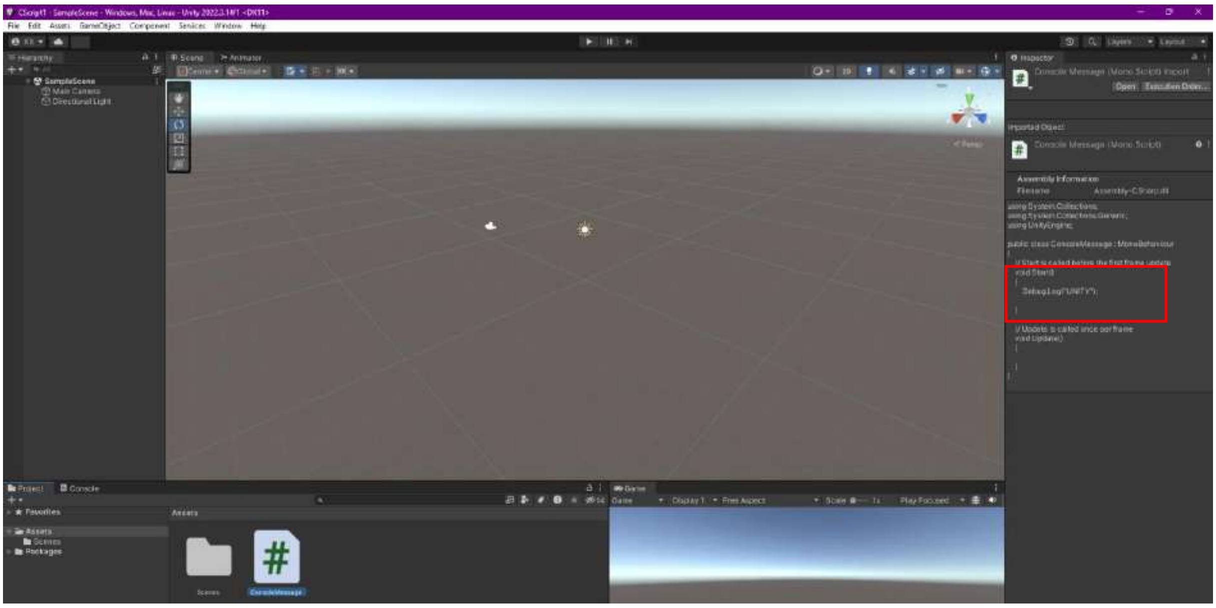
To use commands, we can write our message between two **"…."** For example:

Debug.Log("UNITY");

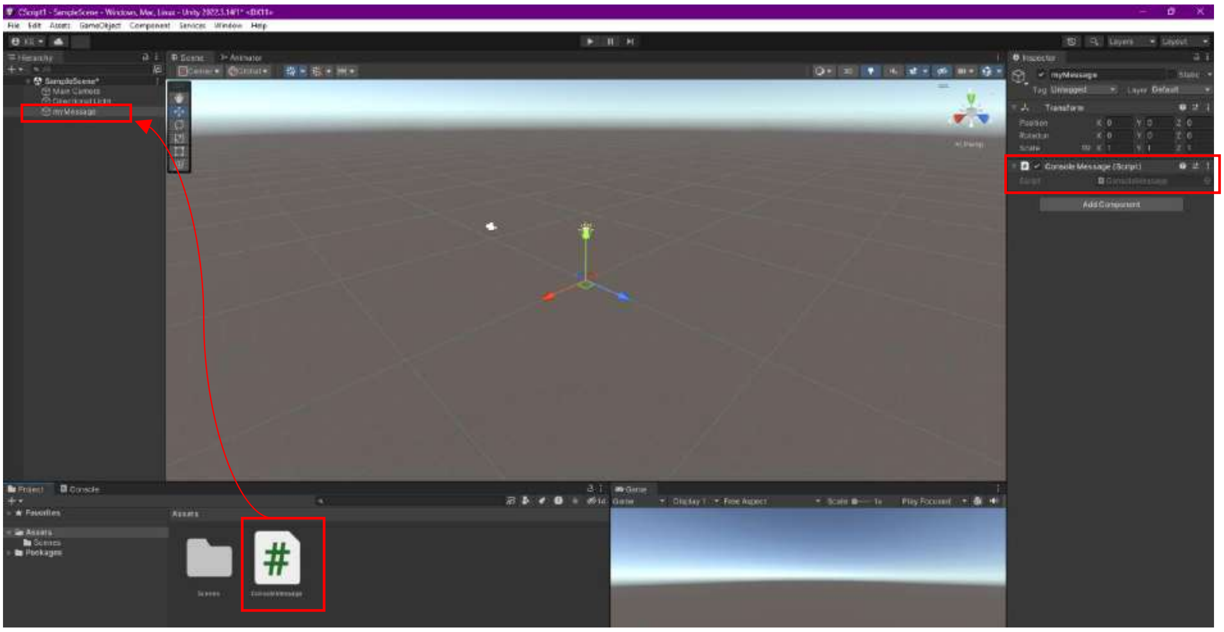
Command sentences must end with **;**. Write the command and save it by pressing **Ctrl+S** or clicking on the icon .



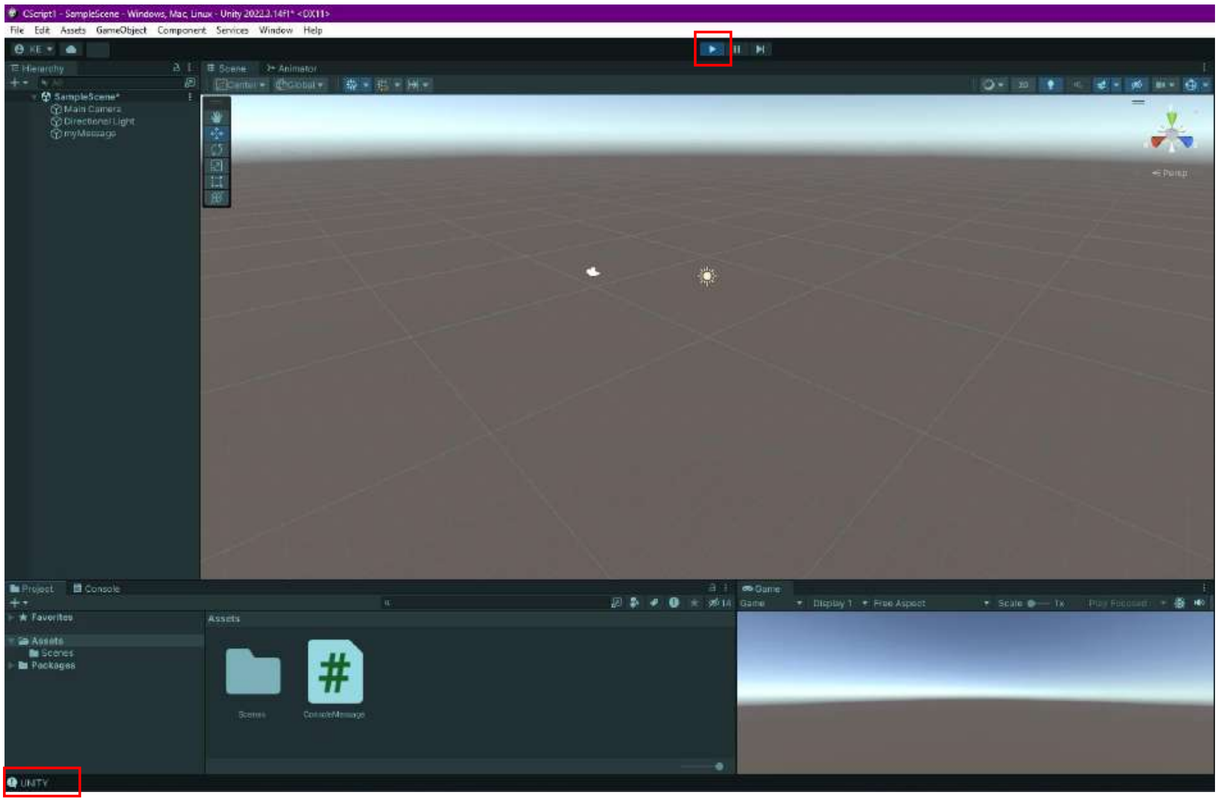
When we return to Unity, this change will also be visible in the **Inspector**.



In order to see the result of a line we added to the template, the project and the code file must be linked. For this purpose, let's click right and add an empty object with **Hierarchy>Game Object**. This object is named **myMessage**. Let's drag the **ConsoleMessage.cs** file and link it to this object.



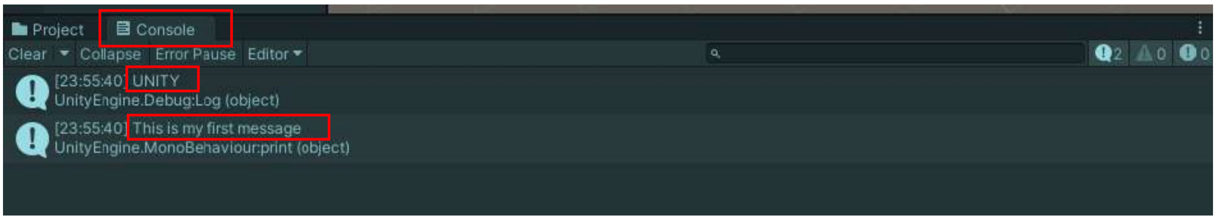
Now, click on **Play Mode** and see the message we wrote at the bottom of the screen. The message can also be viewed in the **Console** section.



Add another line to the program. This time, see that we can also transfer the message with the **print()** command.

```
void Start()
{
    Debug.Log("UNITY");
    print("This is my first message");

}
```

Save and return to Unity and see the result in the **Console** window in **Play Mode**.



4.4.Variables

As stated in the general definitions, program variables are program elements that will carry numbers or various values. Variables are **memory location** names that are allocated space in memory and can take on multiple values within the program flow, and these assigned values can change.

Variables have types according to the information to be assigned and are categorically defined as **public** and **private**, which basically means that they are *open* or *closed* to **external access**.

Immediately following this definition is the **declaration type**: **int** (integer), **float** (real number), **string** (character), **Vector3** (3D vector coordinate information), etc. For example:

public int sayilar;

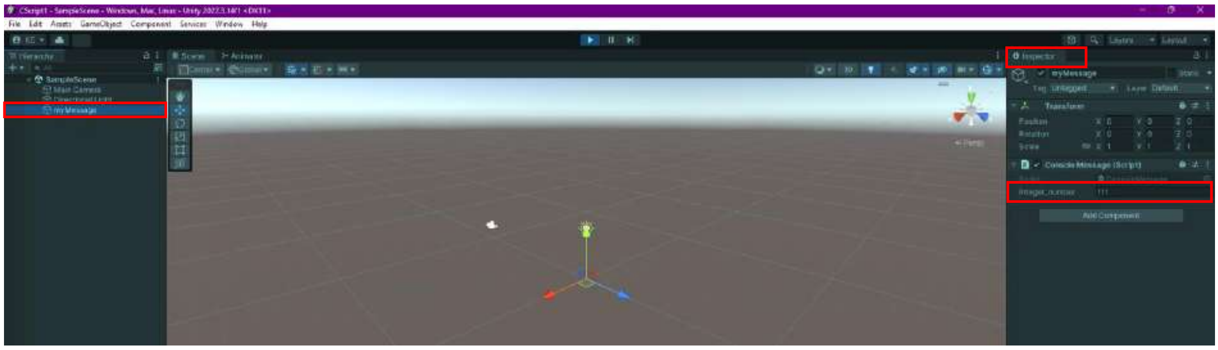
private float angles;

Within the framework of visual programming logic, it is possible to define a variable for each parameter in the **Inspector** section of the editor and assign values to these fields. A variable is normally considered **private** unless it is described in front of another definition.

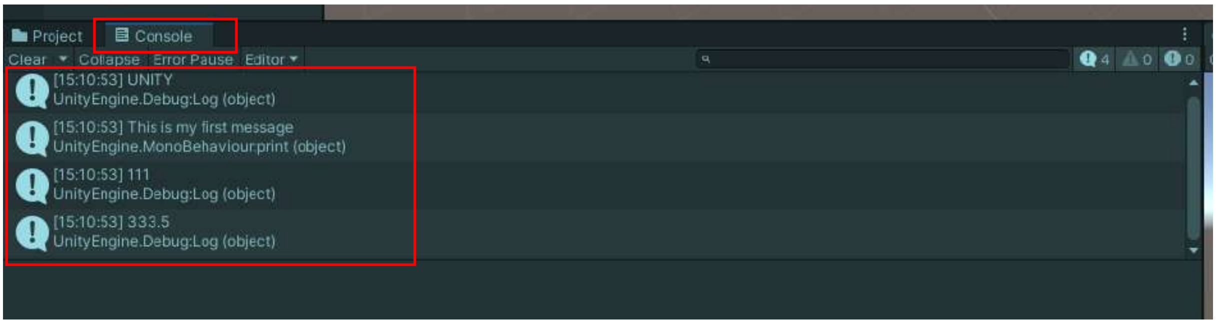
Let's do some applications in our project. Define two variables named **integer_number** and **real_number**, the first one is an integer, and the second one is a *real* number. Also, assign the initial values to the memory in the definition line. In real number assignments, the expression **f** is added to the end of the number, meaning **float**.

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

public class ConsoleMessage : MonoBehaviour
{
    public int integer_number = 111;
    private float real_number = 333.5f;

    // Start is called before the first frame update
    void Start()
    {
        Debug.Log("UNITY");
        print("This is my first message");

        Debug.Log(integer_number);
        Debug.Log(real_number);

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

Now, save and return to Unity and run **Play** Mode. If you pay attention, you can see that the publicly defined variable has become accessible to outsiders in the **Inspector**. It is also possible to enter a new value in this field.

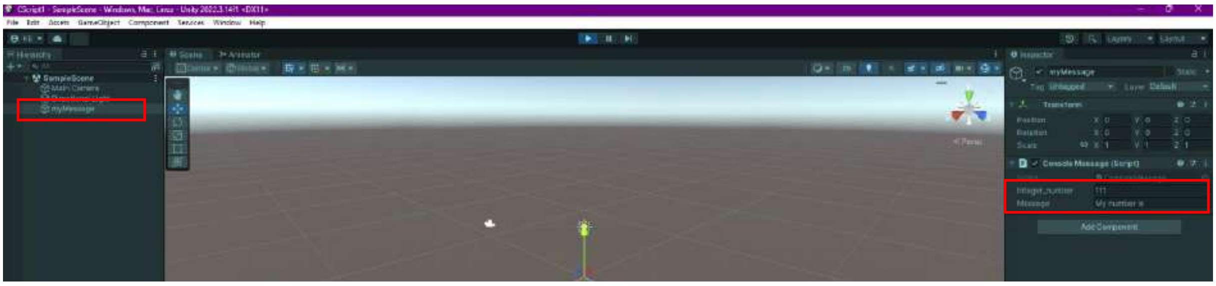


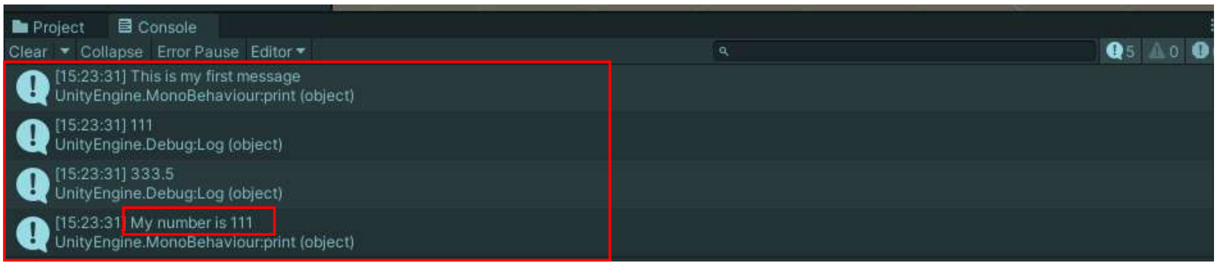
The results of the four message commands in the program will be displayed in the **Console** window.



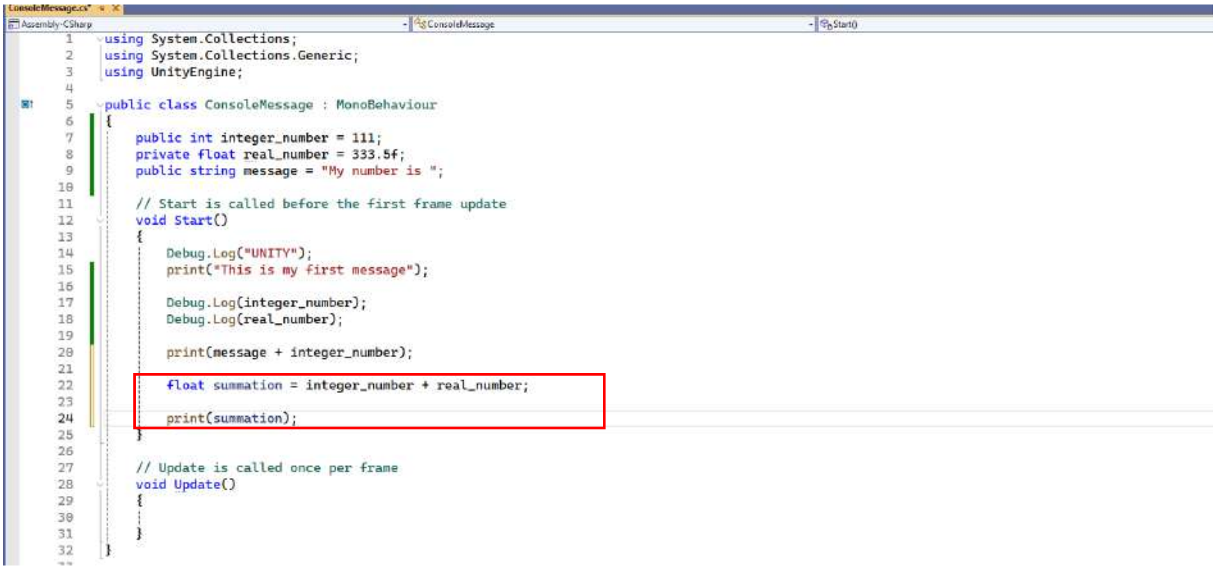
A character (**string**) variable can also be combined with numerical expressions and displayed on the screen. The **+** operator used here is not only for arithmetic addition but also for using two values together.

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

public class ConsoleMessage : MonoBehaviour
{
    public int integer_number = 111;
    private float real_number = 333.5f;
    public string message = "My number is ";

    // Start is called before the first frame update
    void Start()
    {
        Debug.Log("UNITY");
        print("This is my first message");

        Debug.Log(integer_number);
        Debug.Log(real_number);

        print(message + integer_number);

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

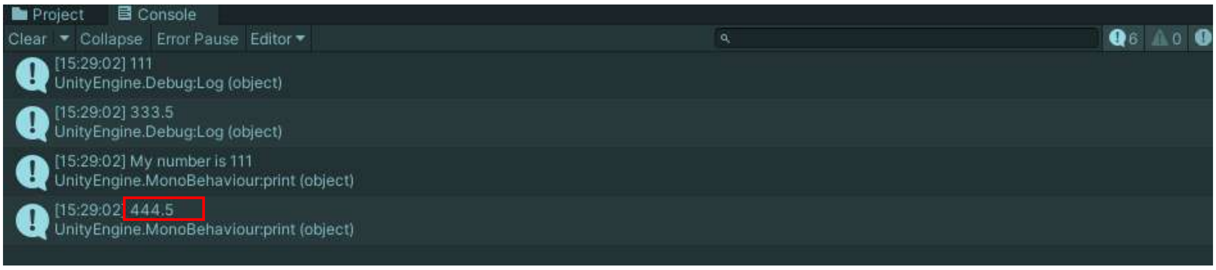




Let's give an example of arithmetic operations. Here, we define a variable in the **Start** method (function) and assign the **sum** of the values of the two previously defined variables.



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

public class ConsoleMessage : MonoBehaviour
{
    public int integer_number = 111;
    private float real_number = 333.5f;
    public string message = "My number is ";

    // Start is called before the first frame update
    void Start()
    {
        Debug.Log("UNITY");
        print("This is my first message");

        Debug.Log(integer_number);
        Debug.Log(real_number);

        print(message + integer_number);

        float summation = integer_number + real_number;

        print(summation);
    }

    // Update is called once per frame
    void Update()
    {

    }
}
```



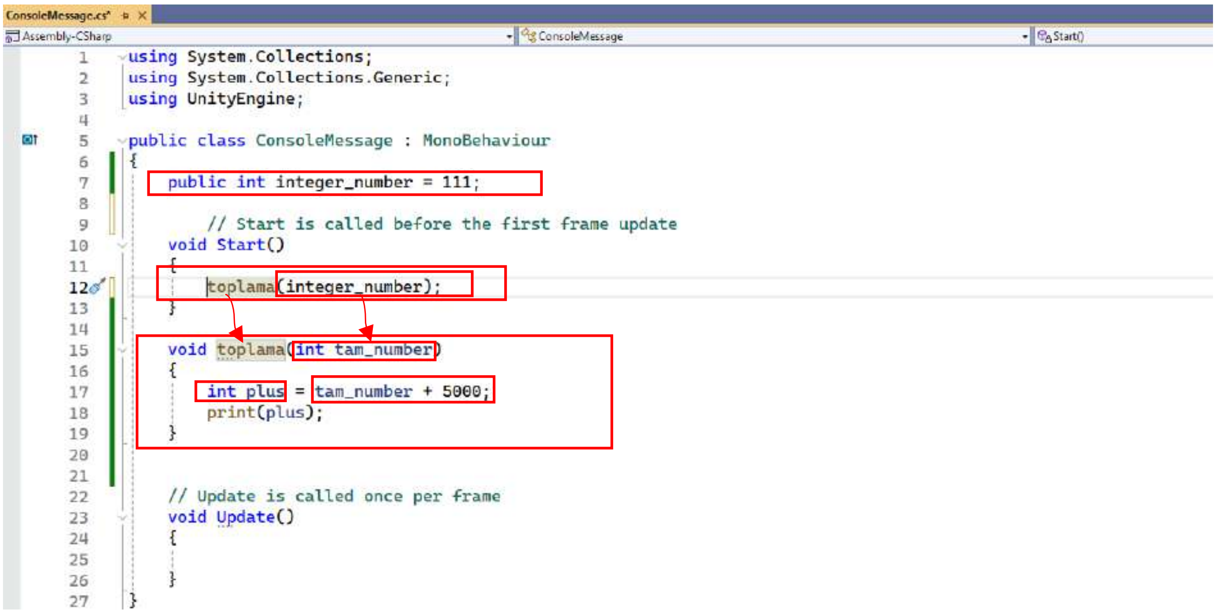
4.5.Methods/Functions

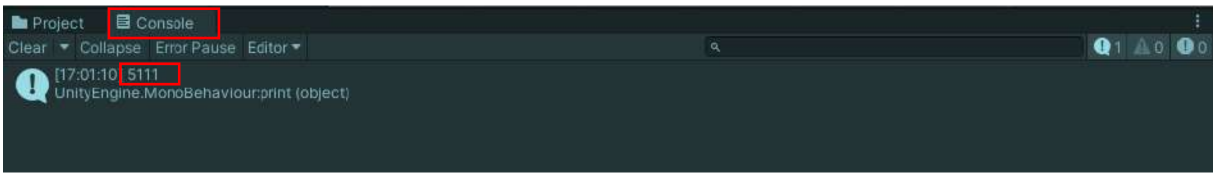
Apart from the **Start()** and **Update()** methods (functions), we can also develop our methods. In these function blocks, which have characteristics such as **type**, **name**, and parentheses, pre-definitions such as public, private, and **type definitions** such as **int, float, and void** can be made.

In order to call and use the created methods in the program flow, it is sufficient to write their names. A **void**-type method is not expected to return a value when it returns to the line it was called. It is not necessary to transfer data to these types of methods. However, in methods with a type such as **int** or

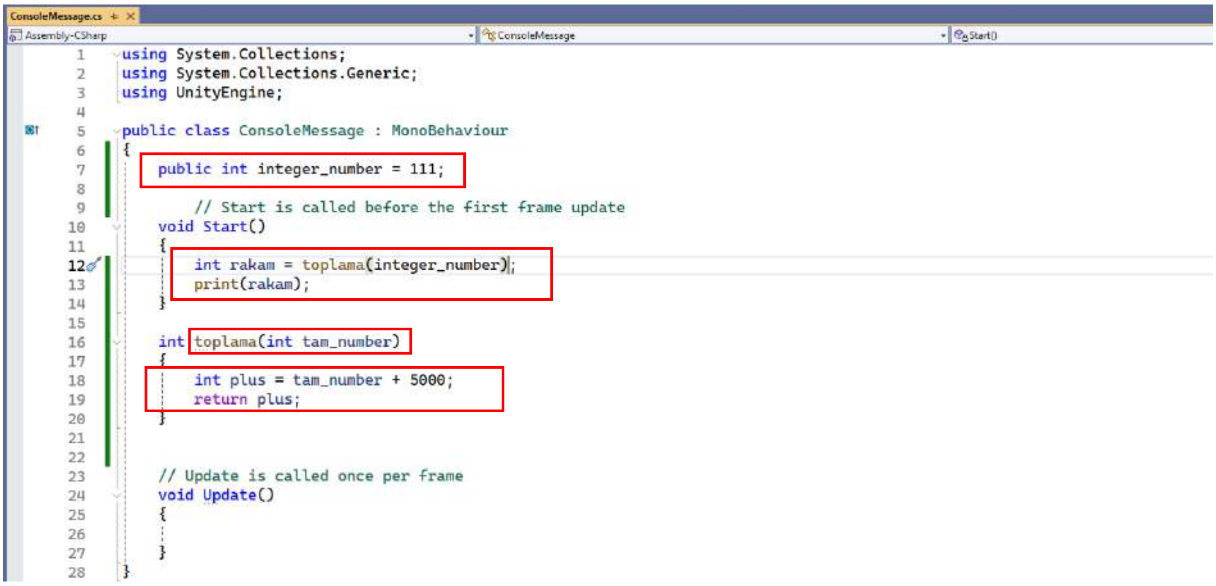
float, it is expected to produce a value of this type when it returns to the line it was called. In addition, there is a value transfer for the method to be processed.

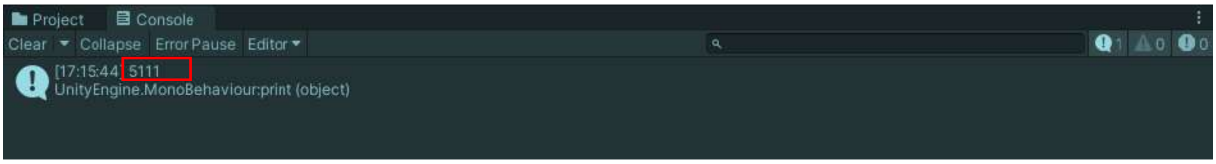
Now, remove some lines from the same project and focus on this issue. Leave only the **integer** variable in the variable definition. Create a **void add()** method inside the **Start()** method. *integer_number* variable value is transferred to this method. In the method definition line, a local variable called *int tam_number* is defined to hold this transferred value. *tam_number* value is directly equivalent to *integer_number*, i.e. 111. After defining another variable named *int plus* in the method, the *integer_value + 5000* operation is performed for the value assignment. The result, 5111, is printed on the **console**.





If the **add()** function had been int **add()** instead of **void**, the same result would have been obtained. However, the **int add()** function was expected to return a value with the **return** command.





We can achieve the same result by performing more variables and operations. The basis of object-oriented programming logic is to create modular program parts. Complex integrated and iterative software provides great savings from repeating the same operations over and over again.

4.6.Arithmetical Operators

Arithmetic operators are used for mathematical operations. These are basically four operations and some additional special operators:

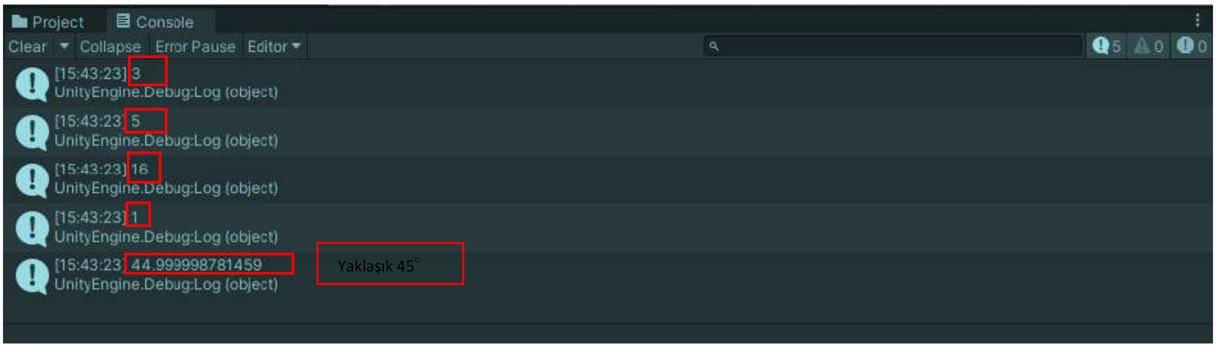
Operator	Function	Example C# code
+	addition	x=x+y; ≡ x+= y;
-	subtraction	w=w-q; ≡ w-=q;
*	multiplication	c=c*(a+b); ≡ c*= (a+b);
/	division (quotient)	d=d/5; ≡ d/=5;
++	add 1	x++; ≡ x=x+1; ≡ x+=1;
--	decrease 1	x--; ≡ x=x-1; ≡ x-=1;
%	mod operator (remainder)	z=10%3 (remainder 1)

Also, some commonly used C# Mathf class methods are listed:

Function	Math function	Meaning	Example C# code
Sqrt(x)	$\sqrt{x}$	Square Root	y=Mathf.Sqrt(x);
Abs(x)	$\lvert x \rvert$	Absolute Value	z=Mathf.Abs(x+y);
Pow(x,y)	x^y	Power	w=Mathf.Pow(x,y);
Tan(x)	tan(x)	Tangent	angle=Mathf.Tan(x*(Mathf.Pi/180)); // radian↔degree
Atan(x)	arctan(x)	Arc Tangent	val=Atan(angle*(180/Mathf.Pi)); // radian↔degree

Let's re-arrange the C# codes of our project in Visual Studio. Here, the square root of 9, the absolute value of -5, the 2nd power of 4 (square of 4), the tangent of 45, and the arc tangent of 1 are calculated. In the C# programming language, angles are in radians. If we want to work in degrees, we need to convert from radians to degrees and from degrees to radians.

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

public class ConsoleMessage : MonoBehaviour
{
    float number1 = 4f, number2 = 2f;
    float number3 = 45.0f;
    float number4 = 9f;
    float number5 = -5f;

      // Start is called before the first frame update
    void Start()
    {
        double x = Mathf.Sqrt(number4); Debug.Log(x);
        double y = Mathf.Abs(number5); Debug.Log(y);
        double p = Mathf.Pow(number1, number2); Debug.Log(p);
        double ang = Mathf.Tan(number3 * (Mathf.PI / 180)); Debug.Log(ang);
        double val=Mathf.Atan(1)*(180/Mathf.PI); Debug.Log(val);
    }

     // Update is called once per frame
    void Update()
    {

    }
}
```



*Yaklaşık: approximately

4.7. Conditional Statements and if statements

In some cases, operations may or may not be done in the program flow. We need to reflect these in the codes with conditional statements.

if(condition)

{…}

In the general template, we can write as follows: if the condition in the parentheses is met (**true**), the part inside the if block is executed. Otherwise (**false**), these lines will not be executed.

It is necessary to make comparisons in conditional sentences. Relational operators are used for this purpose:

>	greater	if(a>b)
>=	greater or equal	if(x>=y*z)
<	smaller	if(alfa<90)
<=	less or equal	if(u+w<=p/q)
==	equal	if(b==c*c)
!=	not equal	if(b*b-4*a*c != 0)

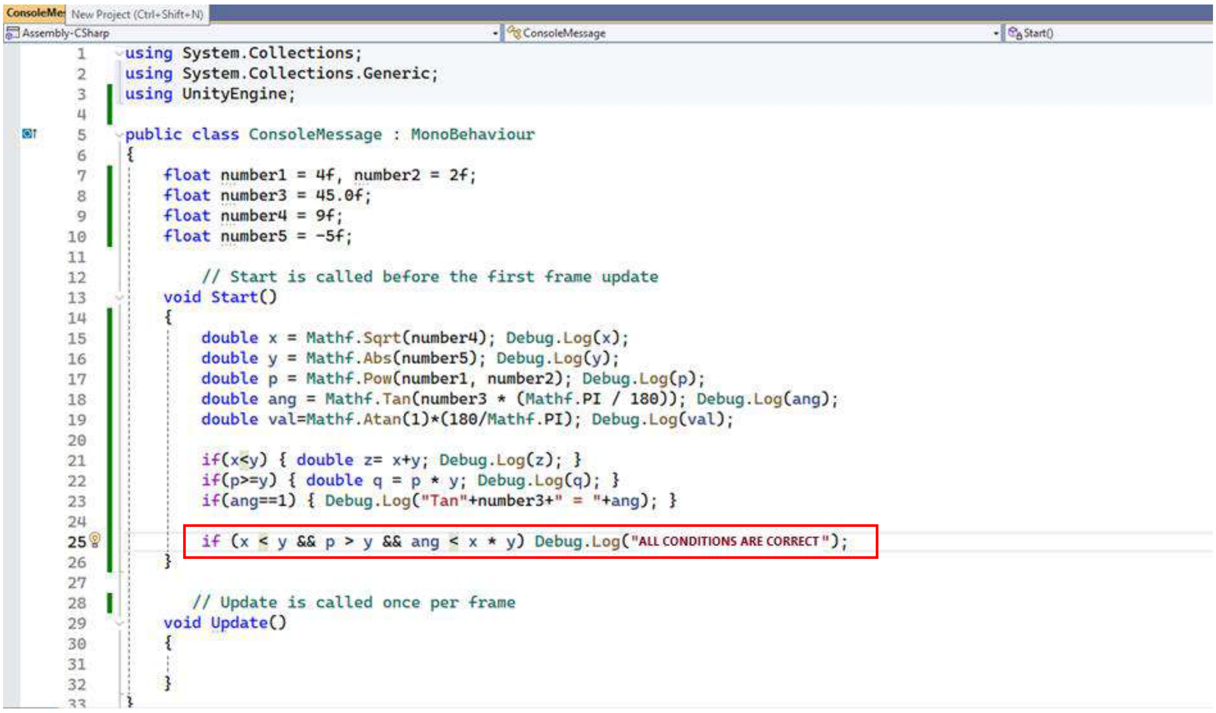
Now, add sample **if** statements to the previous program and see the results:

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

public class ConsoleMessage : MonoBehaviour
{
    float number1 = 4f, number2 = 2f;
    float number3 = 45.0f;
    float number4 = 9f;
    float number5 = -5f;

    // Start is called before the first frame update
    void Start()
    {
        double x = Mathf.Sqrt(number4); Debug.Log(x);
        double y = Mathf.Abs(number5); Debug.Log(y);
        double p = Mathf.Pow(number1, number2); Debug.Log(p);
        double ang = Mathf.Tan(number3 * (Mathf.PI / 180)); Debug.Log(ang);
        double val=Mathf.Atan(1)*(180/Mathf.PI); Debug.Log(val);

        if(x<y) { double z= x+y; Debug.Log(z); }
        if(p>=y) { double q = p * y; Debug.Log(q); }
        if(ang==1) { Debug.Log("Tan"+number3+" = "+ang); }
    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

In cases where there are multiple conditions, it is necessary to use logical operators to connect these conditions:

Operator	Meaning	Example C# code
&&	and	if (a+b >= c+d && a*a<(c/d)) { … }
||	or	if(b*b-4*a*c > 0 || a+b+c<d*e) { … }
!	not	if (!a) { … } // converted to true>false and false>true

Let's add an **if** statement to the project code that connects three conditions with **&&** and see that a message is sent to the console because all of them are true.



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

public class ConsoleMessage : MonoBehaviour
{
    float number1 = 4f, number2 = 2f;
    float number3 = 45.0f;
    float number4 = 9f;
    float number5 = -5f;

        // Start is called before the first frame update
    void Start()
    {
        double x = Mathf.Sqrt(number4); Debug.Log(x);
        double y = Mathf.Abs(number5); Debug.Log(y);
        double p = Mathf.Pow(number1, number2); Debug.Log(p);
        double ang = Mathf.Tan(number3 * (Mathf.PI / 180)); Debug.Log(ang);
        double val=Mathf.Atan(1)*(180/Mathf.PI); Debug.Log(val);

        if(x<y) { double z= x+y; Debug.Log(z); }
        if(p>=y) { double q = p * y; Debug.Log(q); }
        if(ang==1) { Debug.Log("Tan"+number3+" = "+ang); }

        if (x < y && p > y && ang < x * y) Debug.Log("ALL CONDITIONS ARE CORRECT");
    }

       // Update is called once per frame
    void Update()
    {

    }
}
```

For the expressions connected with **&&** to be true, all of them must be true. For only one of the conditions associated with **||** to be true, it is enough for the true result to come out of the parenthesis and for the operation (block) connected to the **if** statement to be executed.

The code information is provided below in text form that allows copy/paste.

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

public class ConsoleMessage : MonoBehaviour
{
   float number1 = 4f, number2 = 2f;
   float number3 = 45.0f;
   float number4 = 9f;
   float number5 = -5f;

      // Start is called before the first frame update
   void Start()
   {
      double x = Mathf.Sqrt(number4); Debug.Log(x);
      double y = Mathf.Abs(number5); Debug.Log(y);
      double p = Mathf.Pow(number1, number2); Debug.Log(p);
      double ang = Mathf.Tan(number3 * (Mathf.PI / 180)); Debug.Log(ang);
      double val=Mathf.Atan(1)*(180/Mathf.PI); Debug.Log(val);

      if(x<y) { double z= x+y; Debug.Log(z); }
      if(p>=y) { double q = p * y; Debug.Log(q); }
      if(ang==1) { Debug.Log("Tan"+number3+" = "+ang); }

      if (x < y && p > y && ang < x * y) Debug.Log("ALL CONDITIONS ARE TRUE");
   }

     // Update is called once per frame
   void Update()
   {

   }
}
```

4.8.Array

They are variables that can carry more than one data and have space made available for them in memory. Many similar data in accordance with the matrix structure are addressed and processed with index values within the group they belong to.

An **array** variable can be one or more dimensions. To allocate space in memory, there is a difference between square brackets **[ ]** in the definition sentence and a field size assignment with the new command.

int[] say;

say= new int[5];

In the example, instead of the variable named say having a single value, it is specified as **int say;** and it is an array as **int[] say;** and then five integers are allocated in memory. The values in an array structure connected to the same variable name are separated from each other according to their **addresses/index** numbers in the array. Although this index sequence number is from 1 to 5 in normal mathematical matrices, the index number starts from **0** in C#. Therefore, the sequence numbers will be between *0-4*.

The following example shows which sequence number element of the **say** variable the value assignment is made to.

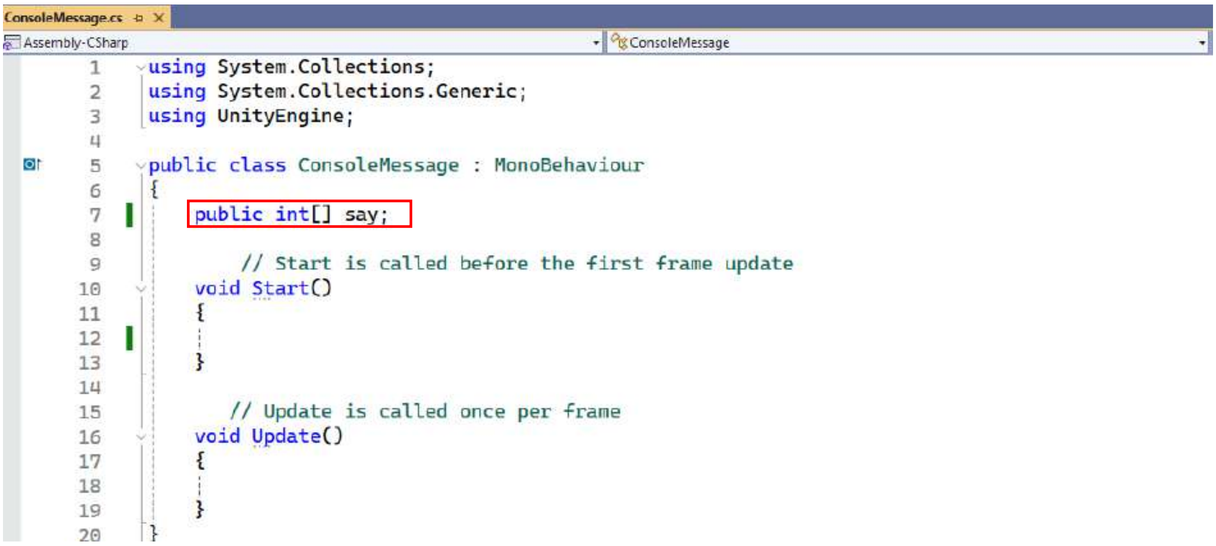
say[0]=3;

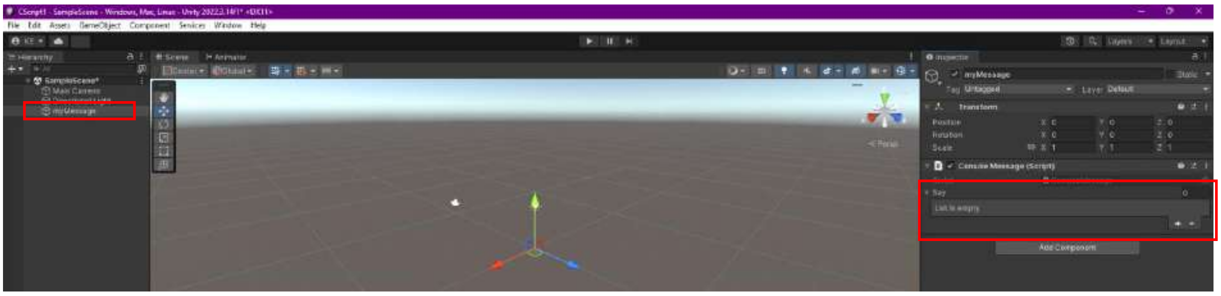
Here, the index value is 0. Its mathematical equivalent is the 1st row.

If we want to print to the console, it will be written in a similar way as

Debug.Log(say[0]);

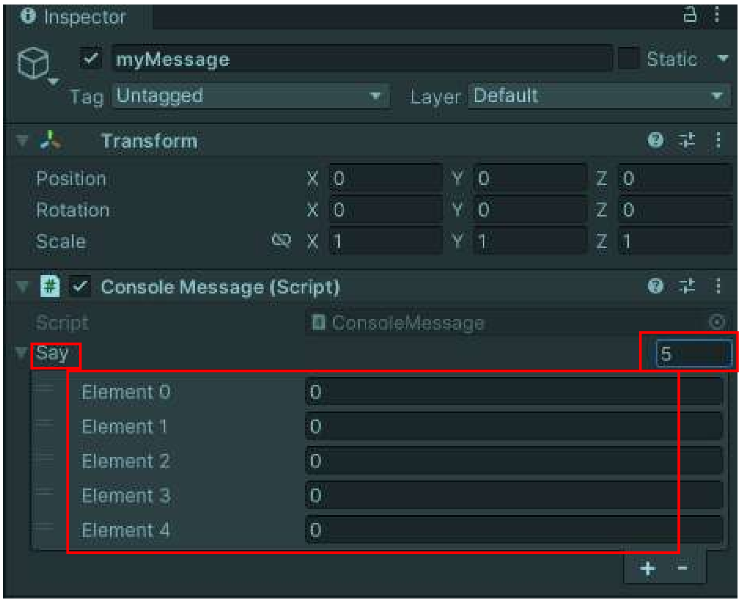
If the **public** property is given when defining the **say** variable, it can be accessed from the editor, size assignment and even element data entry can be done from here.





Here, both the number of elements of the **say** variable can be determined, and subsequently, data can be entered into the field that will be opened as many times as the number of elements determined.

For example, the number 5 was entered, and as a result, space was opened for 5 elements from the index value 0 to 4, and the initial value 0 was assigned to all of them.



4.9.List Variables

List variables have similar properties to array variables. However, it is not necessary to enter the matrix size (number of array elements) at the beginning. It is a structure that creates space for itself in memory as data is entered. The number of elements can increase and decrease. They are not fixed-sized like arrays but have a **dynamic** structure.

When making a definition, memory is created with the list declaration and **new** command.

List <int> rakam;

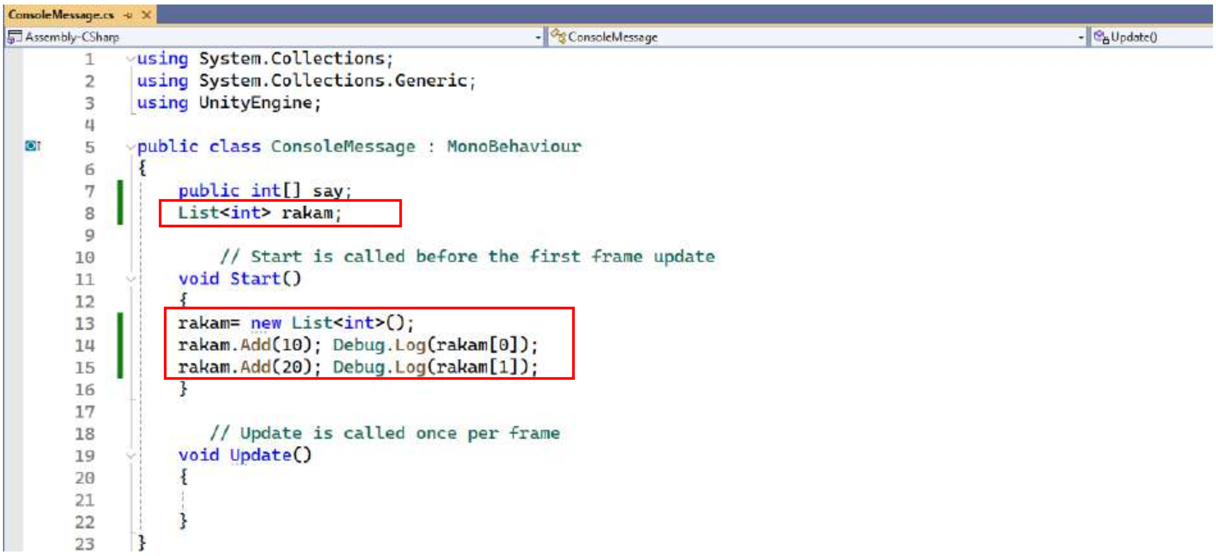
rakam= new List<>();

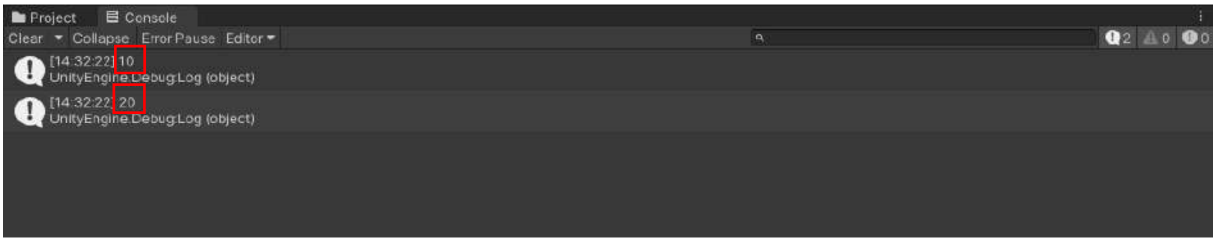
here, the variable type is declared.

rakam.Add(10);

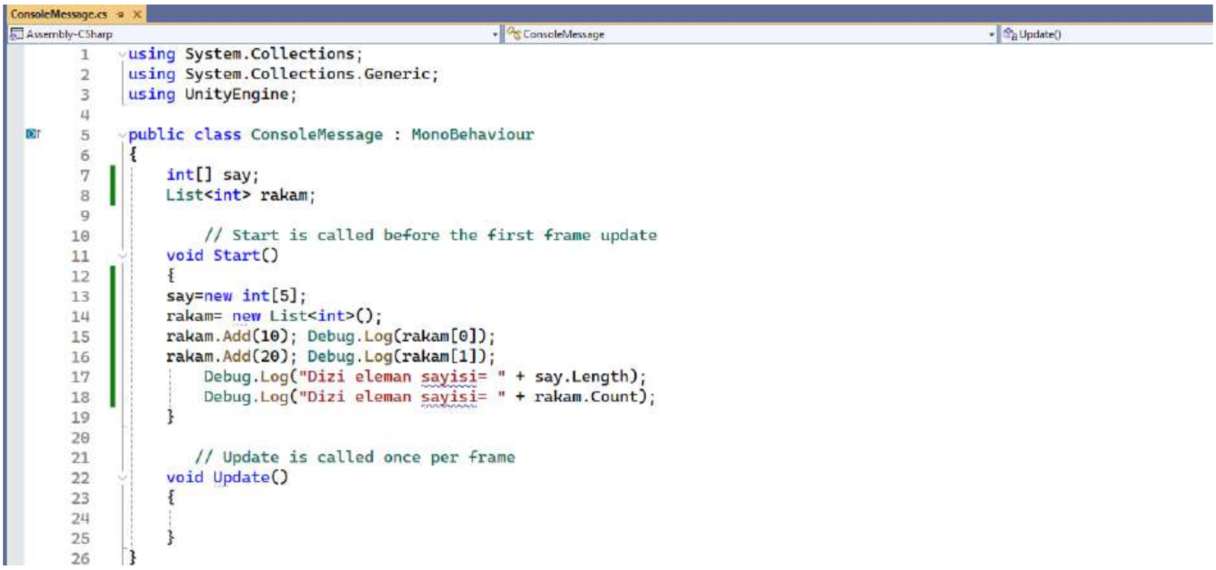
rakam.Add(20);

Add command is used to transfer data, and both spaces are made available in memory and data is assigned, respectively. The array structure and indexing method are the same. If we want to get output,





digit.Count and **count.Length** commands can be used to determine the number of elements in the list and array.



To remove an element from a list,

rakam.Remove(*assigned value*);

or

rakam.RemoveAt(*index number*);

can be used.

After the deletion process, the index order of the elements after the removed element is re-determined and the sequence number is updated.

Let's make an example where **for** structure we saw in the previous part of our topic loops between 0-4 5 times, and the **int i** variable controlling the loop will be assigned a value that is 100 times larger.

```
//using UnityEngine.Random;

public class ConsoleMessage : MonoBehaviour
{
    int[] say;
    List<int> rakam;


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

        say=new int[5];
        rakam= new List<int>();

        for (int i = 0; i < 5; i++)
        {
            rakam.Add(i*100);           // Value assignment
            Debug.Log(rakam[i]);        // Writing on the console
        }
    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

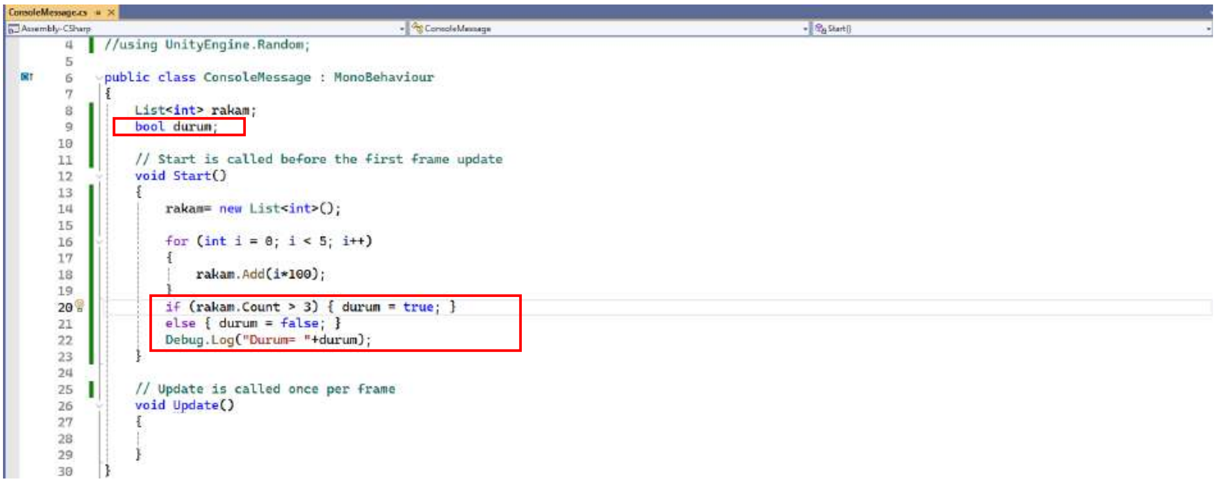
Another important **loop** structure that can be used with arrays and lists is **foreach**. A variable of the same type that can be used in our array or list is defined. The general template for our example is stated as follows:

foreach (int val in rakam)

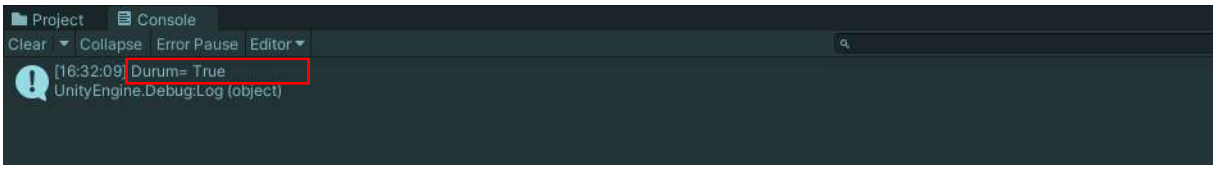
If we reflect it on our code lines,

```
//using UnityEngine.Random;

public class ConsoleMessage : MonoBehaviour
{
    List<int> rakam;

    // Start is called before the first frame update
    void Start()
    {
        rakam= new List<int>();

        for (int i = 0; i < 5; i++)
        {
            rakam.Add(i*100);
        }

        foreach(int val in rakam)
        {
            Debug.Log("rakam= "+val);
        }
    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

bool type variables can only take two values, **true** and **false**. Let's give an example with codes.

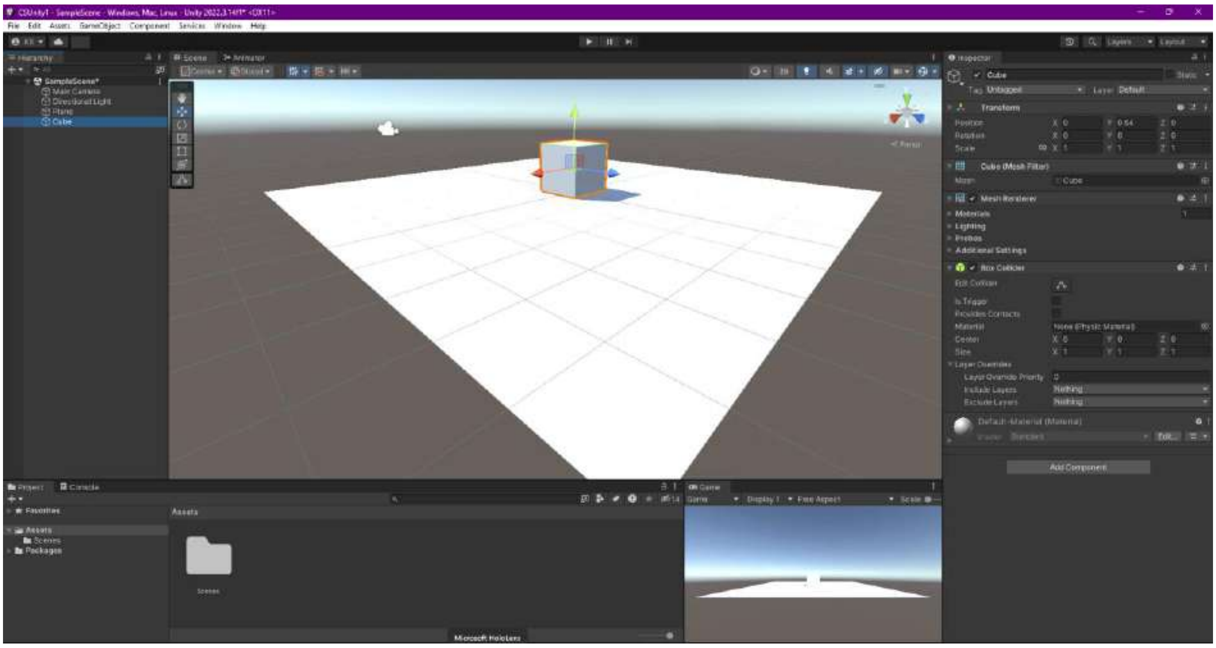


bool status variable is **true** if there are more than three numbers in the list. Otherwise, it is **false**.

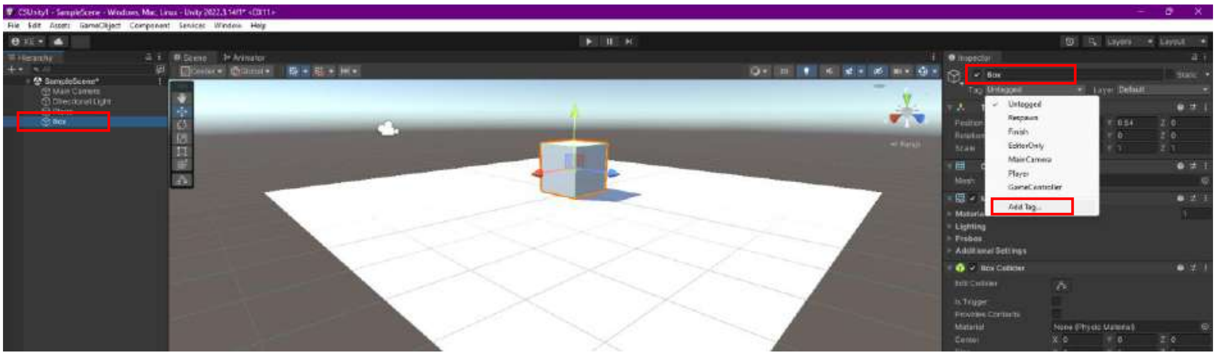


4.10.Basic C# Coding for Unity

In the following parts of our topic, let's see the basic and frequently used information about Unity objects and applications in **C#**. For this, let's start a new project. In the preparation phase, let's add a **Plane** and a **Cube** to our scene.

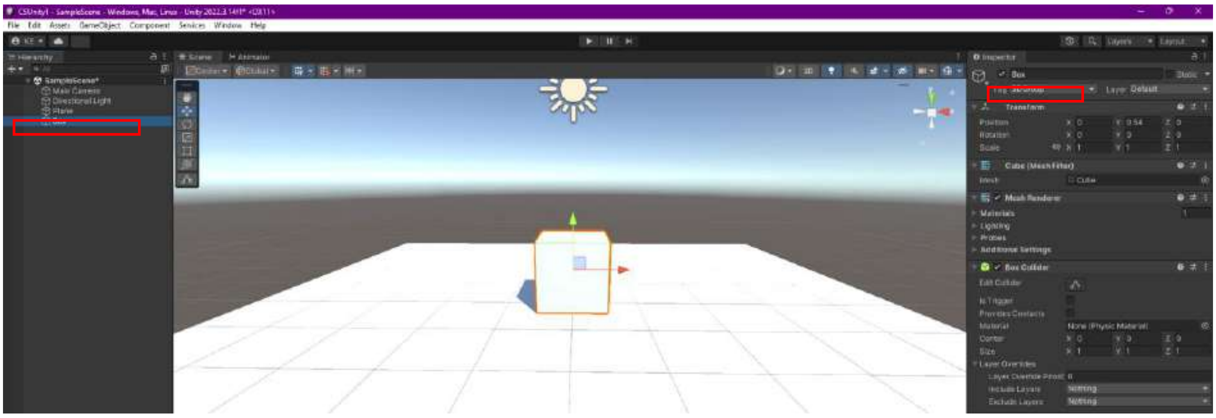


Let's change the name of the **Cube** object to **Box**.

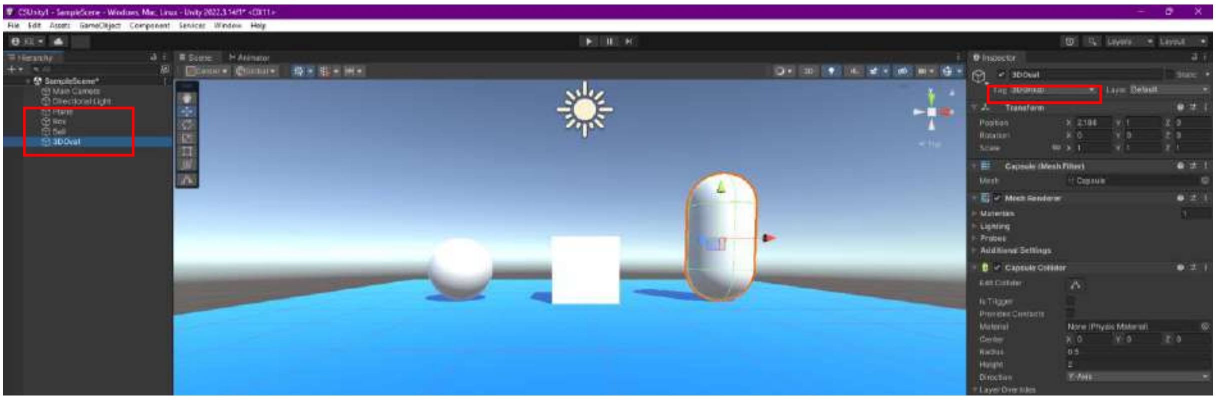


Let's create a new **Tag** named **3DGroup** for our **Box** object and assign it.

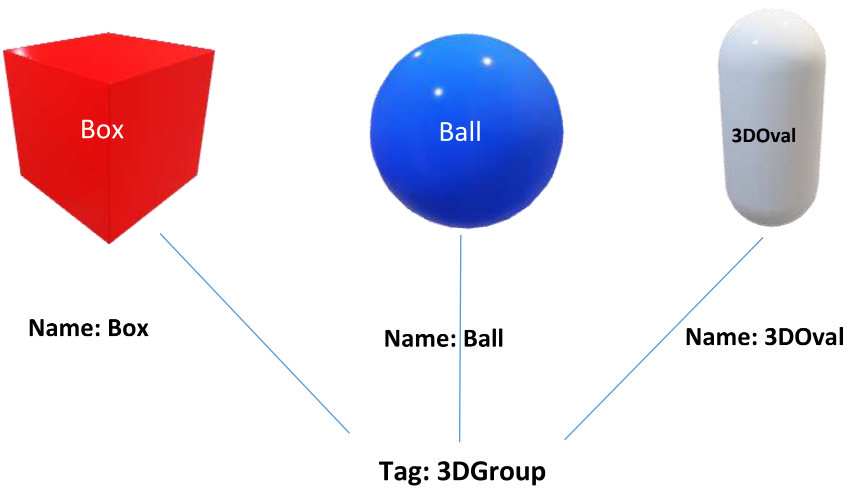
The principle of matrix indexing starting from **0** is also seen here in the listing of the **3DGroup** tag as **Tag 0**.



Similarly, let's add a **Sphere** and a **Capsule** to our scene. Let's name them **Ball** and **3DOval,** respectively, and let's name their **Tag 3DGroup**.



Let's see the **Name** and **Tag** information of our objects as a diagram. **Names** are seen as individual, and **Tag** information is seen as a common group name.



As you can see, the names of the three objects in our scene are different, and their labels are the same. **Name** and **Tag** are two different properties used to access the objects in the scene. **Name** can be used as an individual assignment for each object. However, let's remember that the same names can be given to different objects. **Tag** is generally labeled for grouping purposes.

In an application, for example, we may want to destroy each object based on its name, or we may want to destroy objects that are members of a group collectively. In such cases, we will need to access the **Name** or **Tag** information.

4.11.Object and Component Access

We need to access the object we want to process with code and the **Component** information in the **Inspector** section. Hierarchically, we need to access the object first and then the **Component** section.

For example, we want to access the **MeshRenderer** component (**Component**) of an object. Below is the access coding for two different situations.

In the **first case**, there is a **C#** file linked to the object. That is, the code file has already accessed the object to be accessed.

gameObject.GetComponent<MeshRenderer>(); or in shorter writing,

GetComponent<MeshRenderer>();

We can access the **MeshRenderer** property of the object with command.

In the **second case**, we want to access the **MeshRenderer** component (**Component**) of a different object in the scene that our code is not bound to. When the code file is not bound to this object, we will need a more detailed description. At this point, we can use the **Name** property of the object.

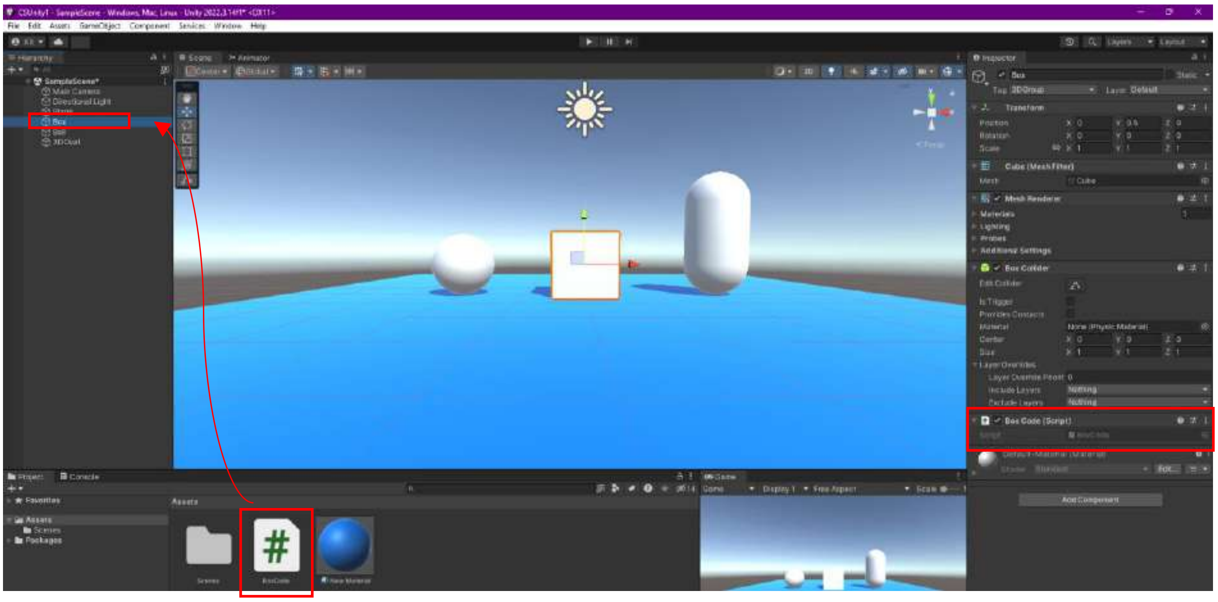
GetObject.Find("Box").GetComponent<MeshRenderer>();

If we want to access all **three** objects, we can write our code using the common group **Tag**.

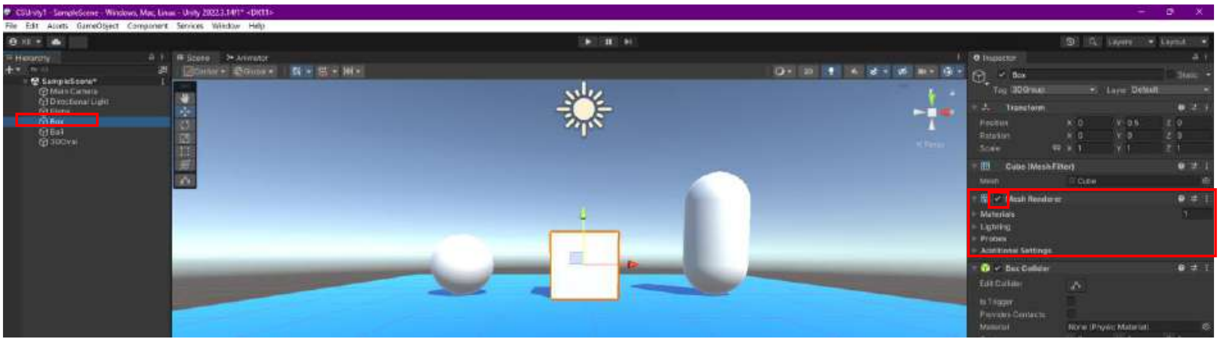
GetObject.FindWithTag("3DGroup").GetComponent<MeshRenderer>();

GetObject.FindGameObjectWithTag("3DGroup").GetComponent<MeshRenderer>();

Now, we can use this basic information in our Unity project. Let's create a **C#** Script to connect to our **Cube** object named **Box** and open it in Visual Studio (here, the file is named **BoxCode.cs**). Let's make it connected to our **Cube** (**Box**) object.



MeshRenderer Component contains settings related to the visibility of the object. The check box in the **Inspector>MeshRenderer** window of our object named **Box** shows whether this feature is active (**true**) or passive (**false**). Since it is currently active, the object is visible, and the box is **ticked**.



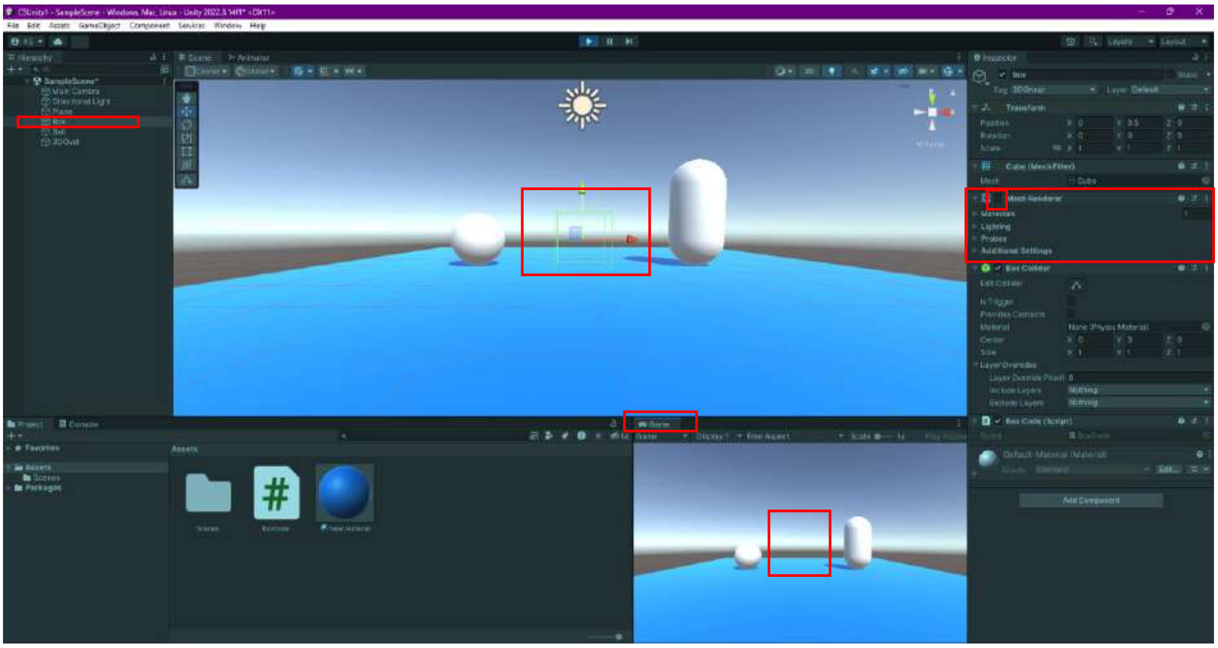
Let's make this object **invisible**. Unchecking the box means assigning the **bool** value called enabled for **MeshRenderer** to false as the code equivalent.

GetComponent<MeshRenderer>().enabled=false;

Let's write this line in Visual Studio, save it and run it in Unity **Play** Mode.

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

public class BoxCode : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        GetComponent<MeshRenderer>().enabled = false;
    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

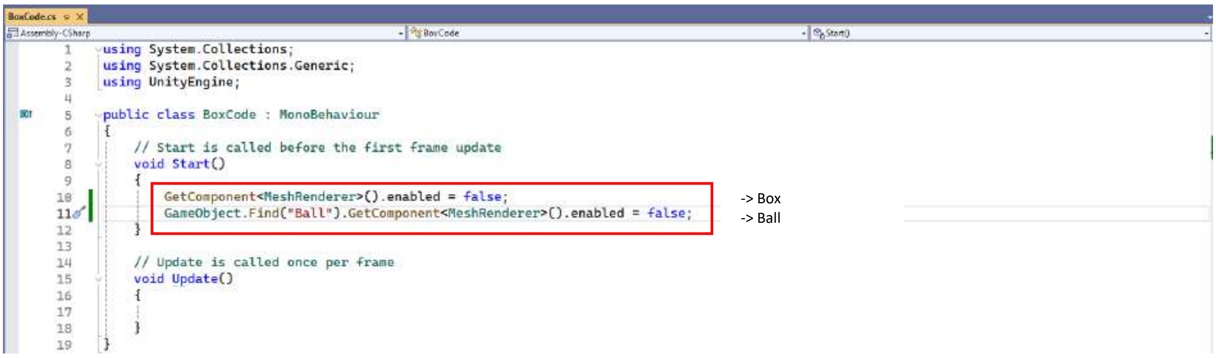
As can be seen, the **MeshRenderer** check box became **unchecked,** and the **Box** became invisible on the **Game** screen.

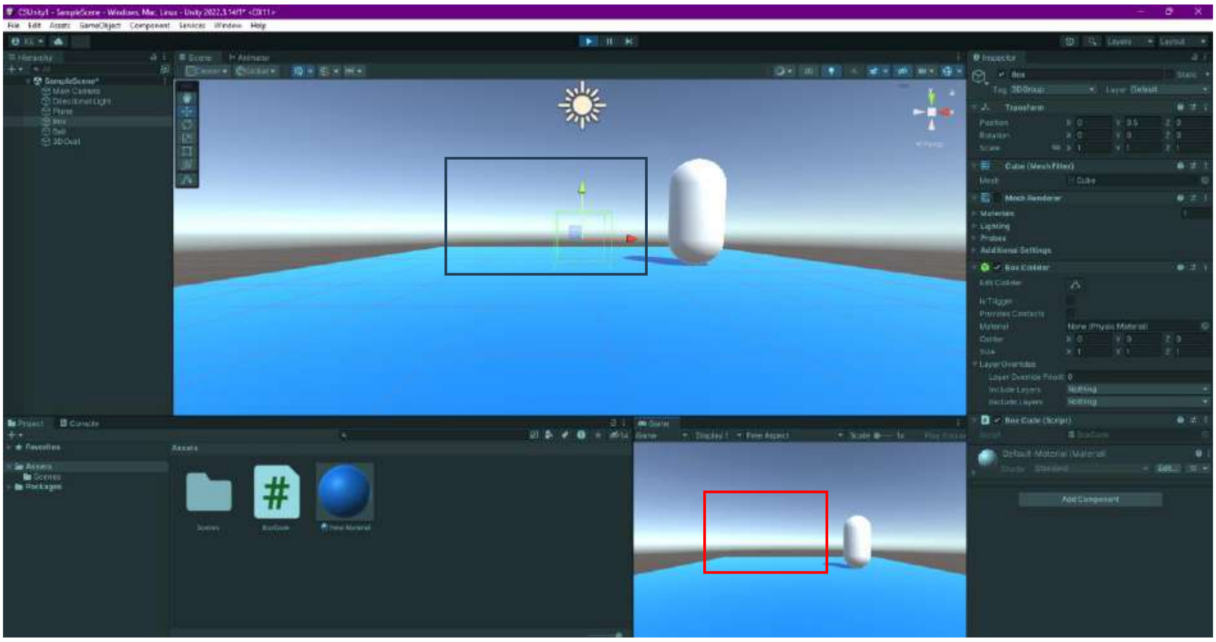


At this stage, let's access another object from our code file that it is not connected to and make it invisible. We can use the information we saw earlier to reach the **Sphere** object, which we named **Ball**.

GameObject. Find(“Ball”).GetComponent<MeshRenderer>().enabled=false;

Here, we first access the object named **Ball** with **GameObject.Find("Ball")** and the **MeshRenderer** active/passive property with **GetComponent<MeshRenderer>().enabled**. We set it equal to false to make it invisible.





If we want to make all objects with **3DGroup tags** invisible, we can use our array variable and **foreach** information since there is more than one object. Firstly,

GameObject[] objects;

Let's create an object matrix with a definition.

objects=GameObject.FindGameObjectWithTag("3DGroup");

Let's include all objects tagged with **3DGroup** in the array with the coding,

foreach(GameObject obj in objects)

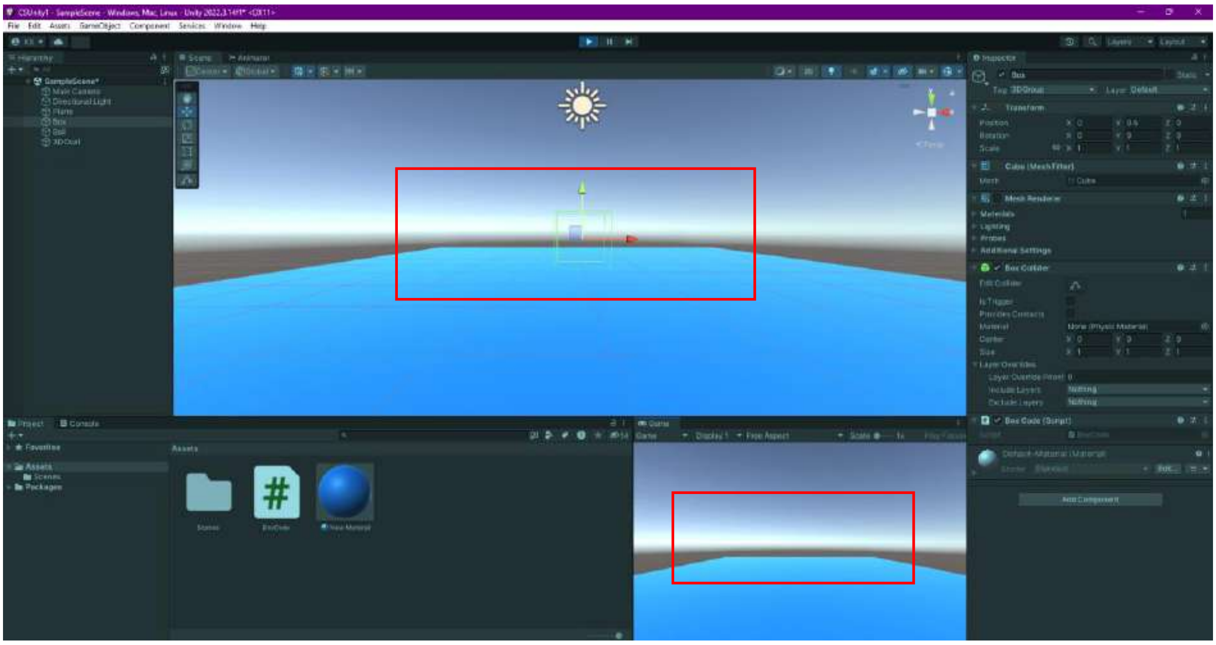
{ obj.GetComponent<MeshRenderer>().enabled=false;

}

Let's uncheck the **MeshRenderer** property of the objects in the matrix with the conversion. Now, let's turn it into a script in Visual Studio.

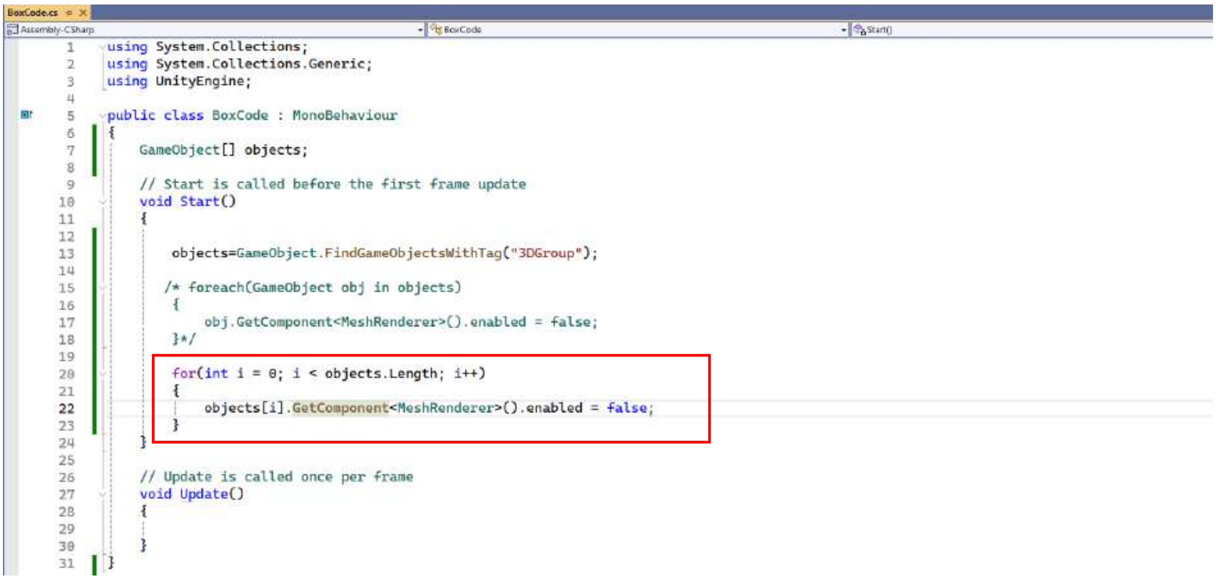
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BoxCode : MonoBehaviour
{
    GameObject[] objects;

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

        objects=GameObject.FindGameObjectsWithTag("3DGroup");

        foreach(GameObject obj in objects)
        {
            obj.GetComponent<MeshRenderer>().enabled = false;
        }
    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

We can see that since the **Tag** value of the other three objects in our scene, except **Plane**, is **3DGroup**, they all become invisible.



Let's do the same process with the **for**-loop pattern to reinforce our knowledge about array variables. For this, use the **foreach** block,

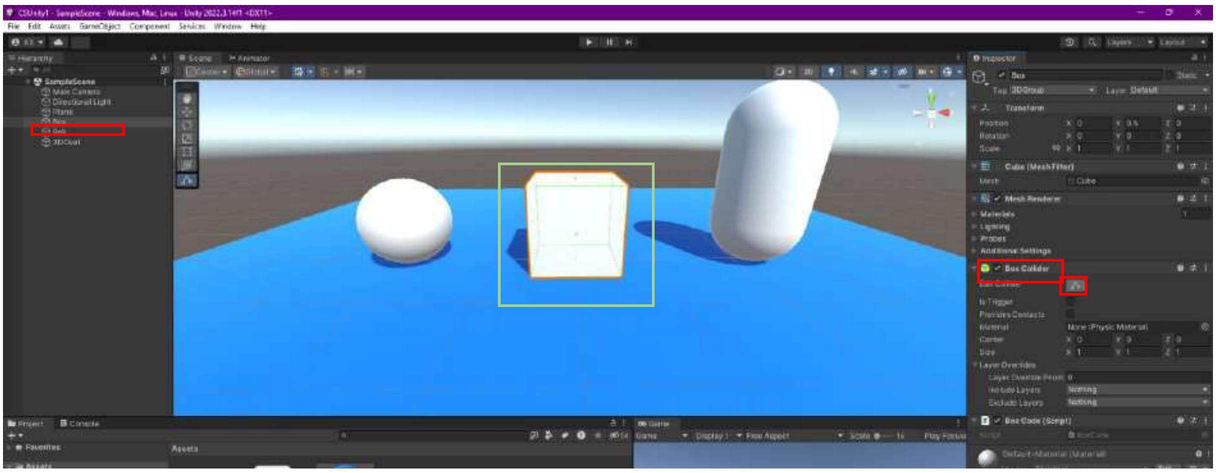
let's make a **comment** line by placing it between /* …. */ and disable it. Then, create a **for** loop block that will loop between 0 and **objects.Length**.



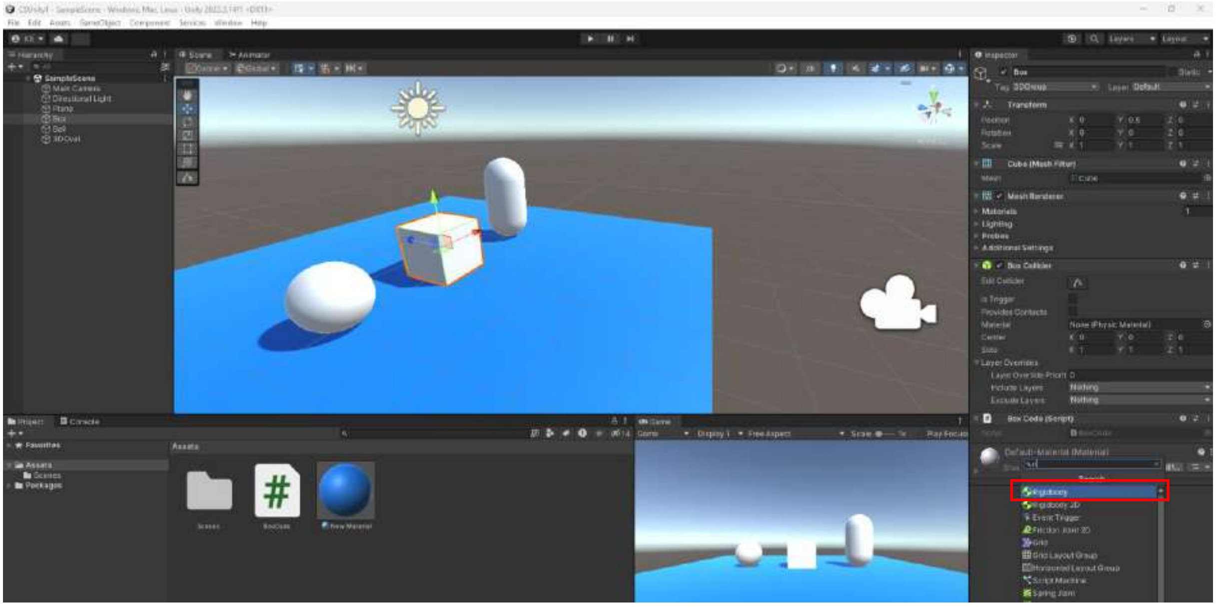
4.12.Collision/Interaction of Objects in Unity – On Collision

In this section, let's try to answer the question of how objects can collide or interact with each other in our applications using code.

First, the **Inspector** content of our objects that will interact should contain **Collider** and **Rigidbody** components. **Collider** means collider, and **Rigidbody** means a physical, rigid body that also has **gravity**. Normally, the **Collider** property of cube, sphere and similar objects that we add with **Create>3DObject** is assigned depending on the shape. **Box Collider** is also added while creating our object named **Box**. **Sphere Collider** is directly assigned to the object named **Ball,** and **Capsule Collider** is directly assigned to the object named **3DOval**. A **collider** can be thought of as a wireframe surrounding an object. If we want to change its shape, we can **edit** its borders, enlarge or reduce it with the button .



Let's add **Inspector>Add Component>RigidBody** to our **Box**, **Ball** and **3DOval** objects.



Collisions can be in the form of a hard collision (**Collider**) and a soft collision (**Trigger**). For the type of interaction between a person entering the water and the water, the **Is Trigger** □ box must be selected.

Collision control functions/methods, included in the C# Unity Engine library.

OnCollisionEnter()

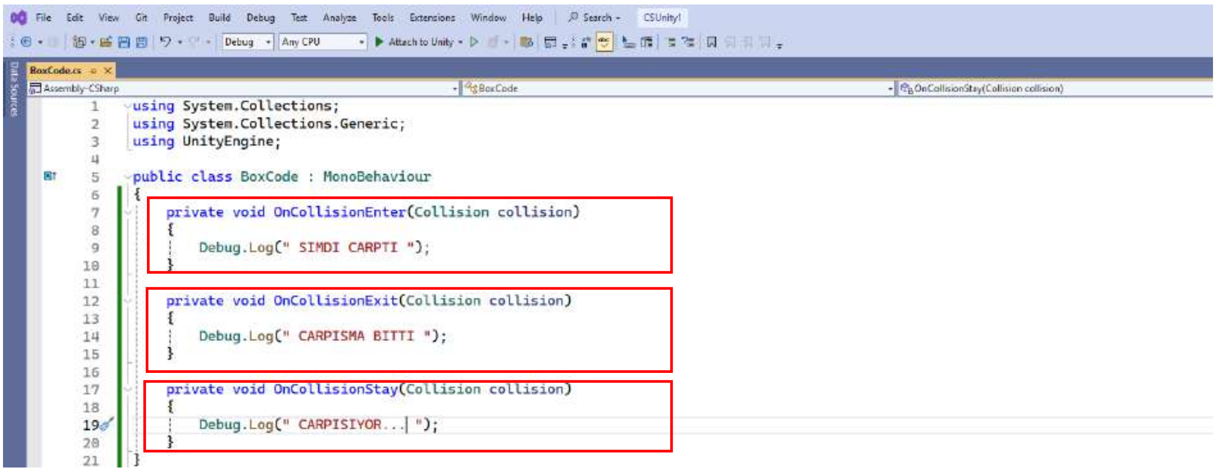
OnCollisionExit()

OnCollisionStay()

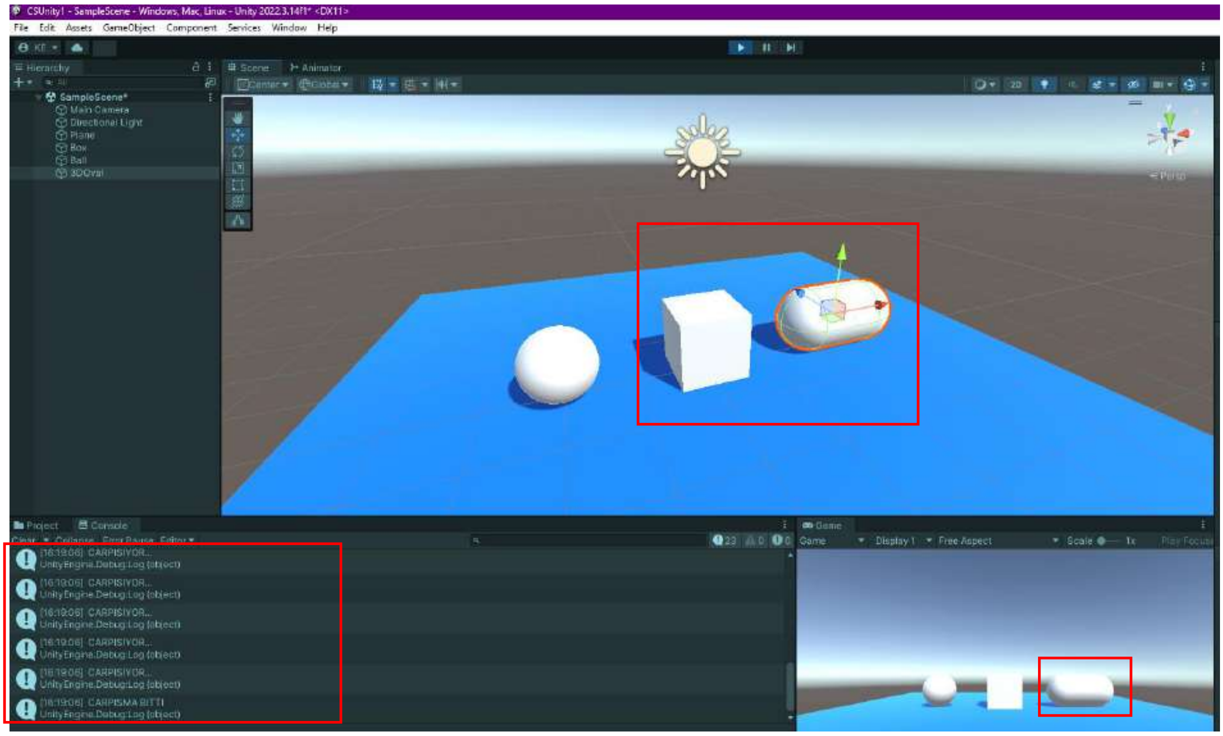
OnCollisionEnter() works once when first contacted. **OnCollisionExit()** works once when the collision is over. **OnCollisionStay()** runs continuously during the collision.

Let's start the application in our Project and write the collision script file. Since we don't need the **Start()** and **Update()** methods in these operations, we can delete them. Instead, let's write three methods that control the collision and let's write our message with the **Debug.Log** command to declare the stage where the collision occurred. The system assigns a collision variable in the parentheses of the **OnCollisionXYZ()** methods. The information about the object that was hit is stored in this parameter. It is possible to change the name of the variable if we want.

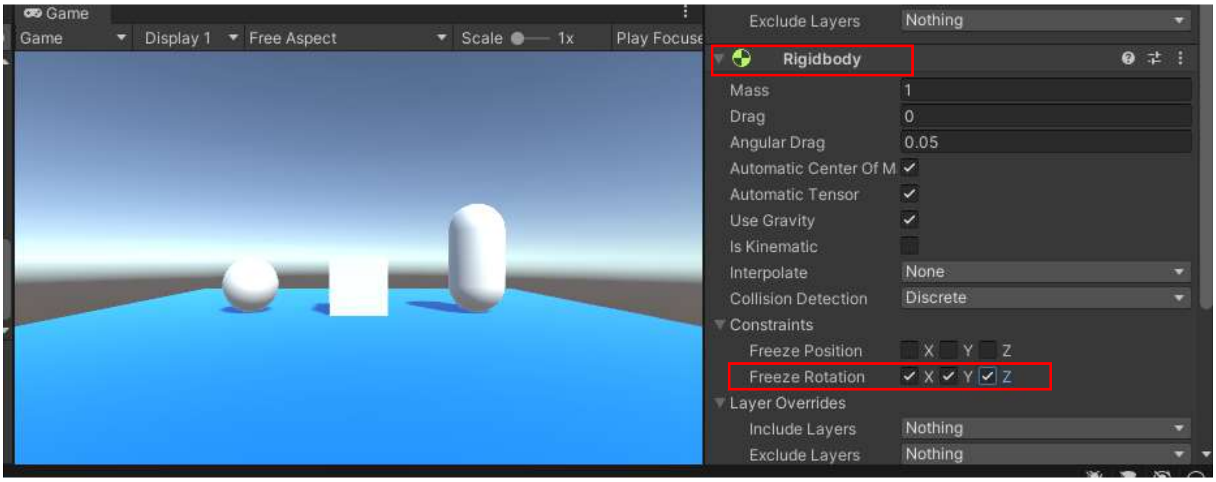
Translation: *SIMDI CARPTI (Now collided)* *CARPISMA BITTI (Collision ended)* *CARPISIYOR… (Now colliding)*



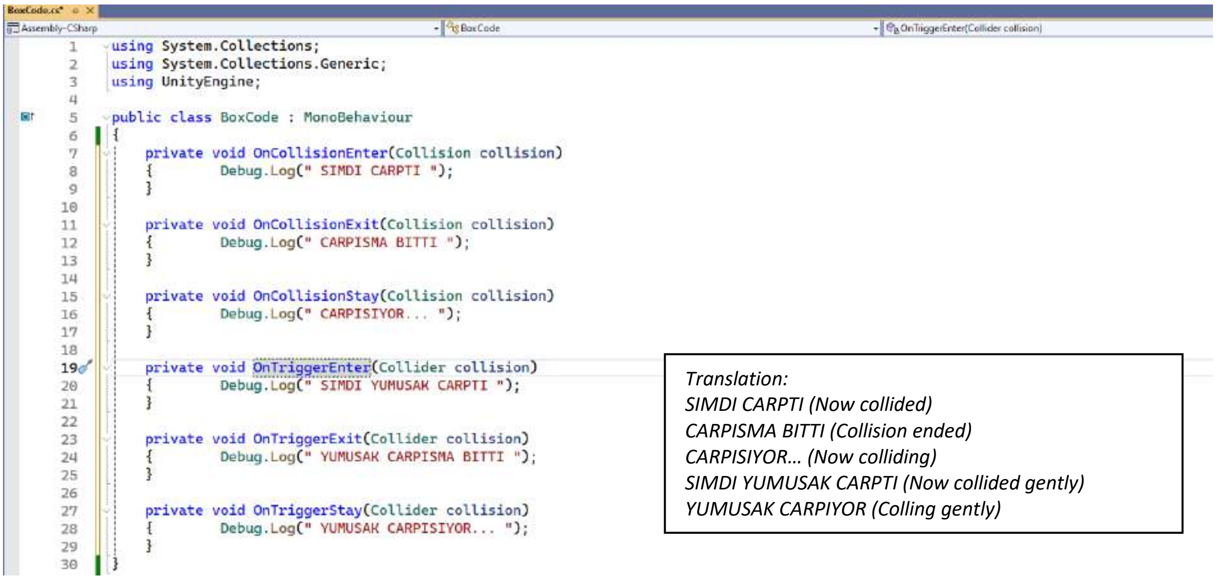
In **Play** mode, let's **drag** the capsule to our cube object, crash it and release it. The impact will knock over the capsule with physical properties. In the meantime, messages will appear in the **Console** window indicating that the collision has occurred, is ongoing and has finished.



If we want the capsule to tip over, click the **X, Y** and **Z** check boxes under **RigidBody>Constraints>Freeze Rotation**.



Let's also see the **soft collision**. For this, make a small change in the code lines. With the same logic, place the **OnTriggerEnter(), OnTriggerExit()** and **OnTriggerStay()** methods and the relevant **Debug.Log()** messages.

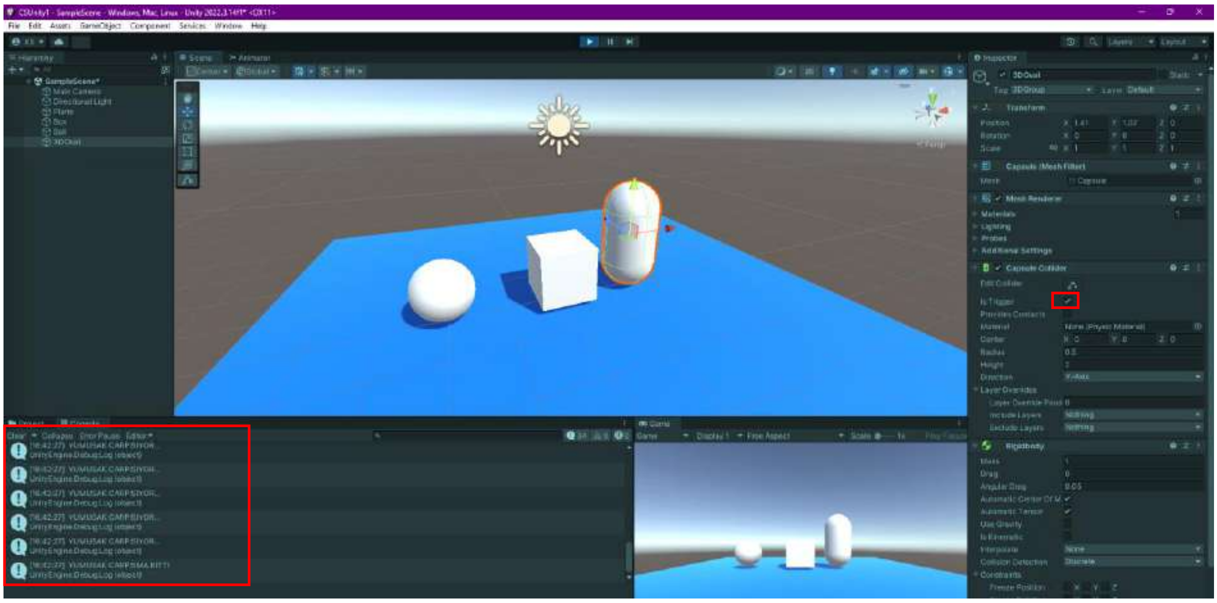


```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BoxCode : MonoBehaviour
{
    private void OnCollisionEnter(Collision collision)
    {
            Debug.Log(" SIMDI CARPTI ");
    }

    private void OnCollisionExit(Collision collision)
    {
            Debug.Log(" CARPISMA BITTI ");
    }

    private void OnCollisionStay(Collision collision)
    {
            Debug.Log(" CARPISIYOR... ");
    }

    private void OnTriggerEnter(Collider collision)
    {
            Debug.Log(" SIMDI YUMUSAK CARPTI ");
    }

    private void OnTriggerExit(Collider collision)
    {
            Debug.Log(" YUMUSAK CARPISMA BITTI ");
    }

    private void OnTriggerStay(Collider collision)
    {
            Debug.Log(" YUMUSAK CARPISIYOR... ");
    }
}
```

Translation:
SIMDI CARPTI (Now collided)
CARPISMA BITTI (Collision ended)
CARPISIYOR… (Now colliding)
SIMDI YUMUSAK CARPTI (Now collided gently)
YUMUSAK CARPIYOR (Colling gently)

Notice that in the **OnTrigger** commands, the variable definition in parentheses is **Collider**.

When we run it in **Play** Mode like this, we will still get the hard collision messages. In order to see the result of the codes we added, we need to click the **Is Trigger** check box of at least one or two of the objects that will collide. After this is done, the capsule that collides with the cube will give the soft collision messages. It will also not fall over…

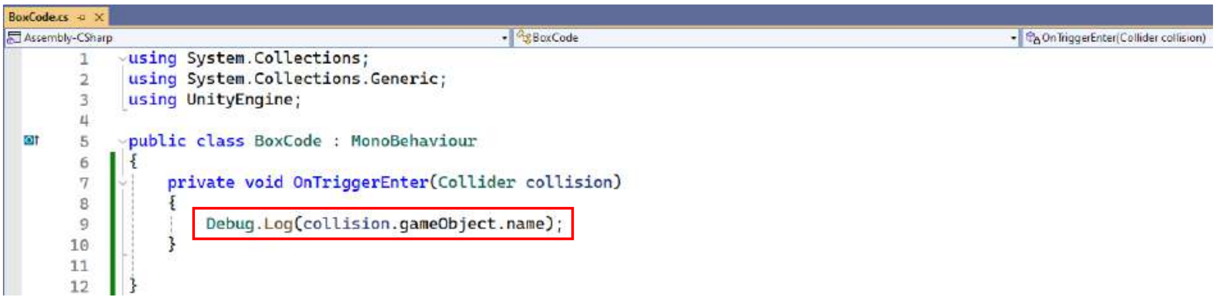


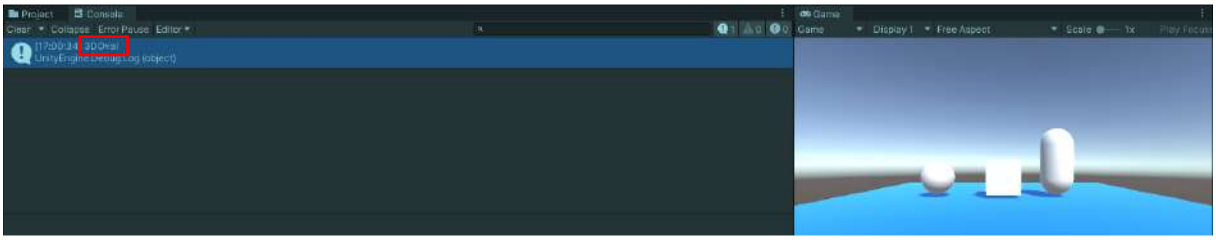
We have stated that the variable that is automatically opened in our method carries information about the collided object. To see it concretely, write the command

Debug.Log(collision.gameObject.name);

to the **OnTriggerEnter()** method.

Let's drag the cube into Unity again and collide it with the capsule. See the message in the **Console**. Do not forget that the code file is connected to the cube, and the **3DOval** information, which is the object it collides with, comes to the **Collider** collision variable in the method.

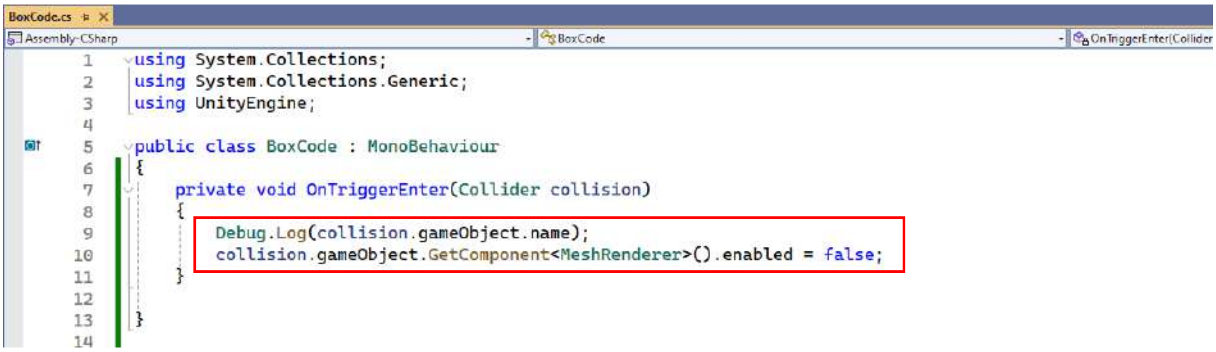


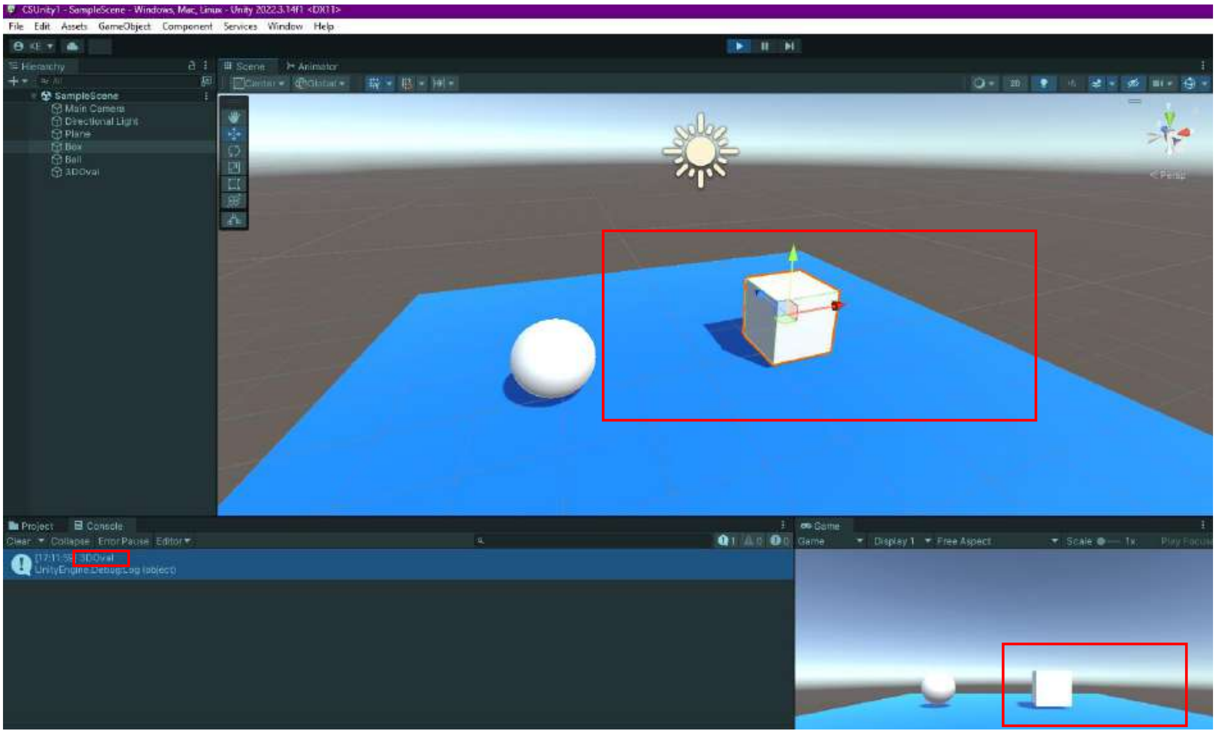


If we can reach the hit object, we can also perform operations on it; for example, we can make it invisible.

collision.gameObject.GetComponent<MeshRenderer>().enabled = false;

In this command, the **gameObject** to which the collision variable defined in parentheses is bound is made inactive and becomes invisible.



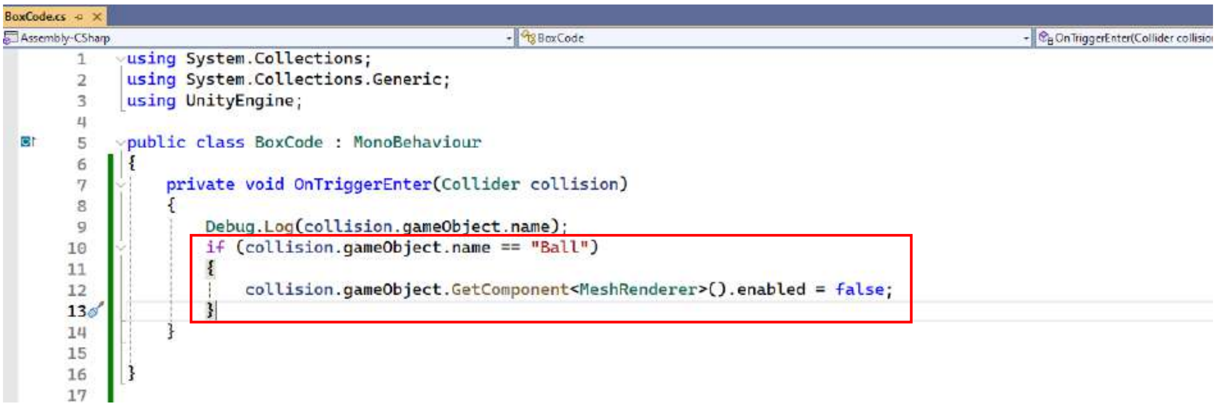


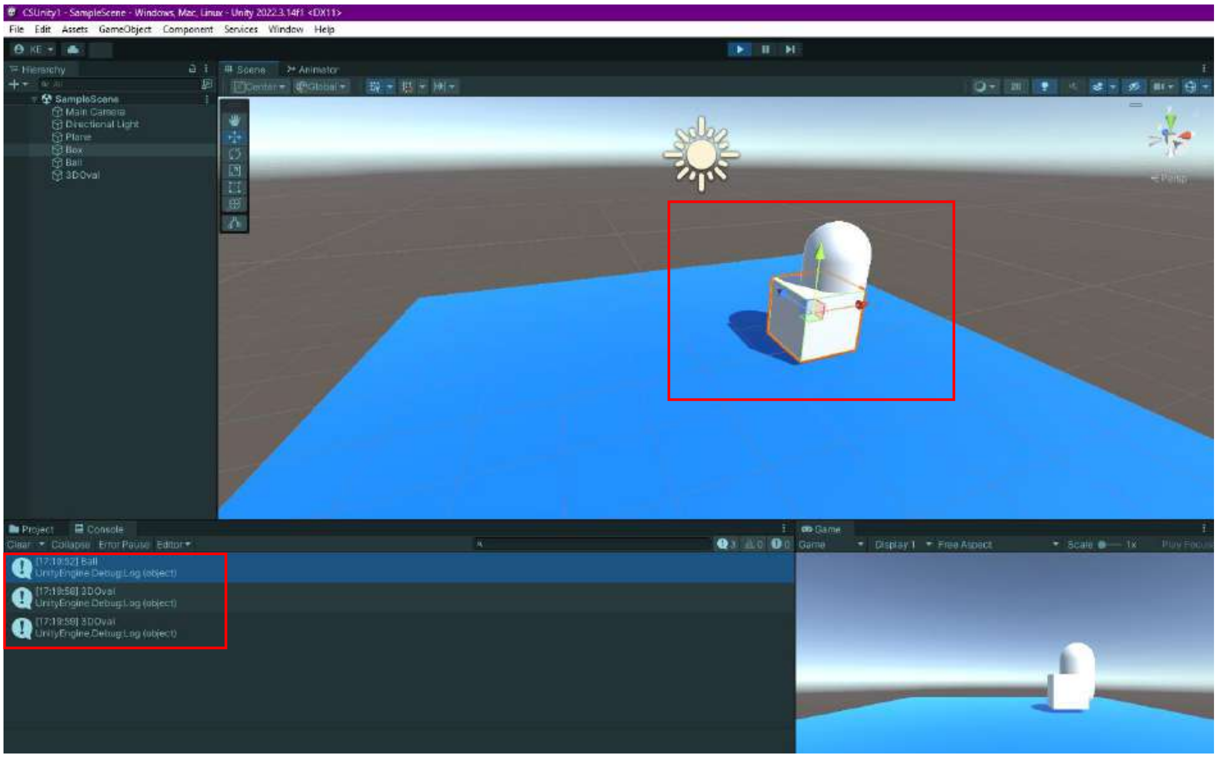
As a result, we understand through a message that it touches the capsule, and we can see that the **MeshRenderer** part of the capsule named **3DOval** is invisible because it is **unchecked**.

If we want to make an object with a certain name invisible, we need to make a conditional statement. Do this application on the **sphere** named **Ball**.

if(collision.gameObject.name=="Ball")
{
collision.gameObject.GetComponent<MeshRenderer>().enabled = false;
}

Now, transfer it to a **C# script** file.





Here, when we touched the cube to our **Ball** object, it both gave a message and destroyed it. However, when we touched the cube to the capsule a few times, it only showed a message but did not destroy it.

4.13.Adding Gestures with Mouse and Keyboard

In the applications, it is a basic need for the user to enter data with the mouse and keyboard.

Considering the dynamics of the movement in the application process, it is understood that it should be used in the **Update()** method.

It is common practice to detect whether a **keystroke** has been made on the mouse or keyboard in Unity and to control it with **if** statements to react to it.

Frequently used **methods** for **keyboard** buttons:

Input.GetKey (*KeyCode.button*);

Input.GetKeyDown(*KeyCode.button*);

Input.GetKeyUp(KeyCode.*button*);

for **Mouse** buttons:

Input.GetMouseButton(*#index*);

Input.GetMouseButtonDown(*#index*);

Input.GetMouseButtonUp(*#index*);

GetKey and **GetMouse**: works continuously
GetKeyDown and **GetMouseDown**: works once when pressed
GetKeyUp and **GetMouseUp**: works once when pressed

#if the index value is **0,** then **left** mouse button
#if the index value is **1** then **right** mouse button

We can write the first codes in our project.

For example, when the **right arrow key** on the keyboard is pressed, we can send a message to the console to test that it is detected. For this,

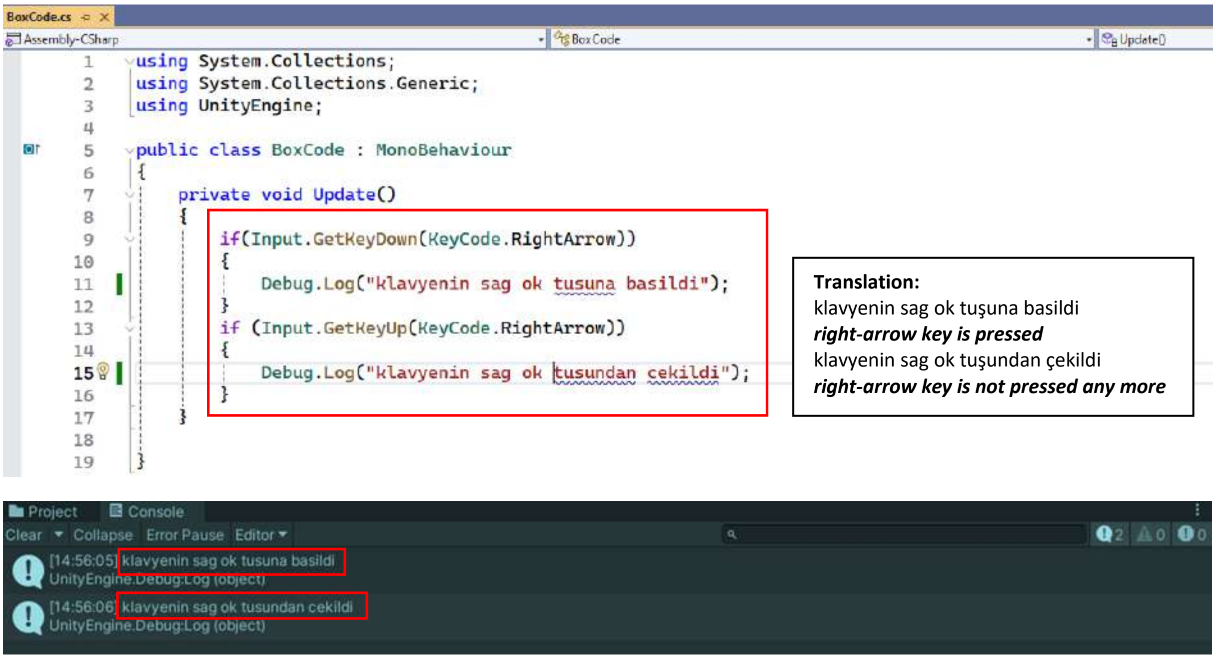
```
if(Input.GetKeyDown(KeyCode.RightArrow)
        {  Debug.Log(" klavyenin sag tusuna basildi ");  }
if(Input.GetKeyUp(KeyCode.RightArrow)
        {  Debug.Log(" klavyenin sag tuşundan cekildi ");  }
```

Translation:

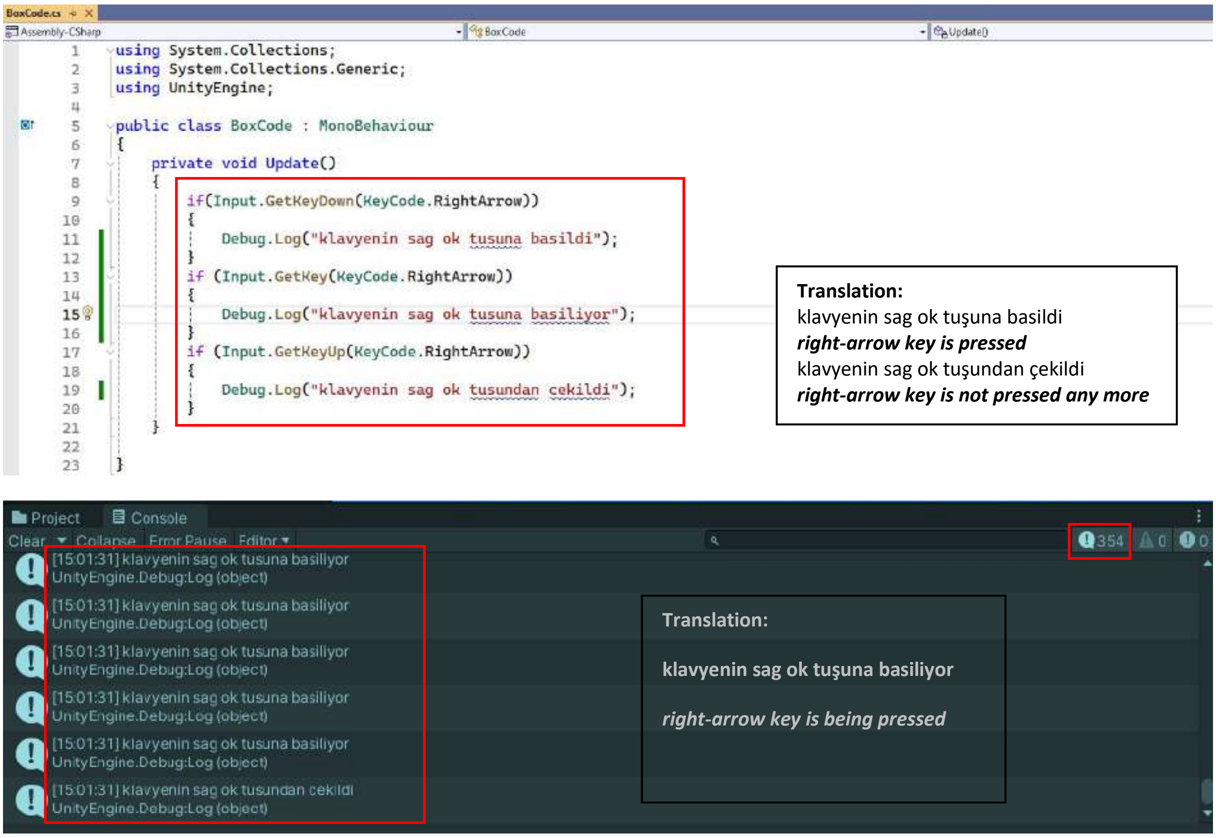
klavyenin sag ok tuşuna basildi

right-arrow key is pressed

Now, on Visual Studio:



If we add a third **if** statement as **if(Input.GetKey(KeyCode.RightArrow))**, it will give the message **Debug.Log("keyboard right arrow key is being pressed");** (in English) if we press the **right arrow key** (here, the screen message is given 352 times).

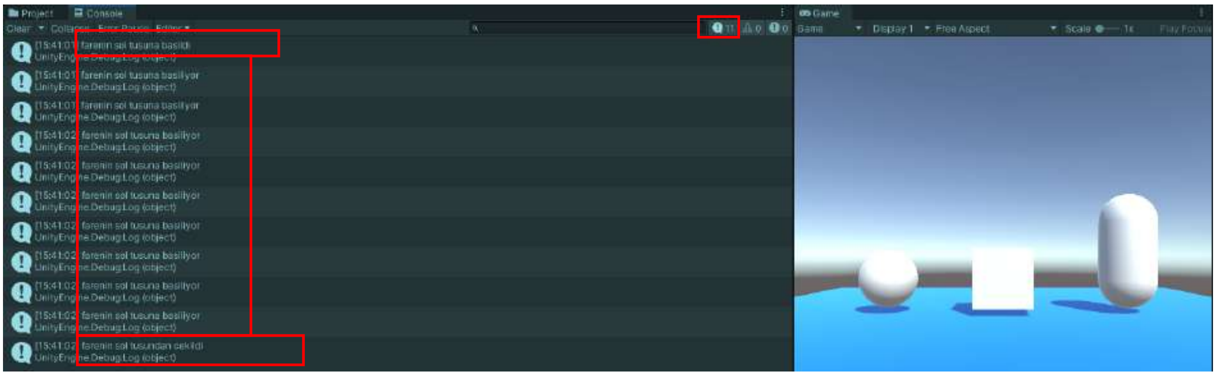


Let's do a similar process for the **left mouse button**. The left button **code** was expressed as **0**. Add to our codes to declare the moment the left button is pressed and released and during the pressing process.

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

public class BoxCode : MonoBehaviour
{
    private void Update()
    {
        if(Input.GetKeyDown(KeyCode.RightArrow))
        {  Debug.Log("klavyenin sag ok tusuna basildi");
        }
        if (Input.GetKey(KeyCode.RightArrow))
        {  Debug.Log("klavyenin sag ok tusuna basiliyor");
        }
        if (Input.GetKeyUp(KeyCode.RightArrow))
        {  Debug.Log("klavyenin sag ok tusundan cekildi");
        }

        if (Input.GetMouseButtonDown(0))
        {  Debug.Log("farenin sol tusuna basildi");
        }
        if (Input.GetMouseButtonUp(0))
        {  Debug.Log("farenin sol tusundan cekildi");
        }
        if (Input.GetMouseButton(0))
        {  Debug.Log("farenin sol tusuna basiliyor");
        }
    }

}
```

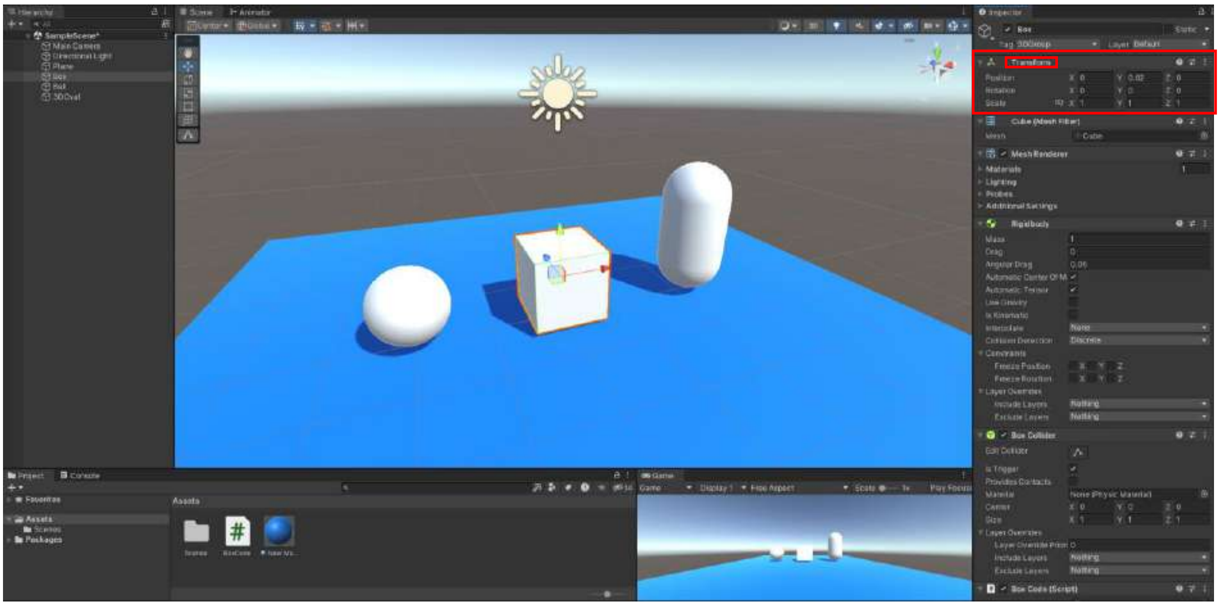
Translation:
farenin sol tuşuna basildi
left button of the mouse is pressed
farenin sol tuşundan çekildi
left button of the mouse is not pressed

Even a short press and release gives a total of 11 messages.



4.14.Controlling and Moving Objects

It is possible to access all **Component** properties in the **Inspector** section of an object in the scene with C# codes. **Inspector>Transform** properties contain the most basic parameters considered in components. These are **Position (x, y, x coordinates), Rotation (axial rotation angles in x, y, z directions)** and **Scale (scale values in x, y, z directions)**.



To make changes in this area, the **transform.Translate()** function will be used. In addition, by accessing the **Rigidbody** component, movement control can be provided with the **velocity()** and **AddForce()** commands.

Movement according to the object's coordinates,

transform.Translate(x,y,z);

or according to the coordinates of the scene,

transform.Translate(x,y,z,Space.World);

we can do it with command templates.

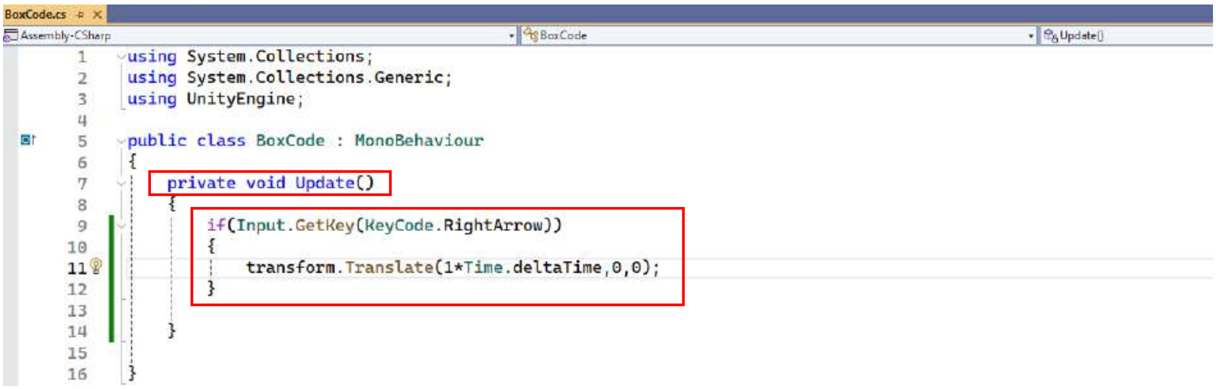
For example,

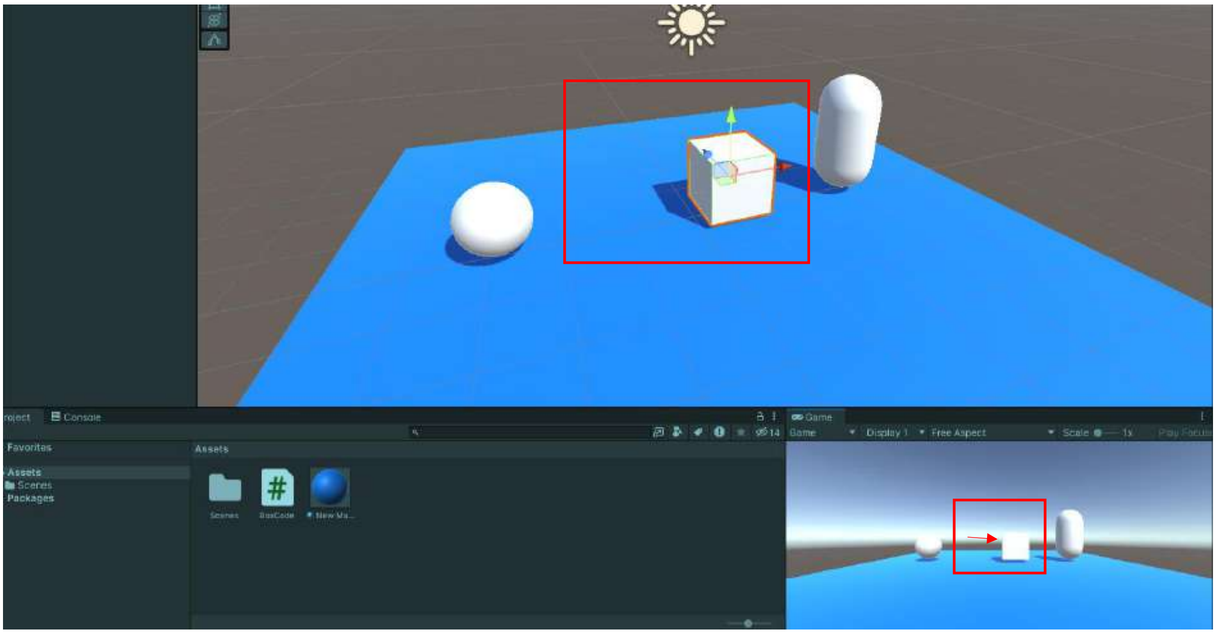
with the **transform.Translate(1,0,0);** command, **1** unit **forward** movement is provided on the **X** axis.

When we write this command in the **private void Update()** method, the movement occurs in the **X** direction if we **press** the key. However, the movement speed may differ depending on the configuration of the computers. To solve this problem and to ensure that the movement speed is equal on each computer, the axial movement unit is multiplied by the **Time.deltaTime** time value.

transform.Translate(1*Time.deltaTime,0,0);

Let's apply this information to the codes connected to the object named **Box** in our project. If we press the **right arrow key** on the **Play** Mode Game screen, we will see that the **Box** object moves **forward** on its own **X**-axis.





For an **upward** movement, it is possible to write as

transform.Translate(0,1*Time.deltaTime,0);

for a **downward** movement, it is possible to write as

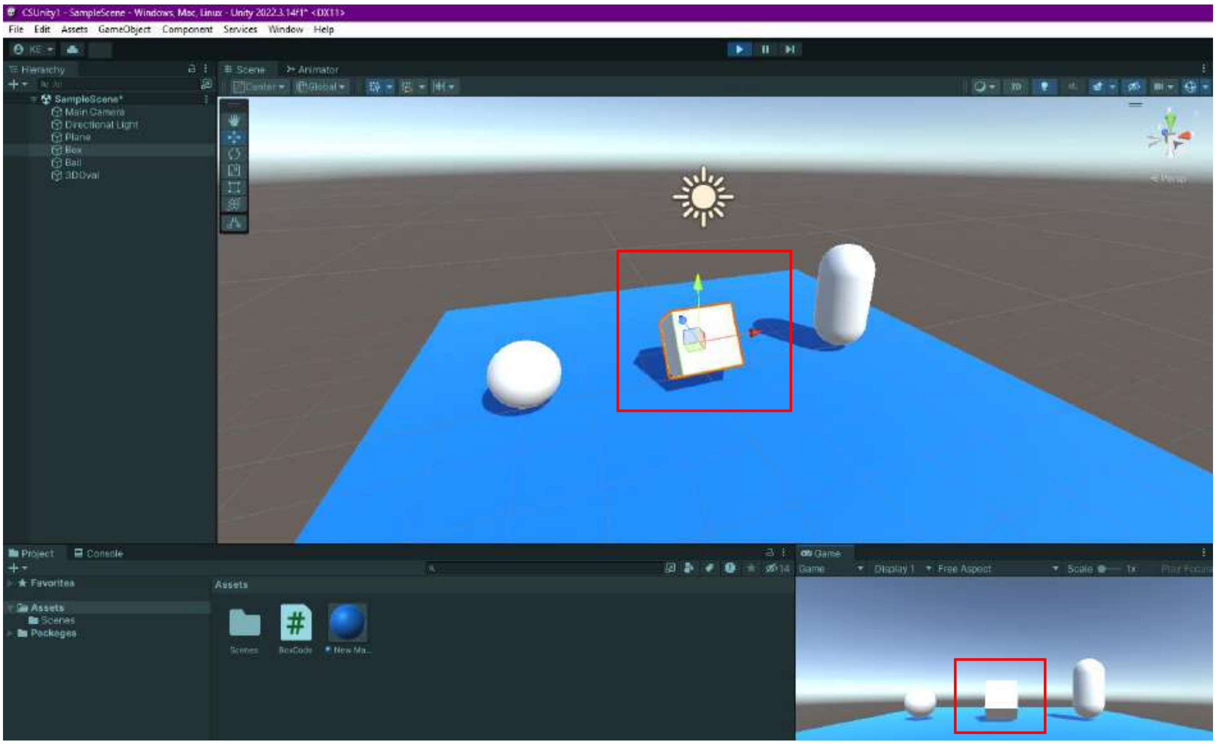
transform.Translate(0,-1*Time.deltaTime,0);

If we want to **rotate** the object, we must access the **Inspector>Transform>Rotation** parameter and use the relevant command. Similar to the **Position** control, we can use the **transform.Rotate(x,y,z);** and **transform.Rotate(x,y,z,Space.World);** template. For example, if we want it to **rotate** as long as we press the **right arrow key** on the X-axis, we can write

transform.Rotate(1*Time.deltaTime,0,0);

However, since **1*Time.deltaTime** will **rotate** very slowly, a value of **10*Time.deltaTime** or higher can be entered.

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

public class BoxCode : MonoBehaviour
{
    private void Update()
    {
        if(Input.GetKey(KeyCode.RightArrow))
        {
            transform.Rotate(10*Time.deltaTime,0,0);
        }

    }

}
```



For the **Scale** operation, it will be necessary to write a different code than the other command lines. We can use an expression of the **Vector3** variable type and enlarge our **Box** object in all directions as long as the **right arrow key** is pressed.

transform.localScale += Vector3.one*Time.deltaTime;

For **scaling** in certain directions, try the terms **forward, up, down, back, left,** and **right** instead of one in the expression **Vector3.one*Time.deltaTime**. We can set up the equation with the - sign to shrink.

transform.localScale -= Vector3.one*Time.deltaTime;

4.15.Interacting and Animating with Rigidbody

Rigidbody is a **physics component** element. Let's continue our topic on **Box**, which is the object that we connected to C# Script in the project. We added **Inspector>Add Component>Rigidbody** element to our **Cube** object named **Box**. We will need to access this element (**Component**) with codes and add **GetComponent<Rigidbody>()** and velocity function to animate it by giving it a speed value. It is possible to set its speed with a multiplier value and **Time.deltaTime**.

GetComponent<Rigidbody>().velocity=Vector3.right*50*Time.deltaTime;

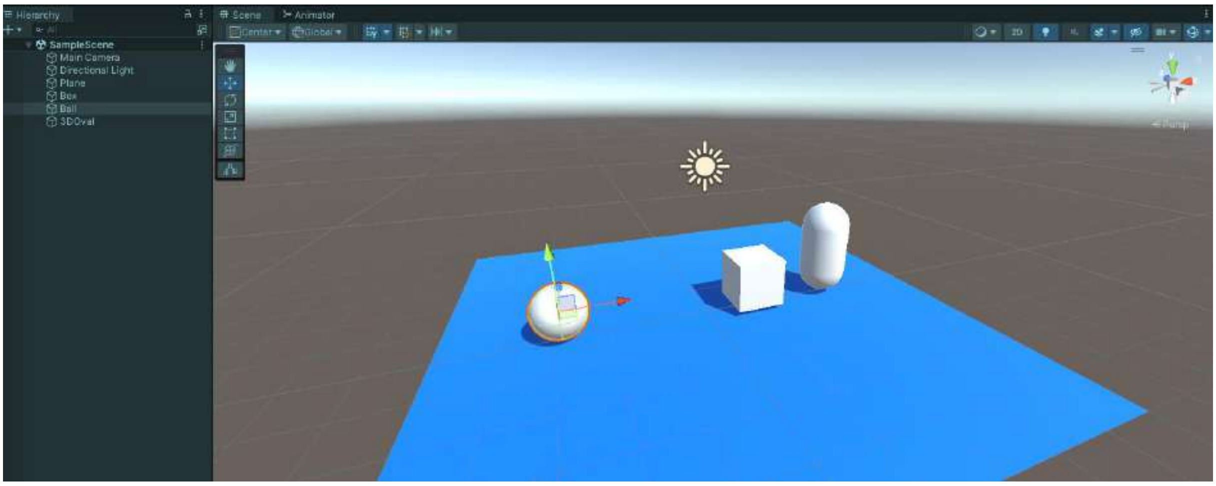
In order for this expression to take a **3D vector** value with the term it is equal to, in the scene,

Vector3.right // 1 unit right on the X axis
Vector3.left // 1 unit left on the X axis
Vector3.up // 1 unit up on the Y axis
Vector3.down // 1 unit down on the Y axis
Vector3.forward // 1 unit forward on the Z axis
Vector3.back // 1 unit back on the Z axis

on its **axis**;
transform.right // X axis 1 unit right
transform.left // X axis 1 unit left
transform.up // Y axis 1 unit up
transform.down // Y axis 1 unit down
transform.forward // Z axis 1 unit forward
transform.back // Z axis 1 unit back

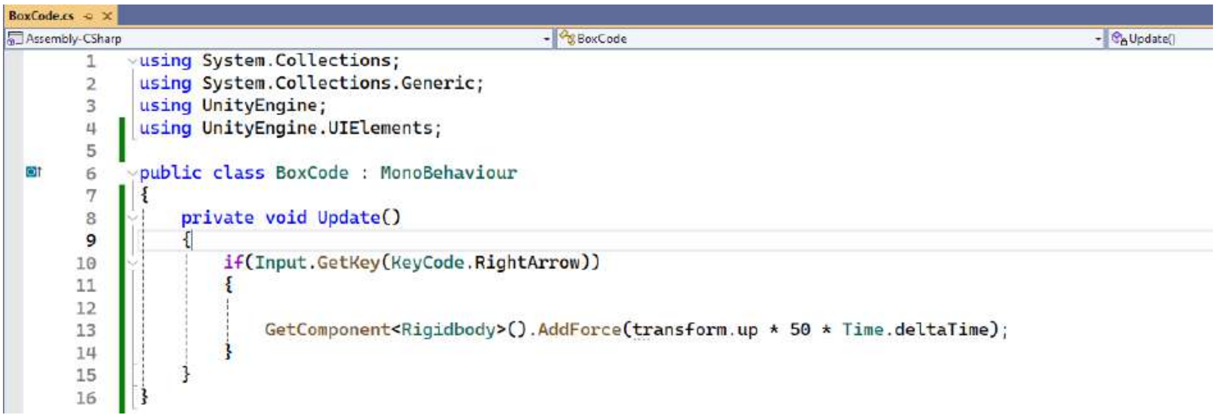
When we apply it to the script file, we can see the movement of the **Box** object when we press the right arrow key. Here, we see that once we start the movement, it continues at a constant speed.

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

public class BoxCode : MonoBehaviour
{
    private void Update()
    {


        if(Input.GetKey(KeyCode.RightArrow))
        {

            GetComponent<Rigidbody>().velocity = Vector3.right * 50 * Time.deltaTime;
        }

    }

}
```

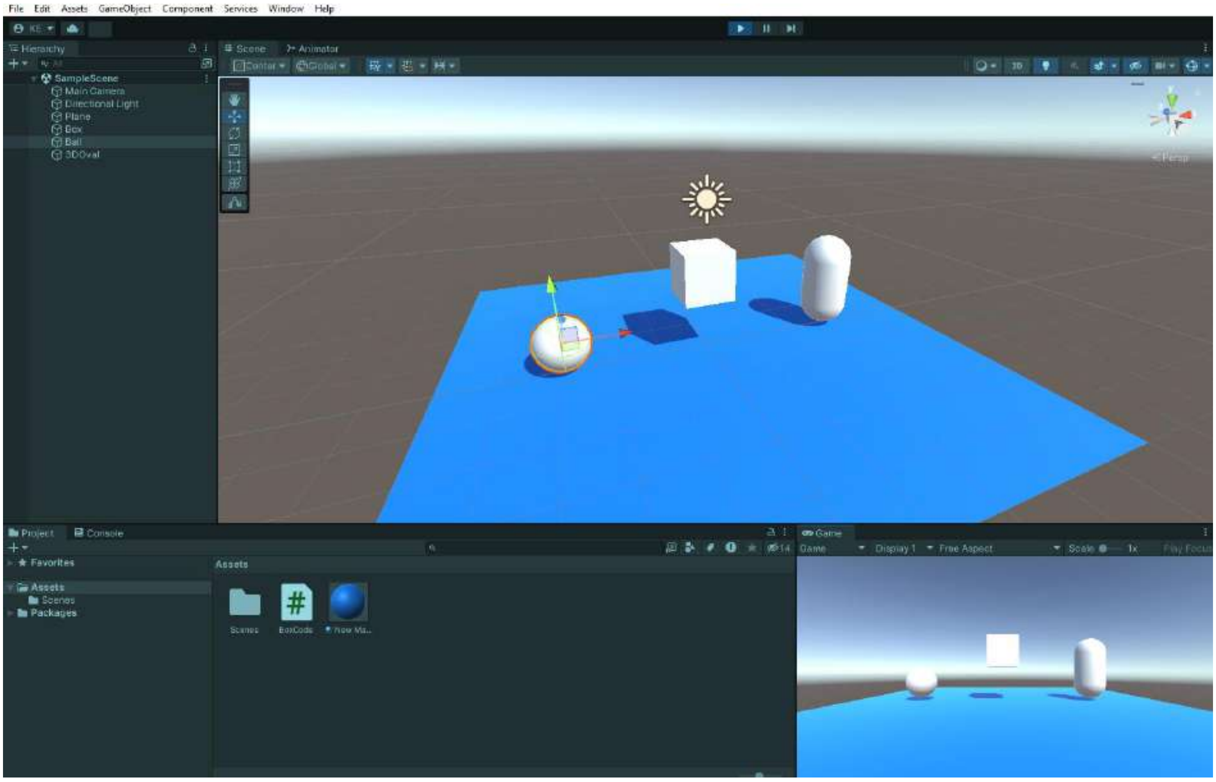


AddForce articulation can also be used to move with force application. Let's structure the command to move the **Box** object up,

GetComponent<Rigidbody>().AddForce(transform.up*50*Time.deltaTime);



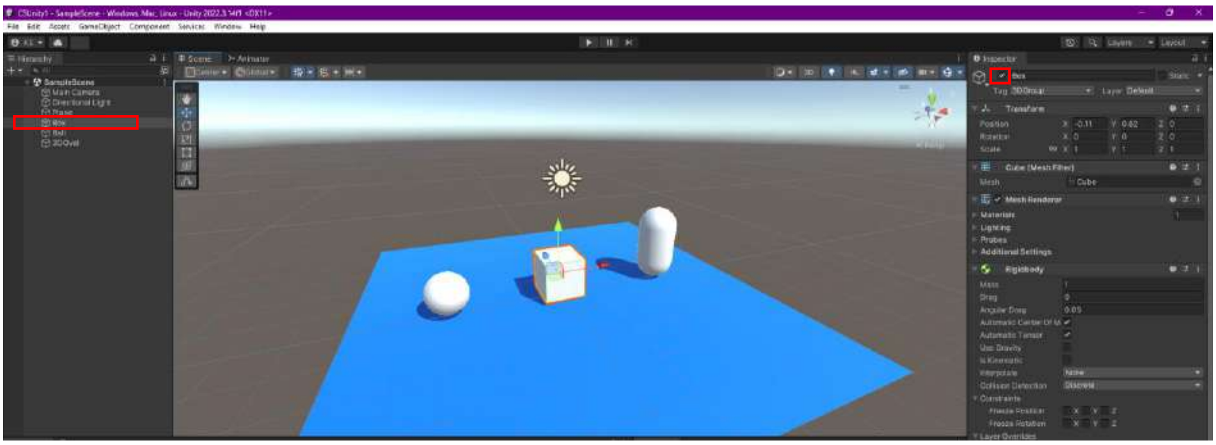

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UIElements;

public class BoxCode : MonoBehaviour
{
    private void Update()
    {
        if(Input.GetKey(KeyCode.RightArrow))
        {

            GetComponent<Rigidbody>().AddForce(transform.up * 50 * Time.deltaTime);
        }
    }
}
```



As long as we press the right arrow key, we see a movement that gradually accelerates and speeds up.

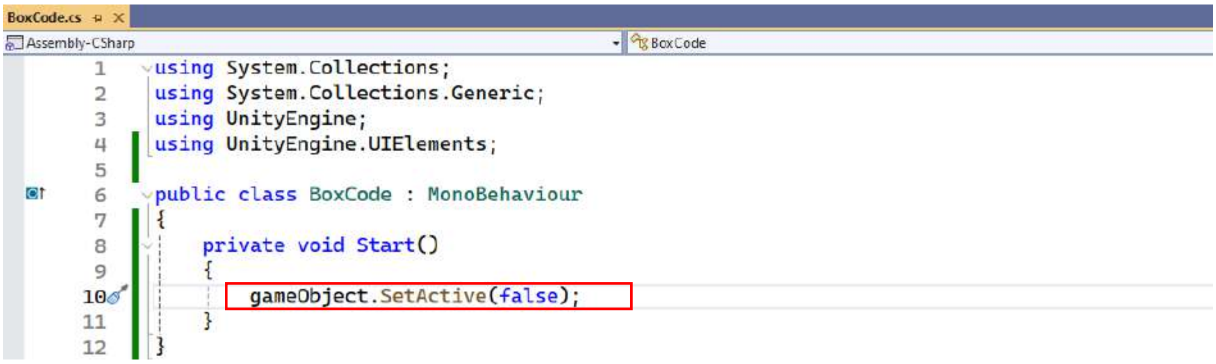
4.16.Activating/Deactivating Objects (SetActive) and Destroying Objects (Destroy)

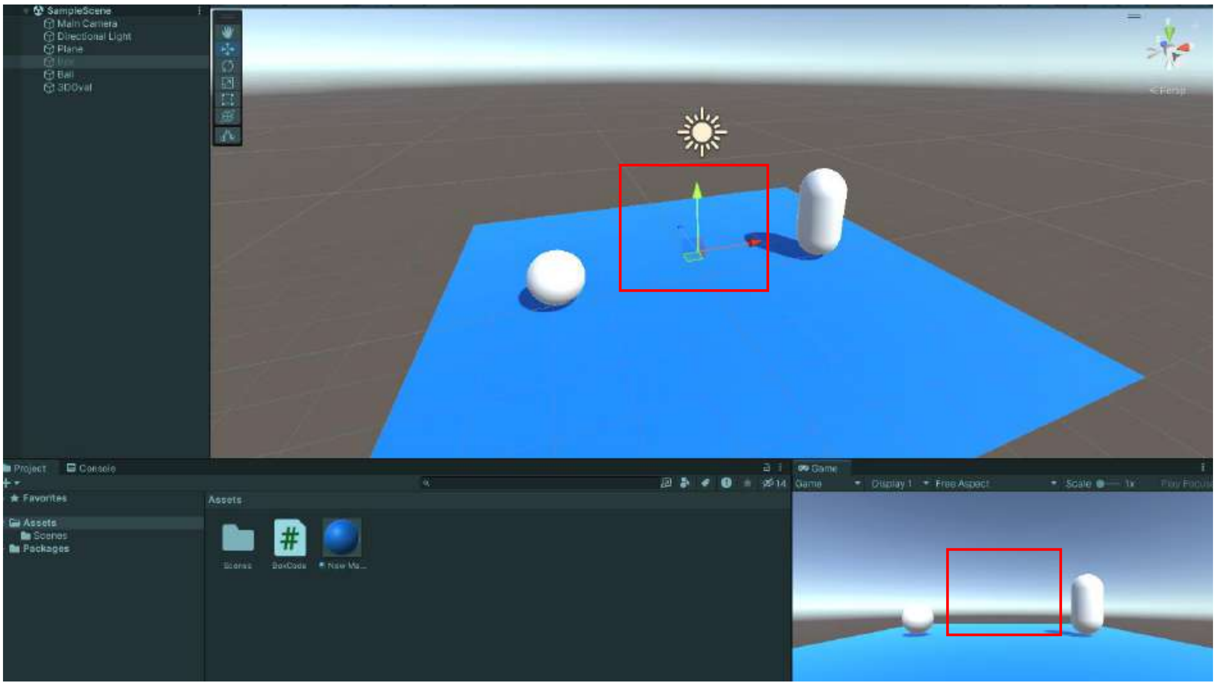
We have seen that elements of objects such as **Inspector>MeshRenderer** can be active or passive. In this section, let's see that the object itself can be made **active** or **passive**. An object can be made completely active or passive by ticking the **check box** on the line with its name in the **Inspector** and making it checked/unchecked. This is not a **Component**/element but the object itself.



When this box is deselected, the object is not deleted, but the object's visibility and all its components become **inactive**.**SetActive()** is the control command used for this purpose.

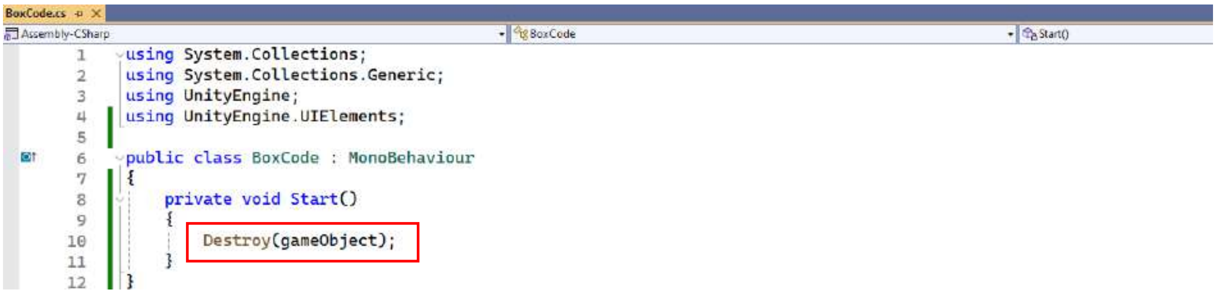
The general form for making an object inactive is **gameObject.SetActive(false);** In the **Play** Mode Game window, the **Box** object will be **inactive** and **invisible**.

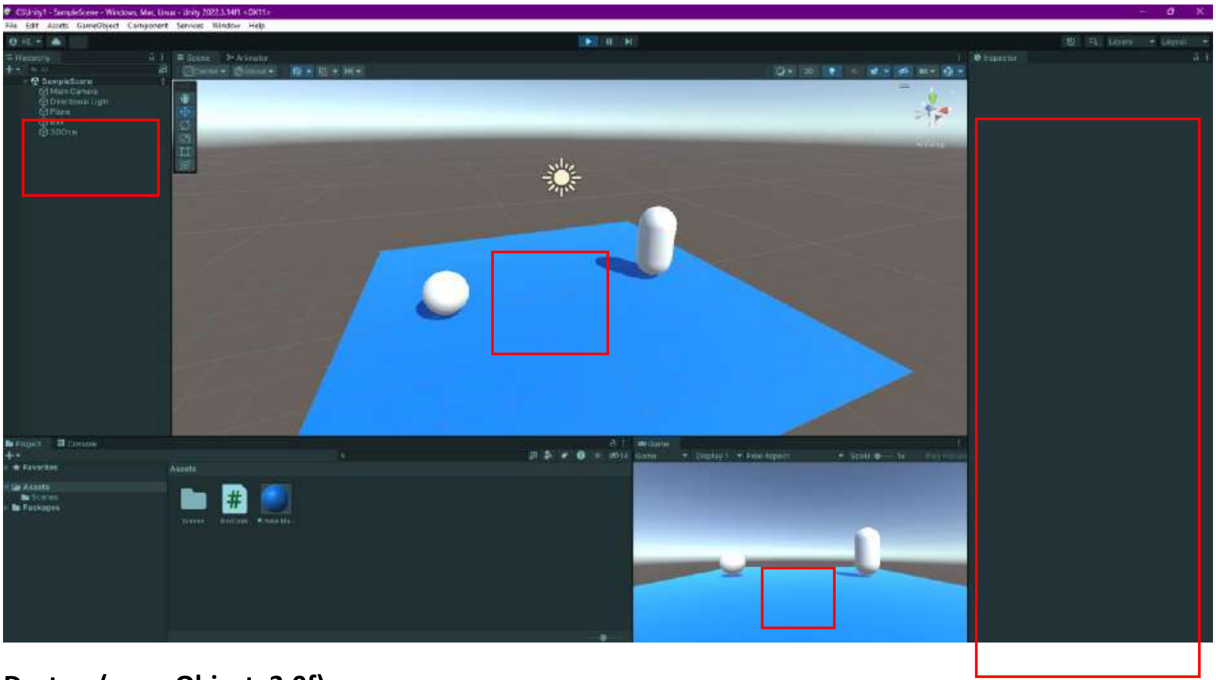




In the case of **SetActive(true)**, its visibility and all elements will be turned on.

Destroy() command is used to delete the object. Its general form is **Destroy(object, time)**. If we write **Destroy(gameObject);** in the script file, the object to which the C# file is attached will be understood as **gameObject** and the **Box** will be completely deleted along with the **Hierarchy** and **Inspector** sections in **Play** Mode.





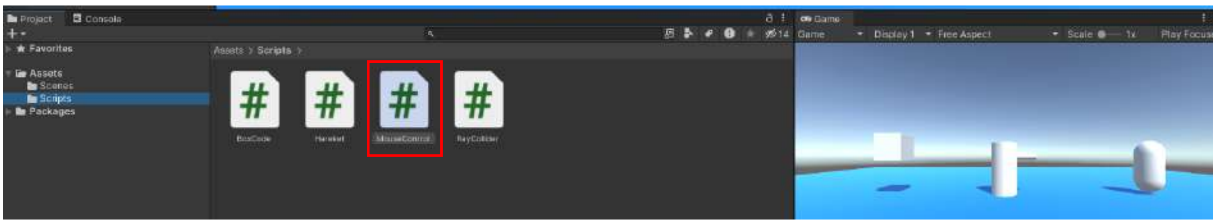
Destroy(gameObject, 3.0f);

If we write as such, the deletion will take place after 3 seconds.

4.17.Object Control with Mouse

It is possible to hold and drag objects onto the scene with the **mouse**. One of the most practical methods for this is to write code for the objects to be controlled with the mouse.

Let's create a C# Script named **MouseControl.cs** under **Assets>Scripts** and open it in Visual Studio.



For the application, we can use the **OnMouseDrag()** method in the Unity library. For this, we will calculate two **Vector3**-type variables. One will carry values for the coordinates of the mouse, and the other will carry values for the coordinates of the object.

We will also define a float-type variable for the third coordinate assignment. **Input.mousePosition.x** and **Input.mouse.Position.y** parameters can be used to transfer the **x, y** and **z** information of the mouse to the **Vector3** type variable that determines the mouse position. We can enter a fixed value for the **z** coordinate.

Vector3 mousePosition = new Vector3(Input.mousePosition.x, Input.mousePosition.y, posZ);
Vector3 objPosition=Camera.main.ScreenToWorldPoint(mousePosition);

Here, the vector variable **mousePosition** is defined, and the mouse position coordinates are assigned.

The vector **objPosition** converts the **mousePosition** information from the screen to the **WorldPoint** setting.

The screen is actually a 2-dimensional plane, and different perspective calculation methods can be used for the third dimension. While there are short explanations in the codes within the scope of the course, various written and visual details about their mathematical, geometric and trigonometric infrastructure can be found on the web.

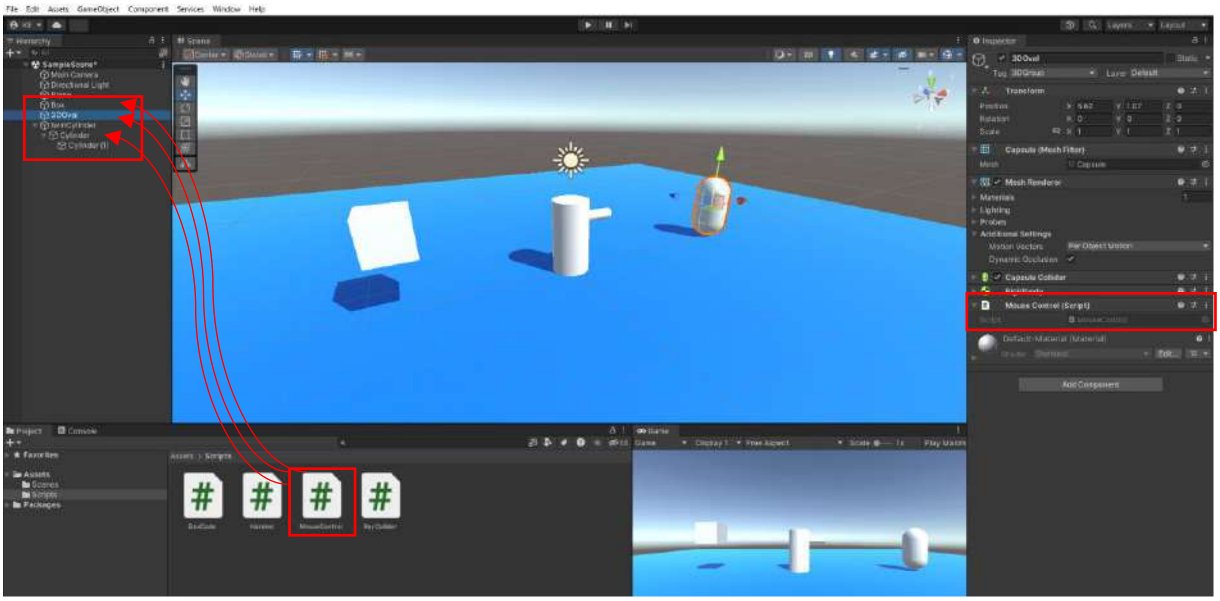
Finally, the transformation of the object to which these codes will be connected will be made.

transform.position = objPosition;

Let's code in Visual Studio.

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

public class MouseControl : MonoBehaviour
{
    private void OnMouseDrag()
    {
        float posZ = 10f;
        Vector3 mousePosition = new Vector3(Input.mousePosition.x, Input.mousePosition.y, posZ);
        Vector3 objPosition=Camera.main.ScreenToWorldPoint(mousePosition);
        transform.position = objPosition;
    }
}
```

We can connect the file we saved to all our objects (*Box, 3DOval and Cylinder*) in the scene.

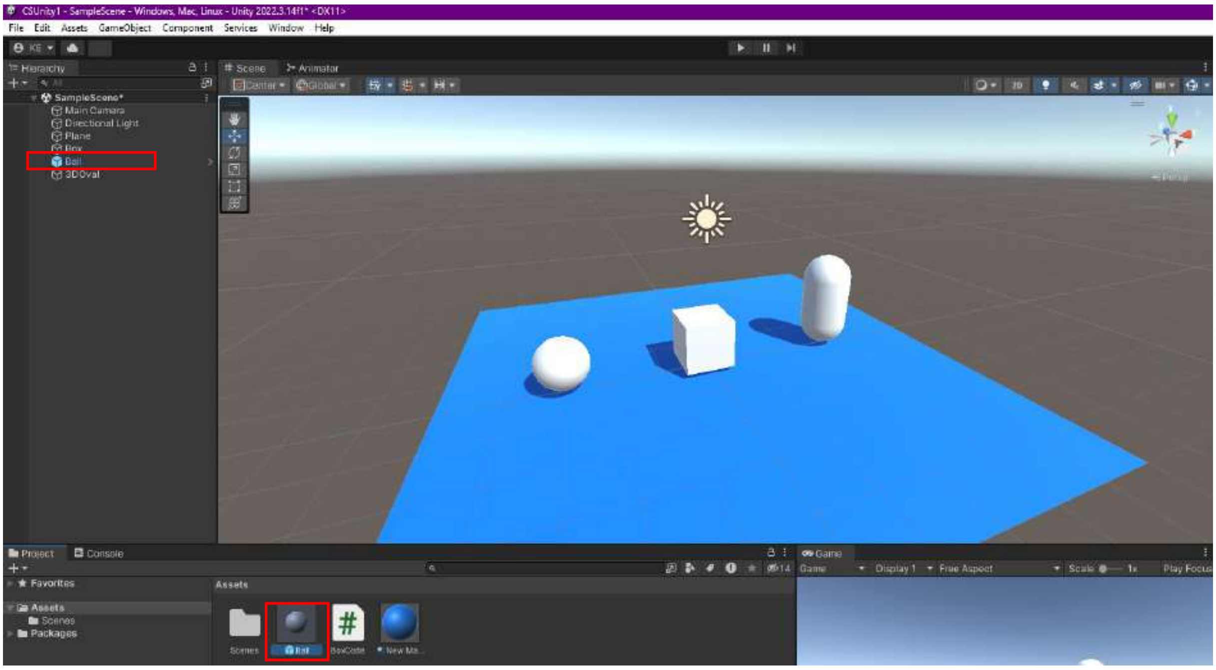


In Play mode, we can observe that we can hold all three objects with our mouse and drag them to the desired direction and height and that those with physical materials and jumping features move in this way.

4.18.Cloning Objects – Prefab and Instantiate

We may want to produce and duplicate objects with codes while the program is running. To do this, we need to create a prefab copy of that object in the **Assets** section in the **Hierarchy**. This form is called prefab.

Let's drag our **Sphere** object named **Ball** on the scene to the Assets section. See that the icon of the **Ball** object changes, and a copy is created in the Assets section. This copy is a **prefab Ball** asset.



If we drag the **Ball** asset into the Assets section to the scene over and over again, **Balls** with the same properties will be created. Changing the property of one of them will affect all of them.

After the **Ball prefab** is obtained, we can delete the Ball object in the scene. Even if this object is deleted from the **Hierachy**, we can drag as many clones as we want from Assets and add them to the scene.

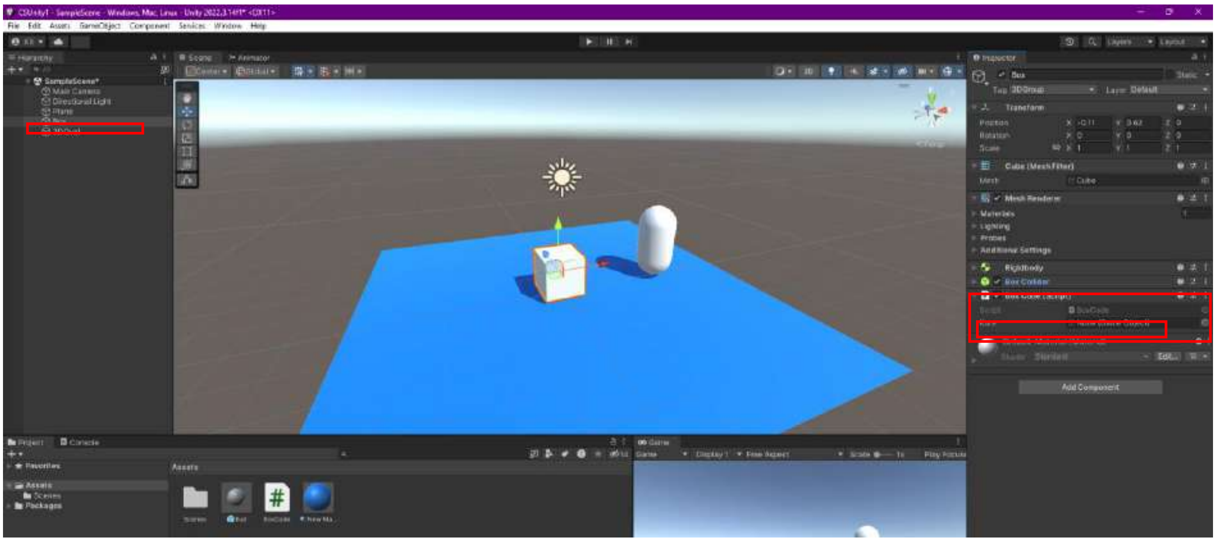
4.19.Adding Objects with Instantiate Coding – Spawn

Instantiate command is used to **add**, **load** and **clone prefab assets** with codes in the scene. This is also called **spawn** (reproduction/egg laying).

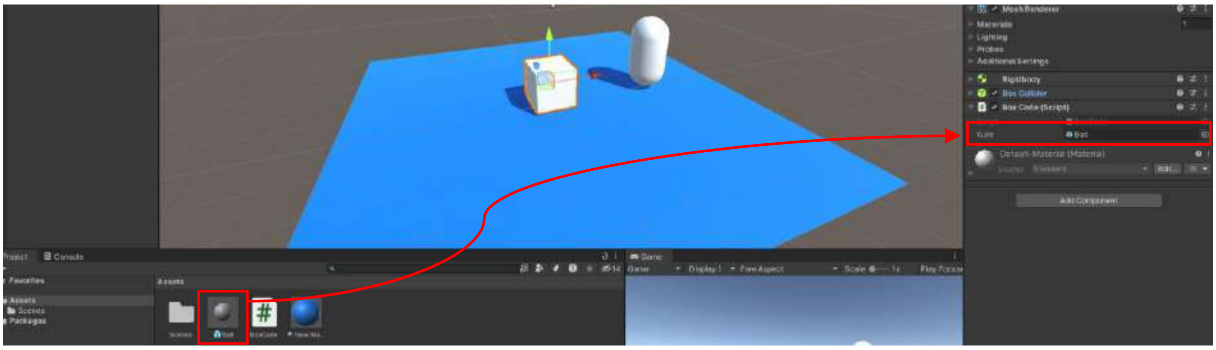
Let's make an edit according to our new title in **BoxCode.cs** file and define a **public GameObject** accessible in the editor. Here, the variable name is given as **kure**.

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

public class BoxCode : MonoBehaviour
{
    public GameObject kure;

    private void Start()
    {

    }
}
```

With this change, a variable field named **kure** has appeared in the **Inspector** section of the **Box** object because it is **public**. However, since a **prefab object** has not been connected yet, it is in the **None (Game Object)** state.



The **Ball prefab** in the Assets section is dragged and connected to this area.



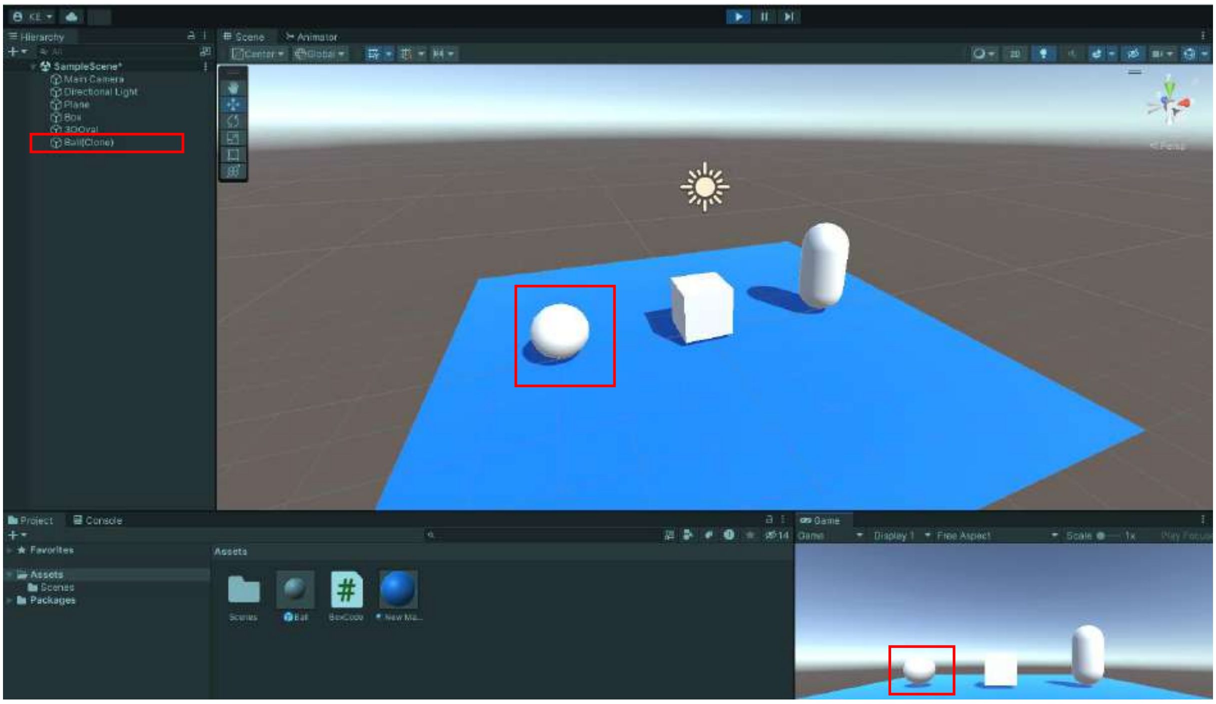
Now, we can use the **Instantiate()** command in the code lines.

The general form of this command is,

Instantiate (object, coordinate(x,y,z), angle(x,y,z))

However, it only allows a prefab copy of the invisible **Ball** object that we deleted from the **Hierarchy** with **Instantiate(kure)** to be added to the scene. **kure** in the **Instantiate(kure)** command is the **kure** defined as public **GameObject** in the code file and connected to the box in the editor with the **Ball prefab**.

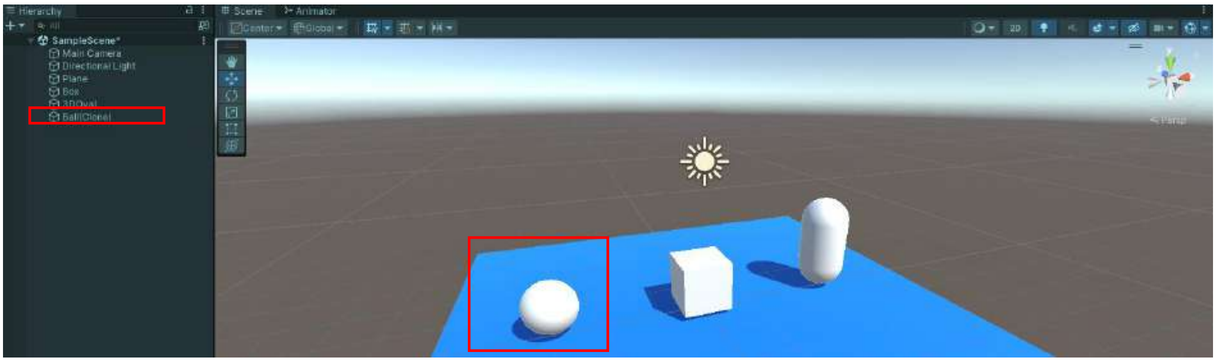
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UIElements;

public class BoxCode : MonoBehaviour
{
    public GameObject kure;

    private void Start()
    {
        GameObject newBalls = Instantiate(kure);
    }
}
```



A **clone** of the **Ball** object is added to the scene with **Play** mode. This object appears in the **Hierarchy** with the name **Ball(Clone)**.

Now, let's write the **Instantiate** command in accordance with the wide format. Here, **kure** object (i.e. **Ball prefab**) to **Box**, the identity function (i.e. its original rotation values) connected to the **Querternian** command used to determine the **position** and **angle** we want to appear; **Quaternian.identity**.

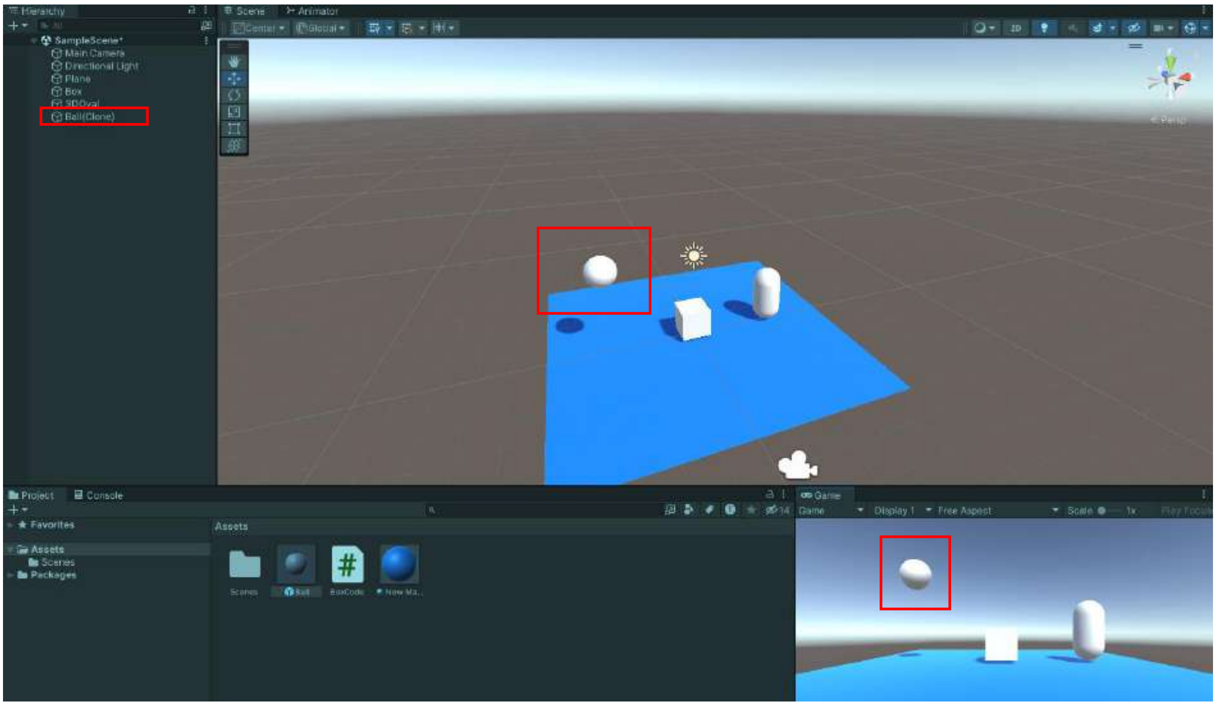
```csharp
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UIElements;

public class BoxCode : MonoBehaviour
{
    public GameObject kure;

    private void Start()
    {
        GameObject newBalls = Instantiate(kure, new Vector3(-3f,0.6f,0f), Quaternion.identity);
    }
}
```



We have obtained a similar image to the previous one. However, our definition and parametric modification capabilities have increased.

Now, specify the coordinate where the sphere will appear as (-3f, 6f, 0f). The **Ball** that appears above will fall due to the **Gravity** effect in the **Rigidbody**.

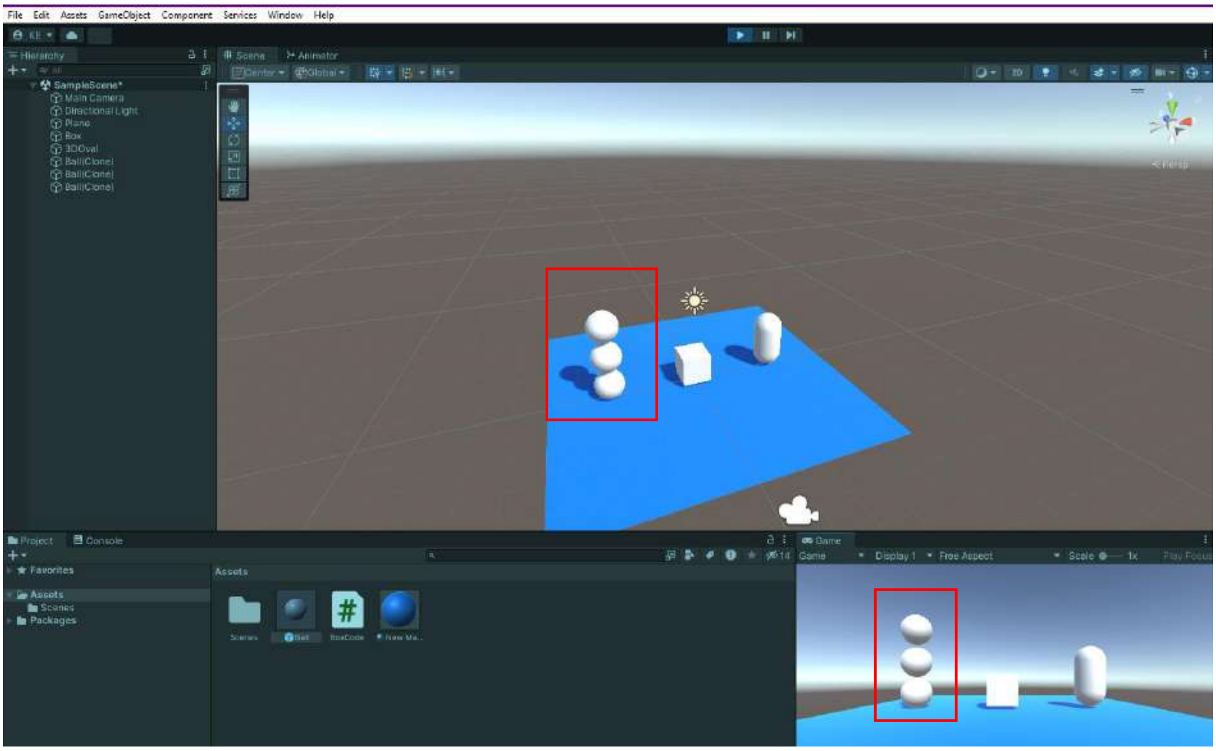
```csharp
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UIElements;

public class BoxCode : MonoBehaviour
{
    public GameObject kure;

    private void Start()
    {
        GameObject newBalls = Instantiate(kure, new Vector3(-3f,6f,0f), Quaternion.identity);
    }
}
```



Let's make changes to the code to clone 3 Balls. For this purpose, let's create a **for** loop and make them fall. With the expression **new Vector3(-3f, 4f+i,0f)** in the loop, each **clone kure** is made to appear four units higher in the **Y** direction (**vertical axis**) by the value of the **i** variable. In this way, the spheres (**kure**) are prevented from appearing at the same point.

BoxCode.cs

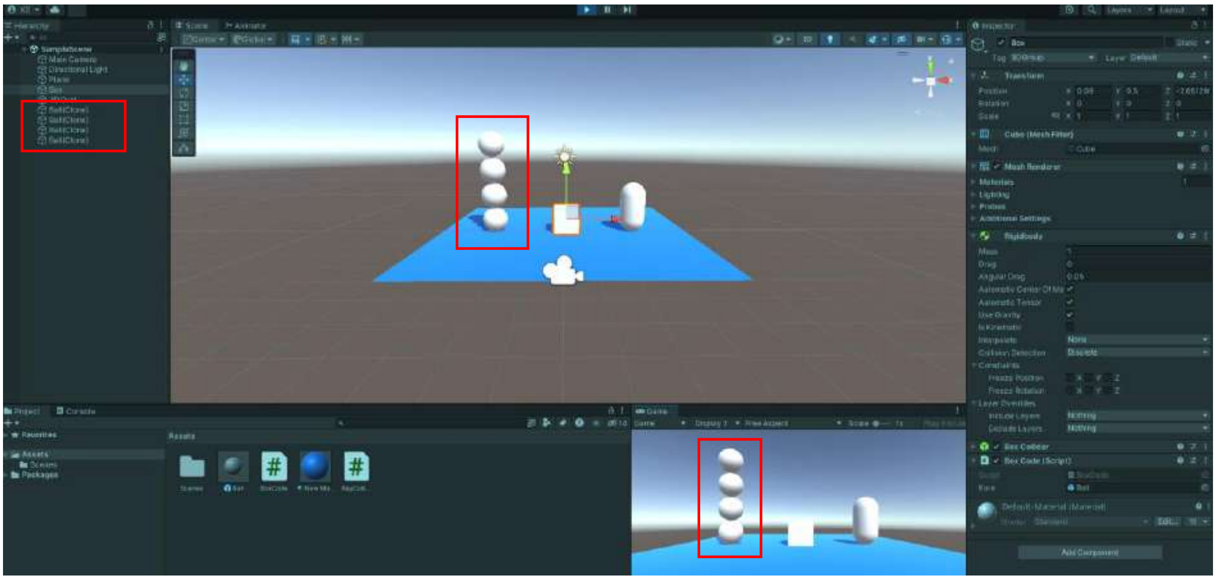
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UIElements;

public class BoxCode : MonoBehaviour
{
    public GameObject kure;

    private void Start()
    {
        for (int i = 0; i < 3; i++)
        {
            GameObject newBalls = Instantiate(kure, new Vector3(-3f, 4f+i, 0f), Quaternion.identity);
        }
    }
}
```



Another scenario is that when any key is pressed the **Ball prefab** object is **added** to the scene.

```
BoxCode.cs
Assembly-CSharp                                   BoxCode

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UIElements;

public class BoxCode : MonoBehaviour
{
    public GameObject kure;

    private void Update()
    {
        if (Input.GetKeyUp(KeyCode.Space))
        {
            Instantiate(kure);
        }
    }

}
```

In its simplest form, as a result of these codes, each time the **Space Bar** is pressed, the **Ball prefab** is added to the same coordinate. However, since it is a **Rigidbody**, they push each other upwards. In the example run, the space bar was pressed four times.

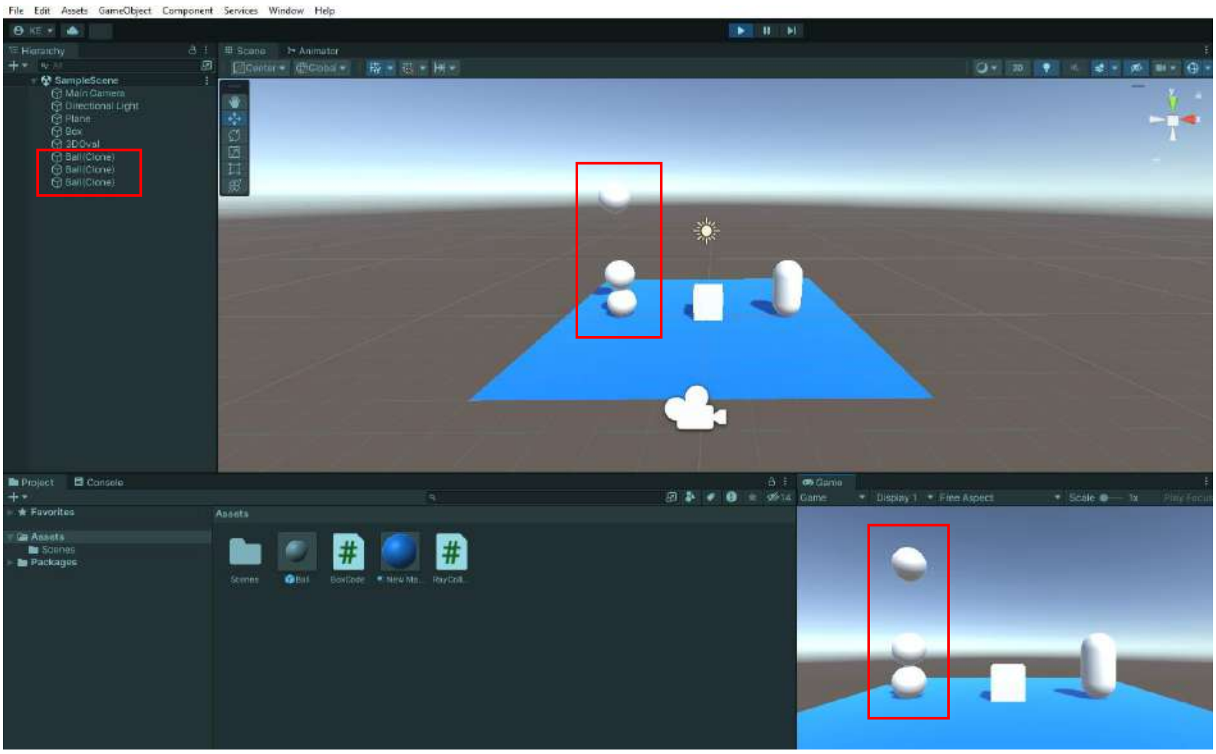


However, if we want the **Ball prefabs** to fall from the top instead of pushing them from the bottom to the top by keystrokes, we need to enter **new Vector3 (x,y,z)** coordinates of the point we want them to fall from.

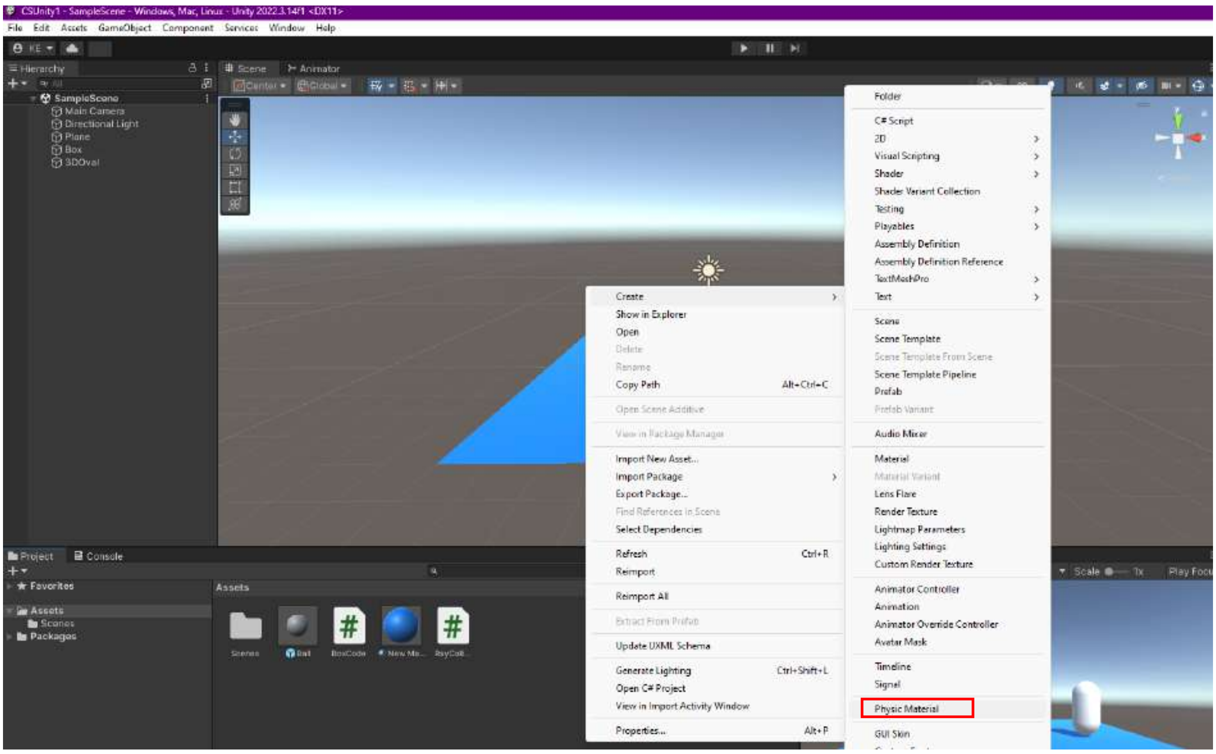
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UIElements;

public class BoxCode : MonoBehaviour
{
    public GameObject kure;

    private void Update()
    {
        if (Input.GetKeyUp(KeyCode.Space))
        {
            Instantiate(kure, new Vector3(-3f,5f,0f),Quaternion.identity);
        }
    }

}
```

When we run the application, **Ball prefabs** start falling from the coordinate **(-3f,5f,0f)** depending on the **Space Bar** keystroke. Also, notice that a **Ball(Clone)** is added to the **Hierarchy** with each keystroke.

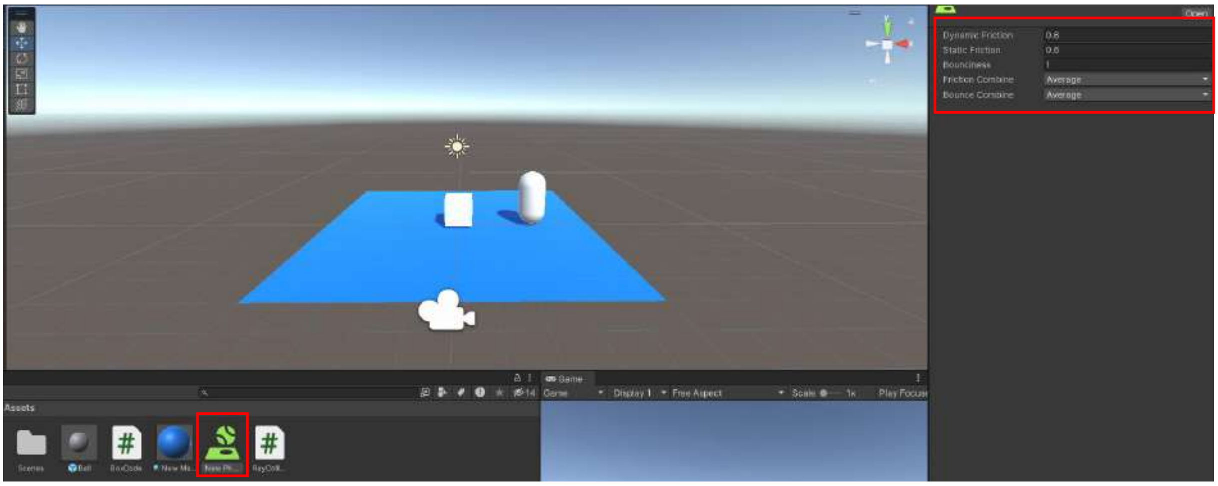
Since there is no physical jumping feature in the **Ball prefab**, the falling spheres (**kure**) pile up on top of each other.



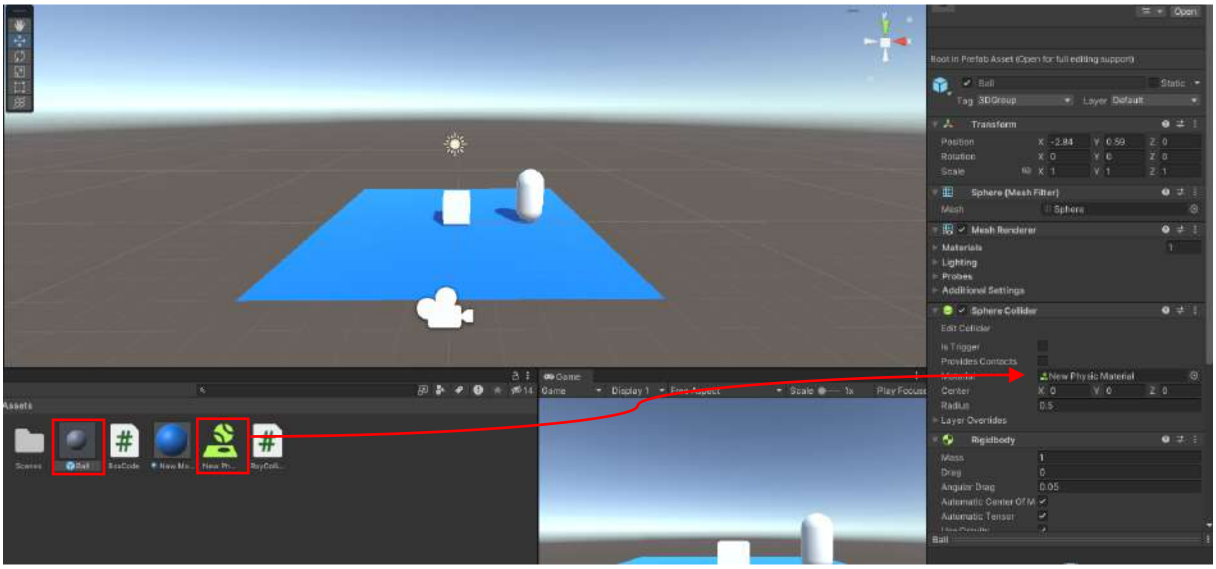
For the **Ball prefab**, first add a **Physic Material** to the Assets section.



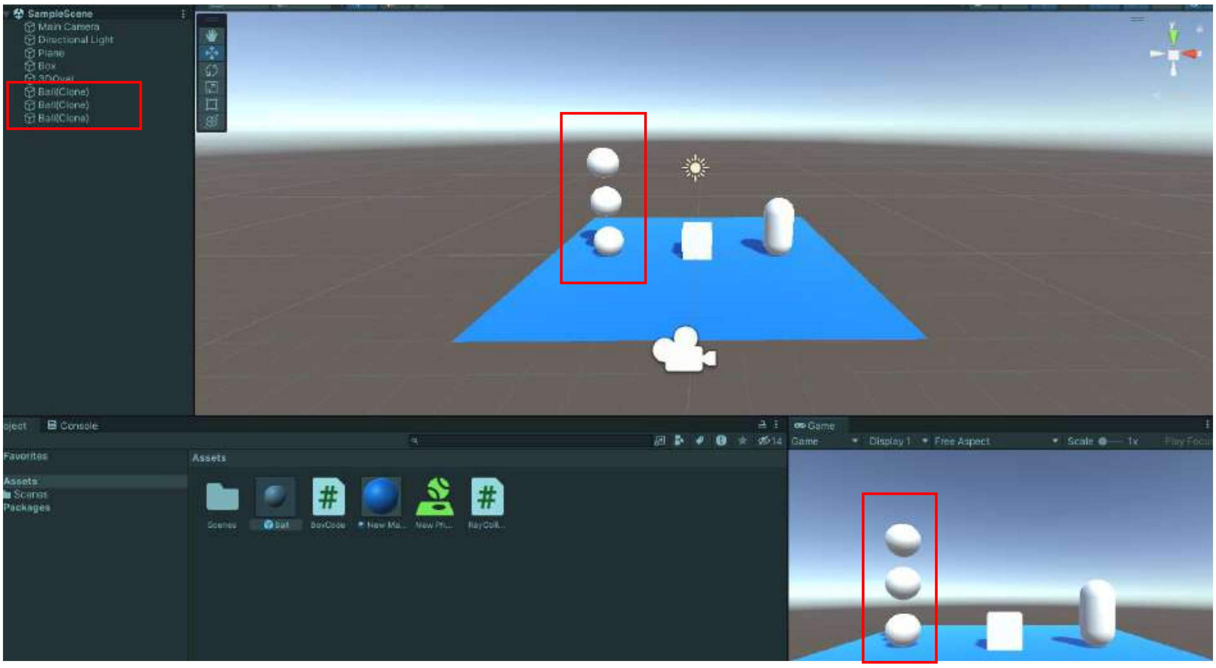
Then, adjust its **bounciness** settings.



Now, drag and connect the **Physic Material** to the **Material** field of the **Ball prefab**.



Run it in **Play** mode again. We can see that the balls bounce under the effect of the physics material, and after a while, they move left and right and fall from the **Plane**.

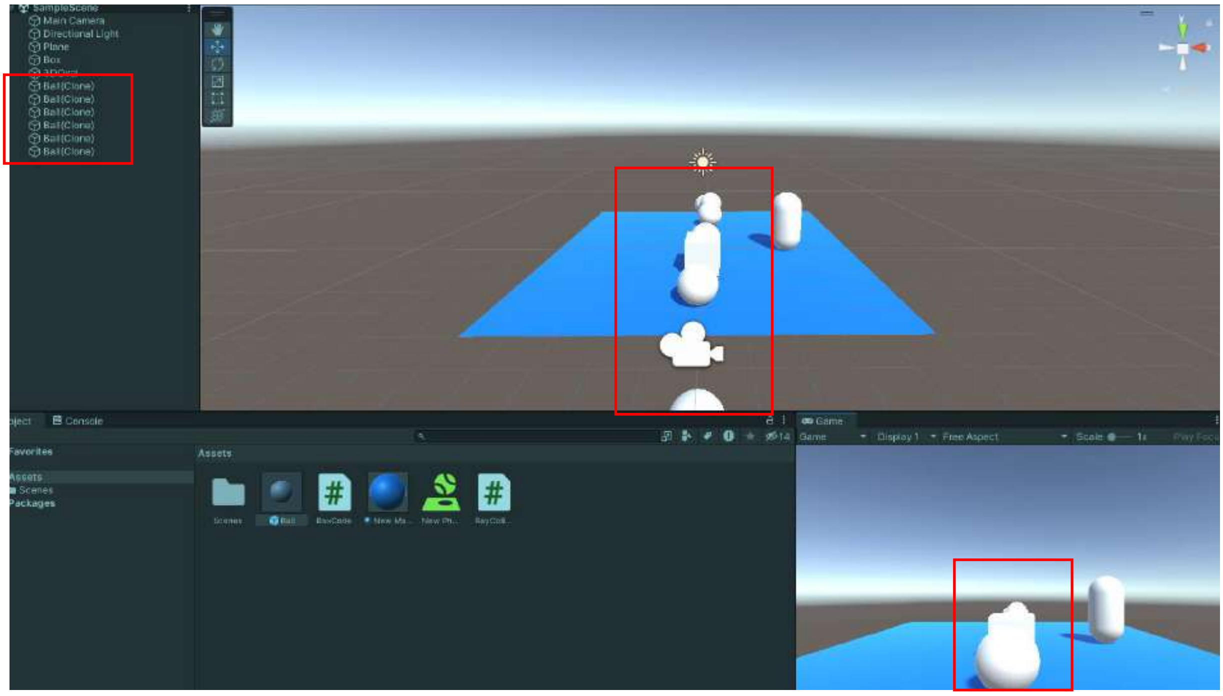


Alternatively, if we write

Instantiate(kure, new Vector3(-3f,5f,0f),Quaternion.identity); instead of

Instantiate(kure, transform.position, transform.rotation);

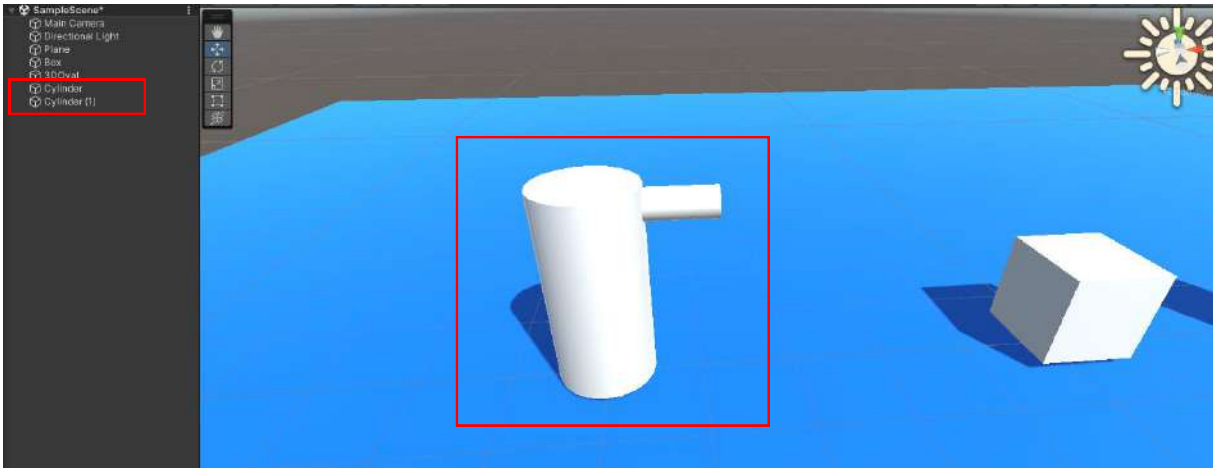
every time we press the key, the code will appear in the **Transform position** and **rotation** of the **sphere** we are connected to. However, this will collide with the present sphere(s). Instead, we can open an empty game object and connect the **code** file to it.



Using Lerp and eulerAgles in Movements

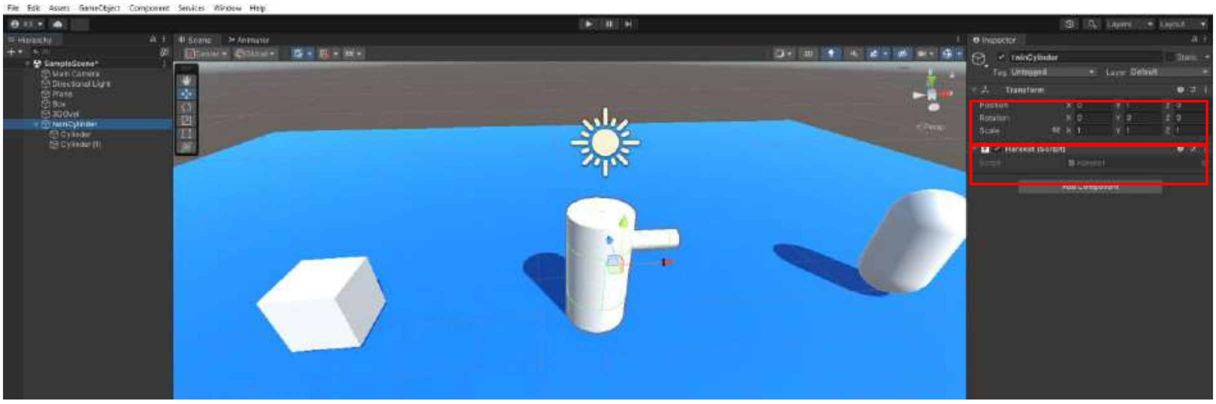
Lerp literally means linear **interpolation**. It is generally used in Unity C# coding to provide accelerated movement. **Euler** is the **angular exponent value** used in **trigonometry**, which forms the basis of **natural logarithms**. It is used to define objects in terms of angles.

Let's create two cylinders, one large and one small, in our scene for the project that will rotate a **GameObject** horizontally by **0, 90, 180** and **270** degrees with keystrokes. Place the smaller cylinder on top of the larger one, as shown in the figure.



Now, create an **empty GameObject** object with **Hierarchy>Create>Empty**. This can be named **twinCylinder**. Drag two cylinders to this object and create a **parent-child** relationship. Let's make sure that all objects are in the **center**. If they are not, move them to the center (**0, 0, 0**) position with **Transform>Reset**.

Add a **new C# Script** file to the Assets section. Drag the file named **Hareket.cs** to the **twinCylinder** empty object and connect it.



Since it will be a **3D vector** movement, define a variable by resetting its initial value.

Vector3 vec = Vector3.zero;

As there will be rotation depending on the key, continuity is required. Therefore, write the codes in the **Update()** method.

Since the rotation will be in the **Inspector>Transform** area, we can check/update the **vector direction/angle** by assigning **Vector3.Lerp** to the **eulerAngles** function as follows.

transform.eulerAngles = Vector3.Lerp (transform.eulerAngles, vec, 0.1f);

Here, it has a template like **Vector3.Lerp(from where, to where, in how much time)**. Because there is **transform.eulerAngles** on both the left and right of the equation, it is understood that there will be a change depending on the **vec** variable (**y-axis**) and not the value of this point. The speed of this update is in **float time type,** and 0.1f is assigned here.

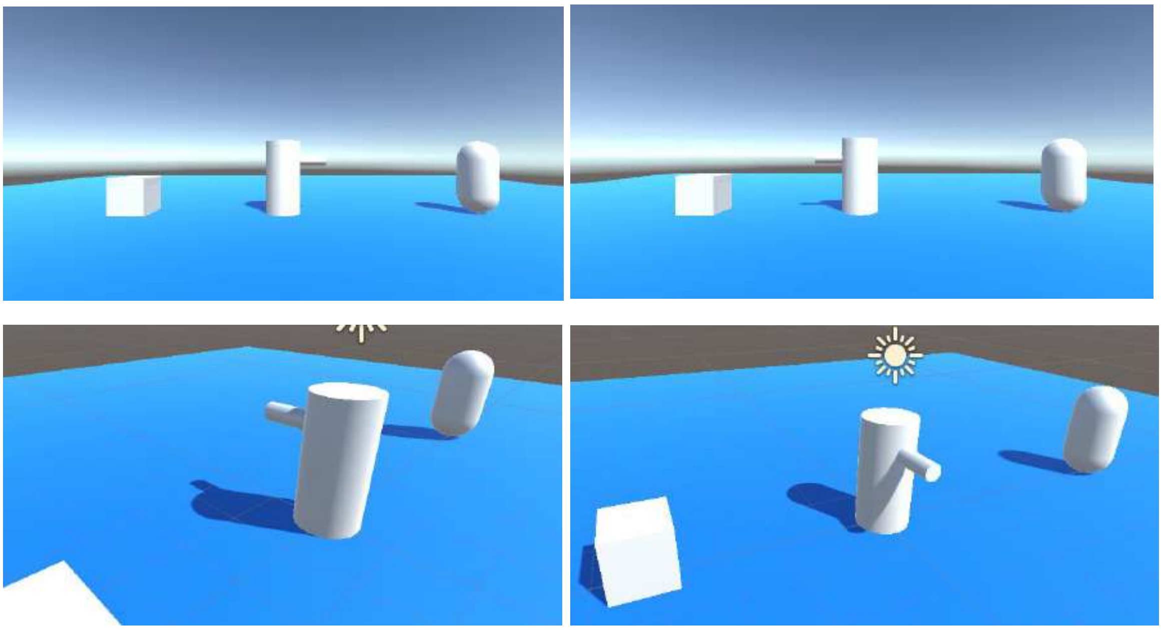
Now, specify what will happen when the **W, A, S, and D** keys are pressed and assign a value to our vector variable **vec**.

```
if (Input.GetKeyDown(KeyCode.W))
{ vec = new Vector3 (0f, 0f, 0f); }
if (Input.GetKeyDown(KeyCode.D))
{ vec = new Vector3 (0f, 90f, 0f); }
if (Input.GetKeyDown(KeyCode.S))
{ vec = new Vector3 (0f, 180f, 0f); }
if (Input.GetKeyDown(KeyCode.A))
{ vec = new Vector3 (0f, 270f, 0f); }
```

The value of the **vec** variable is constantly updated in the **Update()** method. If a key is pressed, it is returned to its new position.

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

public class Hareket : MonoBehaviour
{
    Vector3 vec = Vector3.zero;

    // Update is called once per frame
    void Update()
    {
        transform.eulerAngles = Vector3.Lerp(transform.eulerAngles, vec, 0.1f);

        if(Input.GetKeyDown(KeyCode.W))
        {
            vec =new Vector3(0f, 0f, 0f);
        }
        if (Input.GetKeyDown(KeyCode.D))
        {
            vec = new Vector3(0f, 90f, 0f);
        }
        if (Input.GetKeyDown(KeyCode.S))
        {
            vec = new Vector3(0f, 180f, 0f);
        }
        if (Input.GetKeyDown(KeyCode.S))
        {
            vec = new Vector3(0f, 270f, 0f);
        }
    }
}
```

Since the **code** file is connected to the parent **twinCyliner**, both cylinders in the child state will move simultaneously.

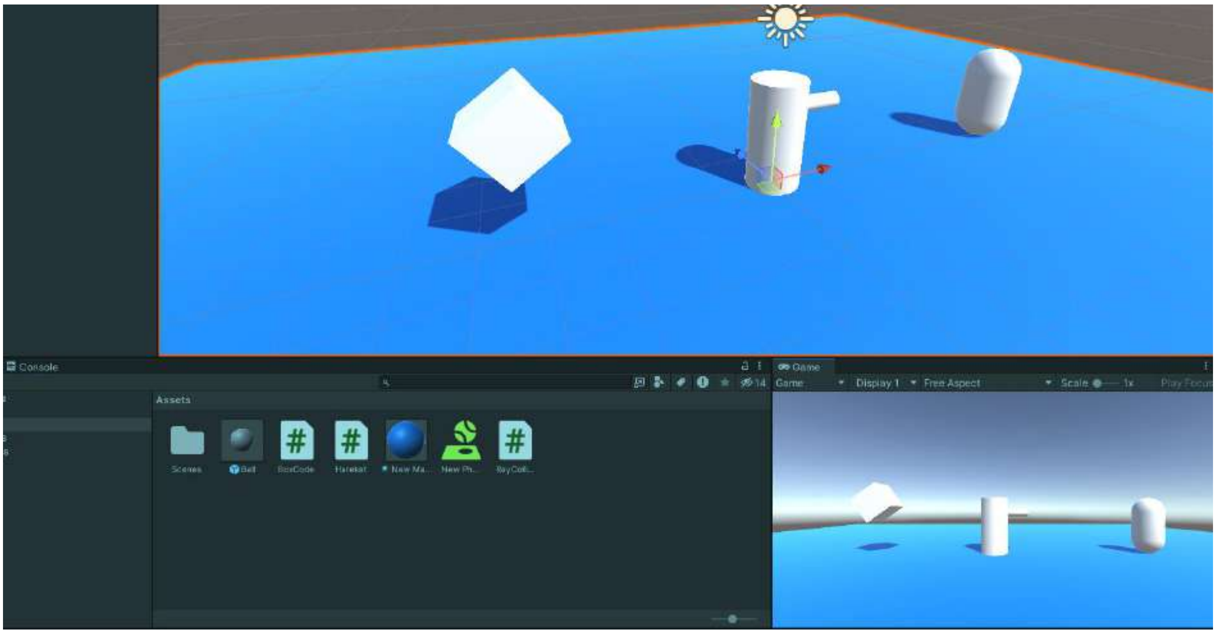


The main purpose of using the Vector3.Lerp function is to soften the sharpness of movements and provide a certain smoothness in turns.

Now, assign an **alternative** rotation **script** to our **Box** object and try this on **Box**. **Position** it up and remove **gravity**. Reorganize the **Motion.cs** code as follows.

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

public class Hareket : MonoBehaviour
{
    public float donmeHizi = 5.0f;

    // Update is called once per frame
    void Update()
    {
        Vector3 rot = transform.eulerAngles;
        transform.rotation=Quaternion.Euler(rot.x, rot.y, rot.z+Time.deltaTime*donmeHizi);
    }
}
```

Let's define the variable **donmeHizi**, which is a **real** and **public number**, and assign it a value like 5.0f. Define the **rot** variable of type **Vector3** in the **Update** method and assign the **transform** information with the **eulerAngles** function.



In **Play** mode, it will be seen that the **Box** object **rotates** at the specified speed without any keystroke.

As the speed parameter is **public**, the editor can also change it.

4.20.Raycast – Collision Control by Ray Spreading

In Unity, **rays** that are **not visible** to the user can be emitted between **objects**, and collision control can be provided.

Previously, **Collider** and **Rigidbody** physics elements were required for two objects to **collide**. In **Raycast-**type collisions, both objects need to have a **Collider**, but a **Rigidbody** is not required. However, there is no harm in having it.

Ray propagation can generally be in two ways:

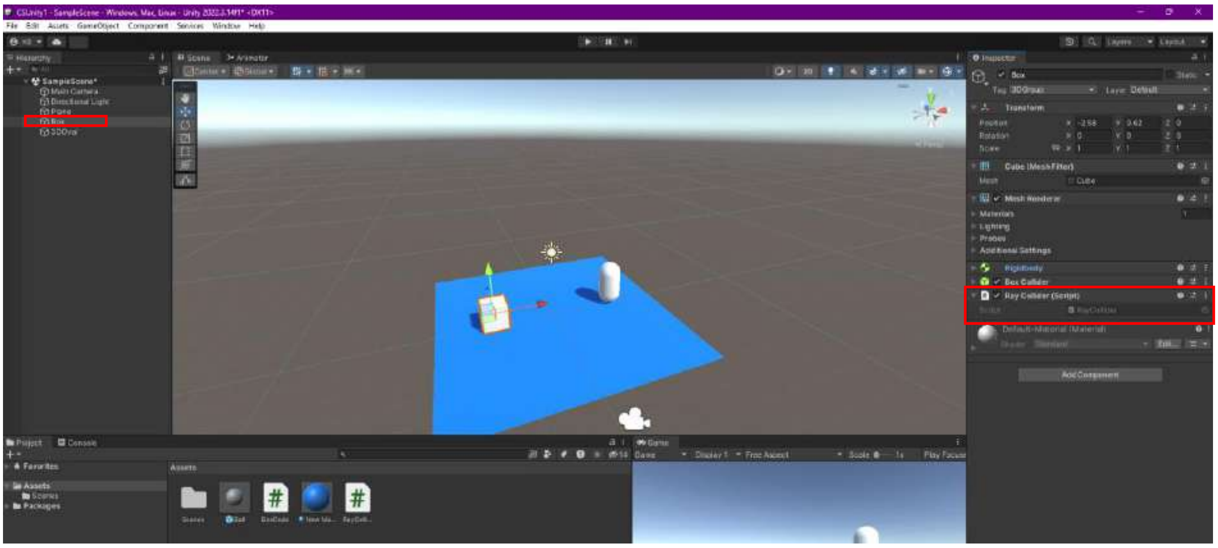
i- From Object

ii- From Camera

iii- From Mouse

i- Ray propagation from Object

Let's come to the Unity project since how it is used in the application will be better understood. We had two objects named **Box** (**Cube**) and **3DOval** (**Capsule**) in the scene. Now, create a **C# Script** named **RayCollider.cs** to connect to the **Box** object.



Create a variable defined as **RaycastHit** in the script file that carries the information about the object that the ray (**Ray**) **hits**.

RaycastHit obj;

Hence, the **obj** variable has the feature of carrying some information about the object that the **ray hits**.

Since the ray will be sent for continuous control purposes, write our codes in the **Update()** method.

If we consider the **ray** as a **physics element**, the code **formula/template** for sending the ray can be written as follows.

Physics.Rascast(start_position, ray_direction, out variable, ray_length)

We can adapt it for our project as follows,

Physics.Rascast(transform.position, transform.right, out obj, 5.0f)

Here,

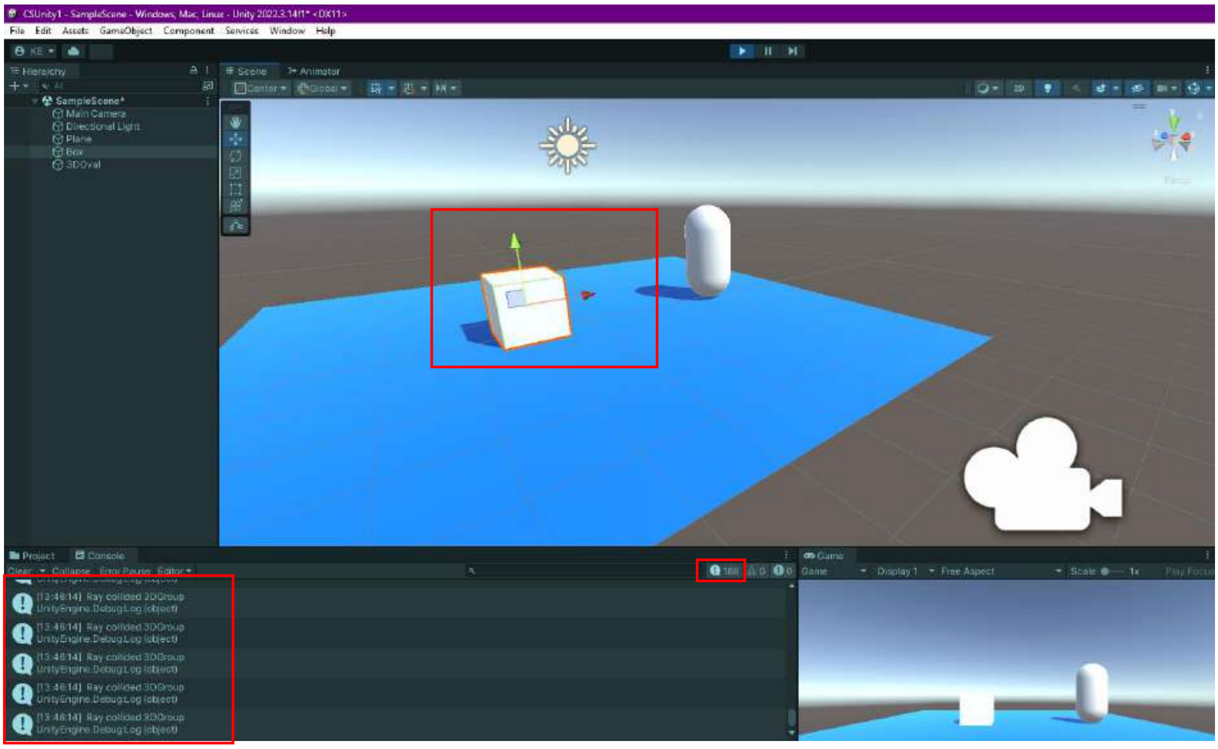
with **transform.position,** we determine the **position** of the **object** (**Box**) to which these codes are connected as the **starting** point. **transform.right** will be sent to the **right side** (**X-axis**) of the **object** (**Box**) (if it were **Vector3**, it would be the **X-axis** according to the **scene** reference). With **out obj**, the information about the object that the **ray hits** is assigned to **obj**, a variable of type **RascastHit**, as an **output**. The maximum **ray length** (**max_length**) is written as a **unit distance** in **float** type.

If we put this code expression in the **if** form, we can determine whether the **ray hit** an **object**. If this **collision** has occurred, it can be tested with a separate **if** statement whether it is the **targeted object**. If it **hits** the **right object**, we can **declare** this to the **console** with a **message**. The coded version of this setup is given below.

if (Physics.Raycast(transform.position, transform.right, out obj, 3.0f)
{ if (obj.collider.gameObject.tag== "3DGroup")
{ Debug.Log(" Ray collided 3DGroup "); }
}

In the second **if** statement, **obj**, which carries the information about the object that the ray **hits** and the **tag** name of the **gameObject** that it collides with (**collider**), **3DGroup**, gives a message with **Debug.Log**.

```
RayCollider.cs
Assembly-CSharp          RayCollider          Update()

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UIElements;

public class RayCollider : MonoBehaviour
{
    RaycastHit obj;

    private void Update()
    {
        if (Physics.Raycast(transform.position, transform.right, out obj, 3.0f))
        {
            if (obj.collider.gameObject.tag == "3DGroup")
            {
                Debug.Log(" Ray collided 3DGroup ");
            }
        }

    }
}
```

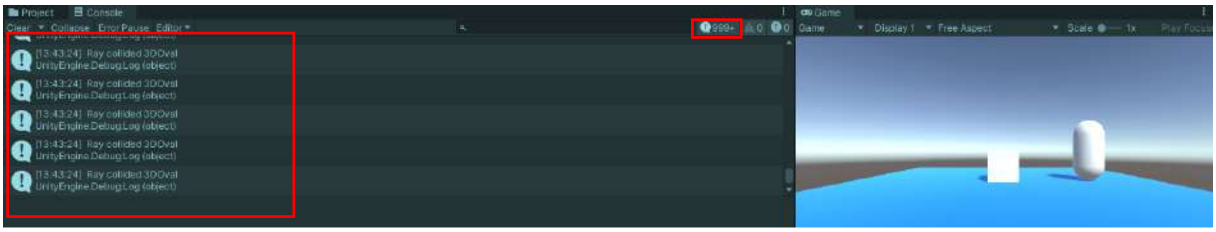


When we run it in **Play** Mode, we pull the **cube** (**Box**) to the **right**. When we reach the **maximum length** of 3.0f, **Debug.Log()** command runs continuously. When we zoom out, the messages are **cut off**.

Alternatively, instead of **transform.right** in the code, we can specify a direction we want with **new Vector3(X, Y, Z)** coordinates [**new Vector3(1,0,0) for the right direction (X)**].

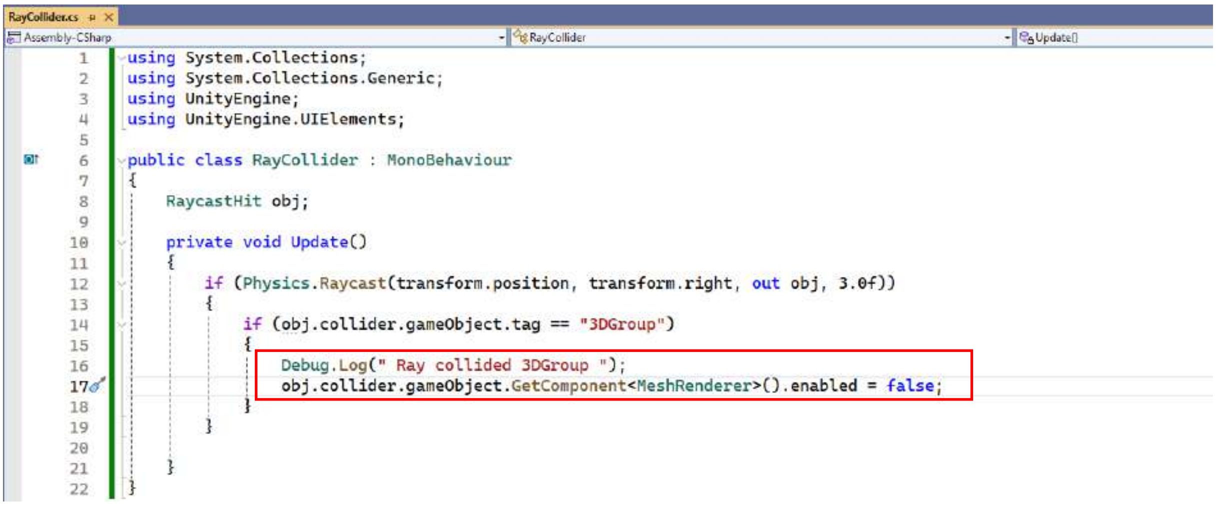
Another alternative is to use a **name** instead of a **tag**.

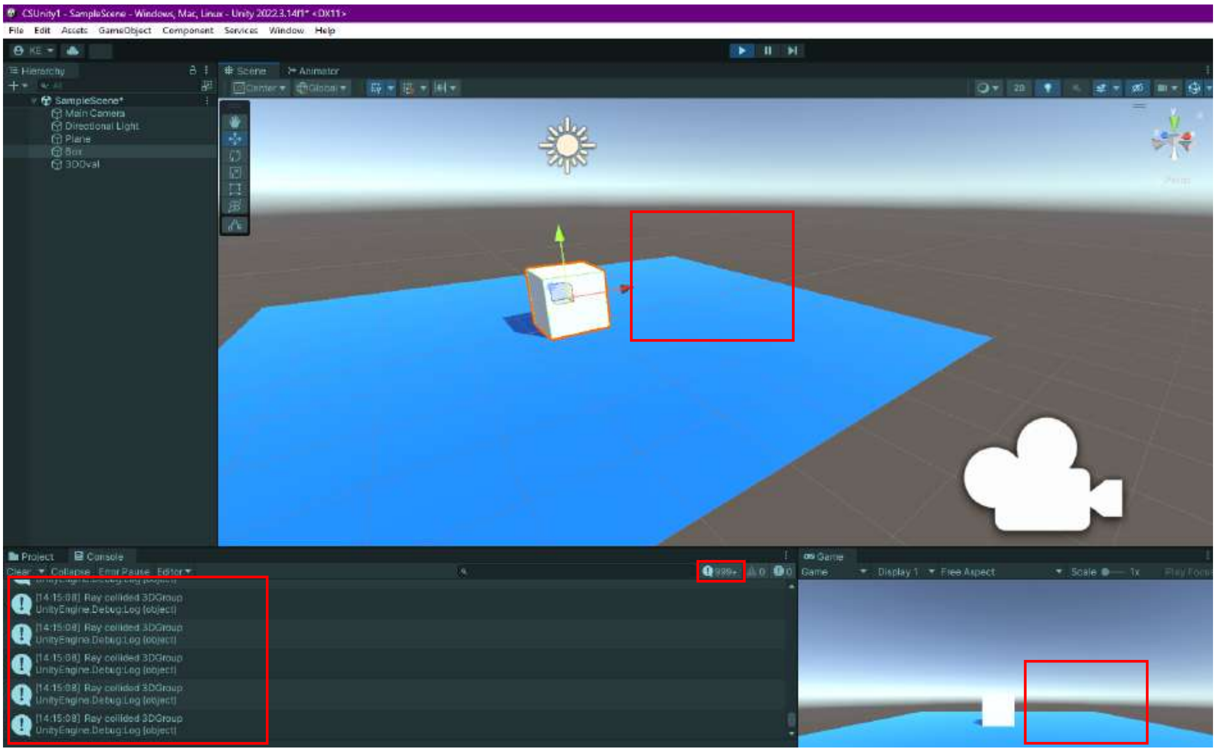
if (obj.collider.gameObject.name== “3DOval”)



After checking that it works with the message when the **ray hits** the object we specified, we can access its **Mesh Renderer** property and make it **invisible**.

obj.collider.gameObject.GetComponent<MeshRenderer>().enabled=false;





When the capsule enters the **ray range**, it becomes **invisible**. However, since it is not deleted, the **Debug.Log** command continues to work.

Alternatively, if you use

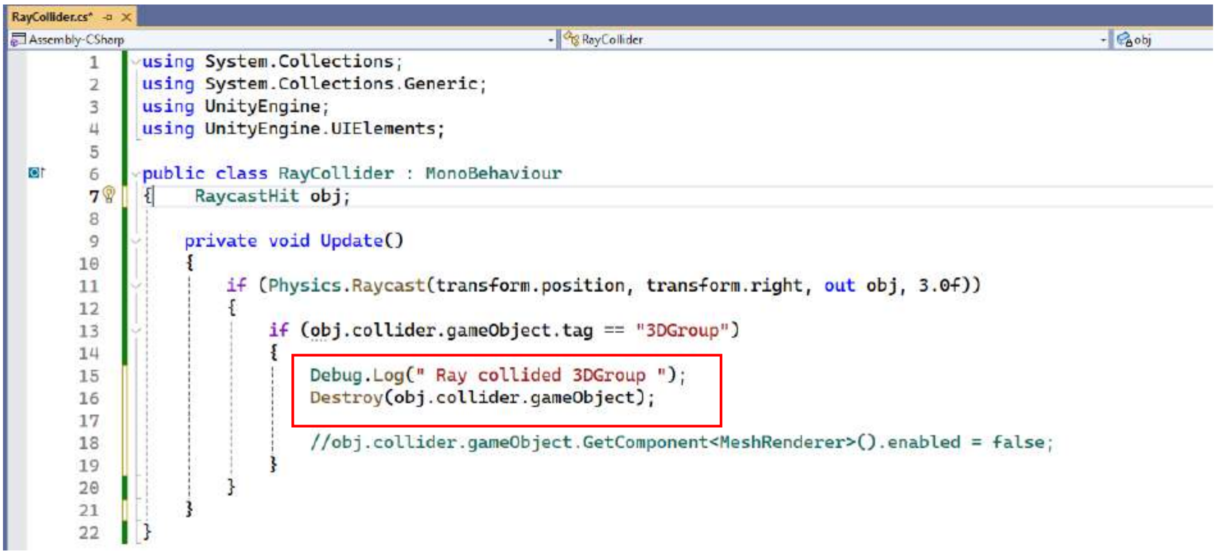
obj.collider.gameObject.SetActive(false);

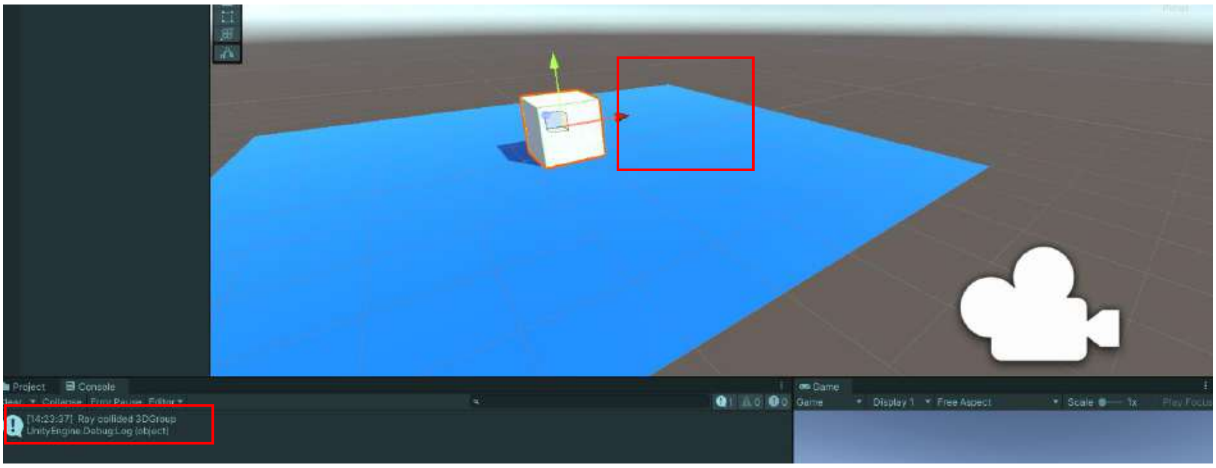
The **capsule** becomes **invisible,** and the **collision** stops.

We can also run it by deleting the object.

Destroy(obj.collider.gameObject);

With this operation, the capsule named **3DOval** is **deleted**, **all** its **components** are **closed,** and the **collision stops**.





4.21.Ray Emission from Camera

To emit a **Ray** (ray) from the **camera**, first, a **ray** is **created**, and for this, a **Ray**-type **variable** must be defined. This variable should find the **camera object** to be used by **name**, reach its **Camera Component field,** and **emit** a **ray** from the point whose coordinate is determined with the **ViewportPointToRay (screen coordinate)** command. Now, let's apply this information to a variable named **Ray** type **isinYay**.

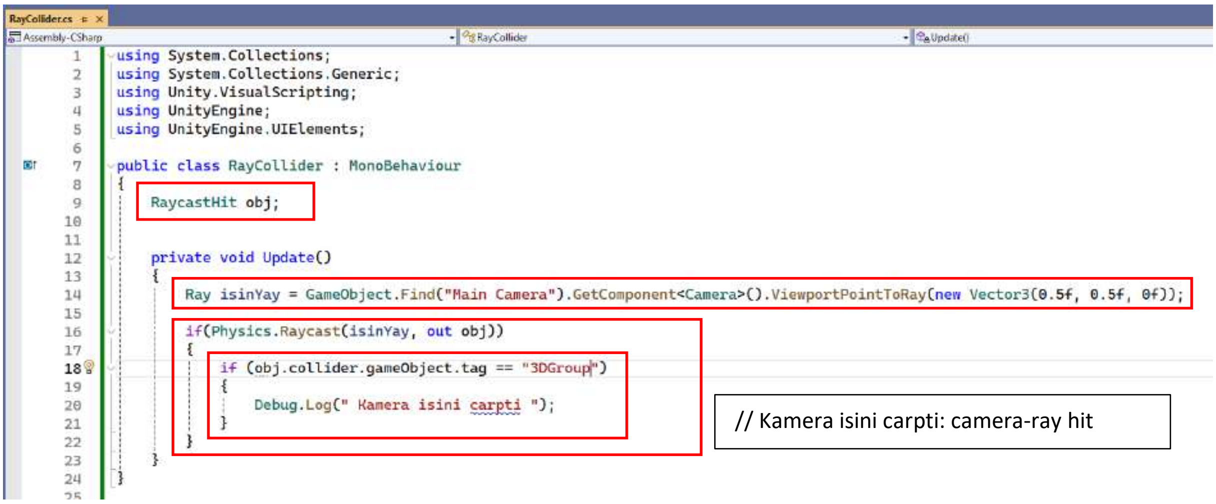
Ray isinYay = GameObject.Find("Main Camera").GetComponent<Camera>().ViewportPointToRay(new Vector3(0.5f,0.5f,0f));

In this section, which can take values between 0 and 1, we mean the **center** point of the **camera** with the coordinate (0.5f, 0.5f, 0). While this variable is taking a value, if the **physical ray** produced **hits** an

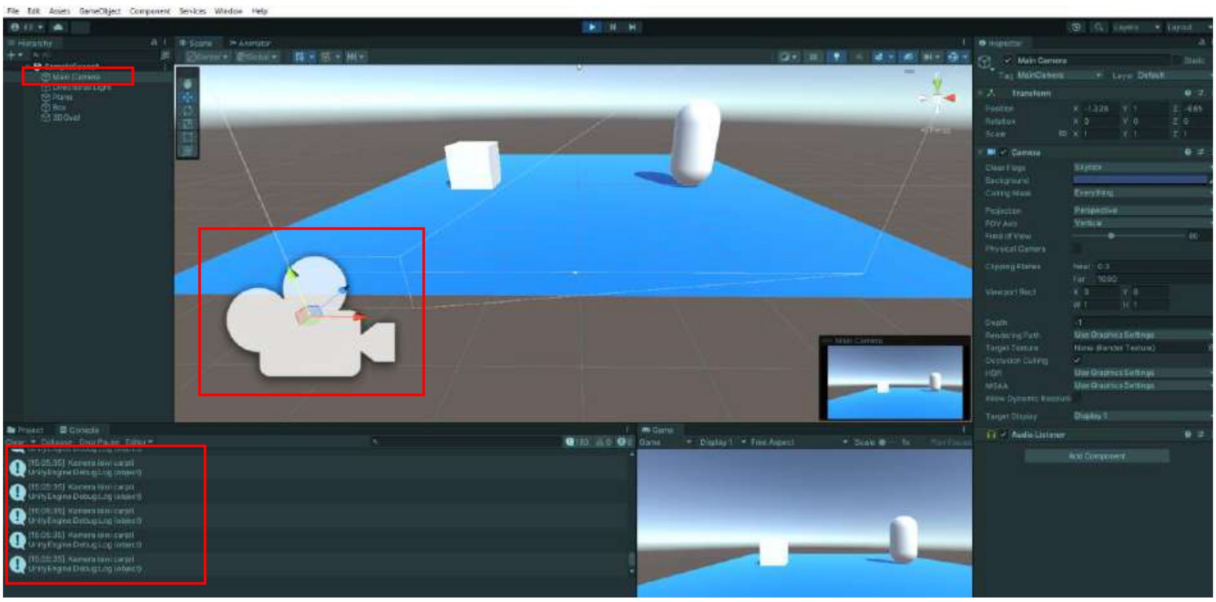
object, transfer its information to our **RastcastHit** type variable **obj**. The following code with a **bool** type **return** can be used to test this.

Physics.Raycast(isinYay, out obj)

To see if this **ray hits** the **object** defined by **tag** or **name**, send a message to the **console** with the **Debug.Log** command.



When we switch to **Play** Mode, this time we need to move the camera **left** and **right** since it is a **ray emitter**. When the **center** point of the camera (0.5f,0.5f,.0f) aligns with the objects whose **tag** information is **3dGroup**, that is, both the cube and the capsule, we can see the **Debug.Log** message reflected in the **console**.



Ray is frequently used in **FPS** (**first-person shooter**) type applications.

We have seen that **Ray** propagation from objects and cameras, and when it hits, it can give a message and change the **Mesh Renderer** settings of the object it hits or its properties.

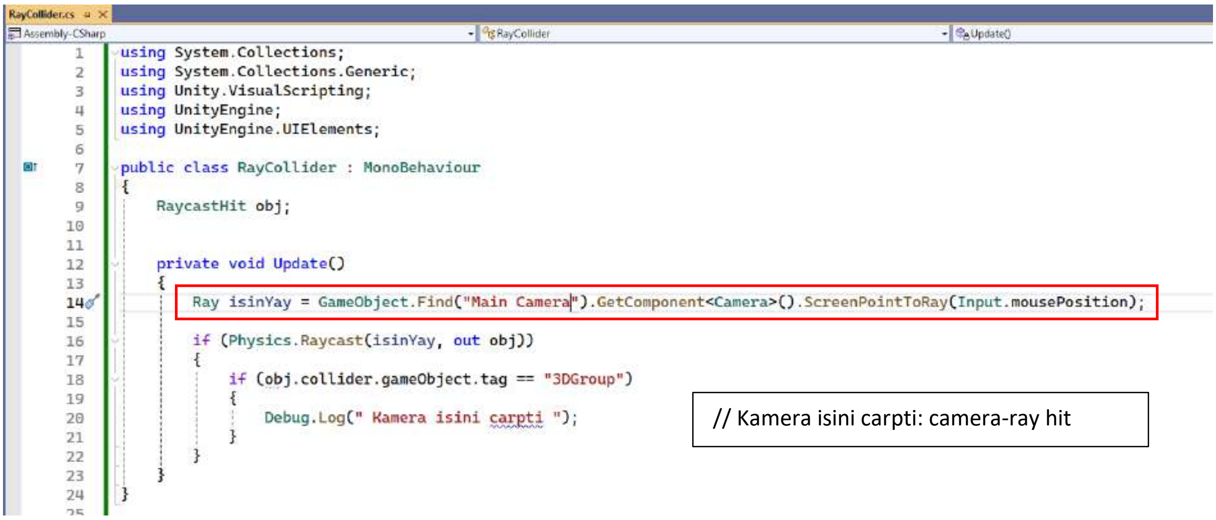
4.22.Ray Propagation from Mouse

Another type of **Ray propagation** is done via **Mouse**. The only **difference** from the previous camera application is the addition of the **ScreenPointToRay** function instead of the **ViewportPointToRay**. Here, using the **mouse position** is a common application, and this can be provided with **Input.mousePosition**.

The following command line can be written for the **Ray** type variable to get a value and emit a ray.

Ray isinYay = GameObject.Find("Main Camera").GetComponent<Camera>().ScreenPointToRay(Input.mousePosition);

Let's apply it to the code without making any other changes to the program.



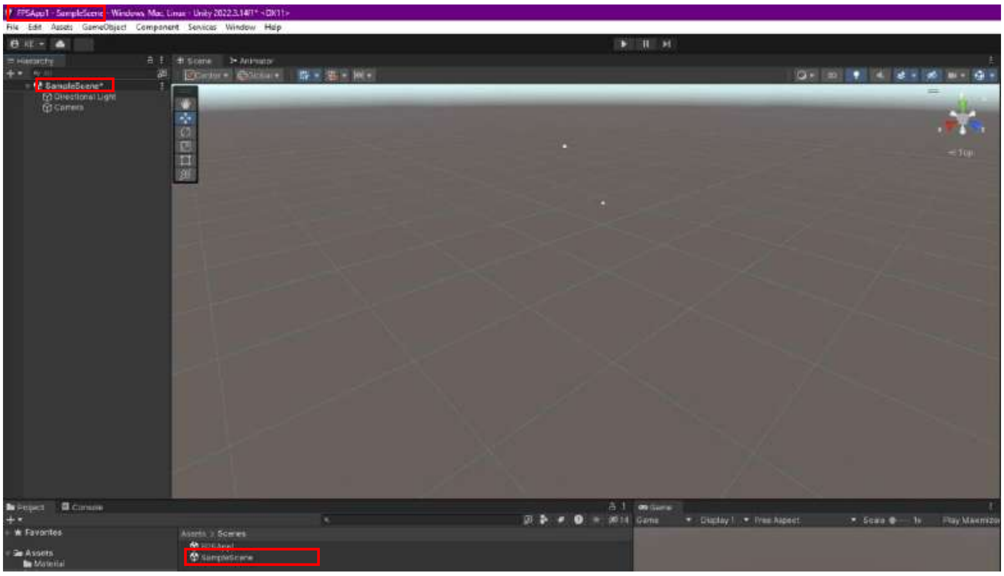
Debug.Log messages were successfully sent to the **console** when the **mouse** was **moved** and **hovered** over both **3DGroup** objects (**cube or capsule**). Messages were also stopped when the mouse was not over these two objects.

In **Physics.Raycast(isinYay, out obj)** expression, a **distance** definition can be added as **Physics.Raycast(isinYay, out obj,100f)**. When this is not the case, the ray length is **infinite**. It is preferable to set a distance in terms of performance and to maintain the computer processor. However, this distance should be determined with a couple of trials and its lower limit should be found.

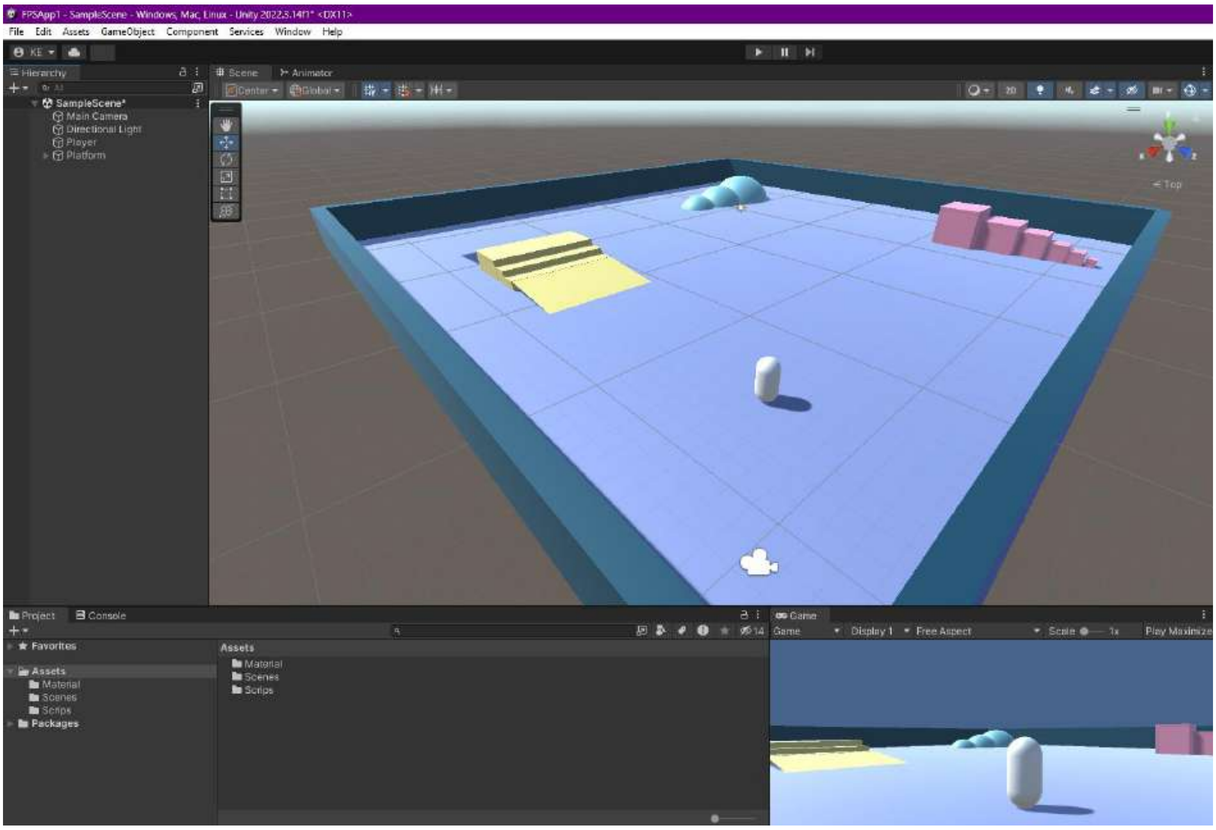
4.23.FPS/P (First Person Shooter/Perspective) Applications

One of the common applications is to move the scene from the eyes and perspective of the person moving in the **3D environment**. Depending on the scenario, various jobs and tasks can be performed during this movement. But first, this movement must be provided. There are **FPS assets** available for use by Unity Technologies or other manufacturers in the Asset Store. In this section, let's see how we can create our character within simple scene setups and provide movement with codes with a few alternative approaches.

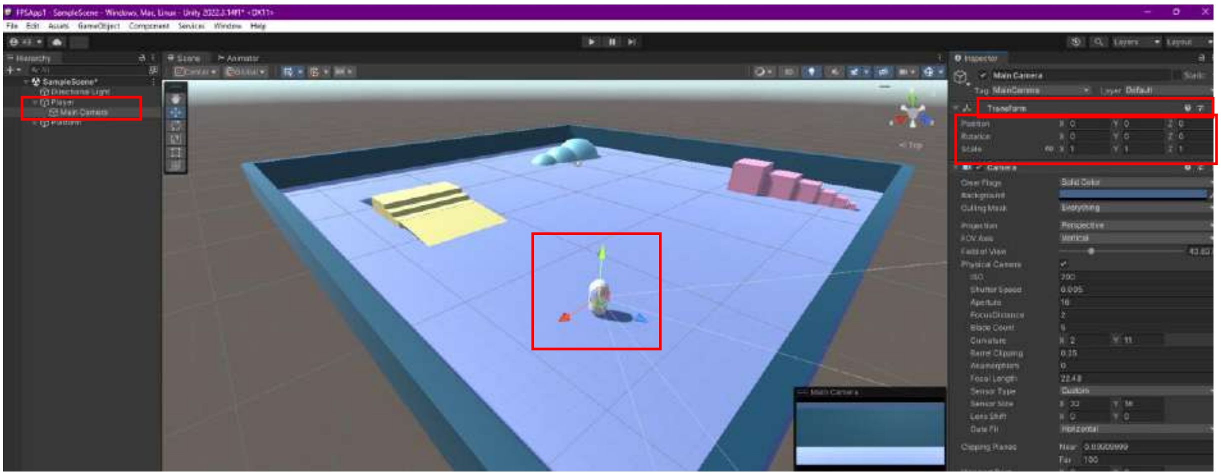
For the first approach, a project called **FPSApp1** was created for the application, the purpose of which is to control movements and camera with **keys** and a **mouse**.



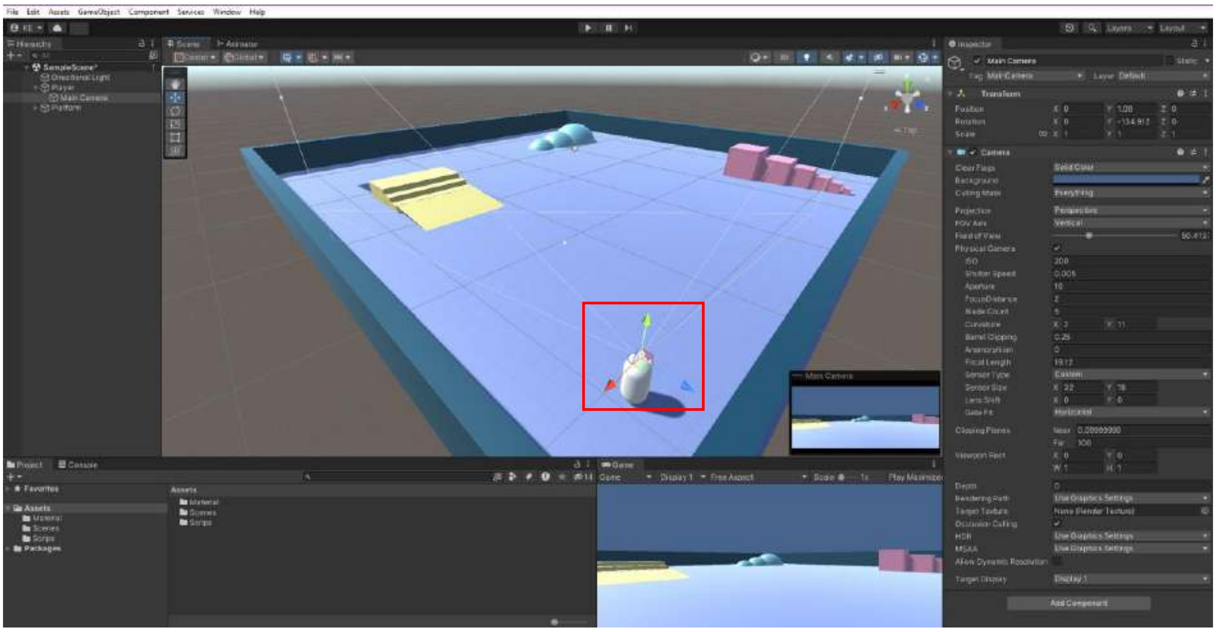
A platform for the application was established for the scene (**SampleScene**). A simple design was made for the platform with a **Plane, Cube**, and **Sphere**. A **Capsule** called **Player** was added.



Let's drag the **Main Camera** under the **Player** (**Capsule**) object and establish a **parent-child** relationship. Reset the camera **transform** properties so that it references the **capsule**.



Now, **position** the camera at the **top** of the **capsule**, which can be considered **head** level, where we loaded the **FPS** role. Adjust the angle of view.



After physically setting up the order, create and develop **two C# code files** required for the **capsule** (**Player**) in the **FPS** role and the **camera** representing our **eye** (*M.Akkuzu* approach was used here).

We created two files named **PlayerControl.cs** and **CameraControl.cs** in the **Assets>Scripts** folder.

Screen geometry, **trigonometric,** and **axial** coordinate calculation information is not included in the content of this document and can be found in various written and visual sources. Theoretical information is applied in this section, and short explanations are provided.

Now, we can write the codes in Visual Studio. Let's specify a **public float** type variable that will carry the speed of the **player** (**capsule**) and a variable for the **z-axis speed**.

Since the movements are continuous, we write the relevant codes in the **Update()** method.
First, transfer the information from the **Input.GetAxis("Horizontal")** and **Input.GetAxis("Vertical")** ready functions to the **float** type **x** and **y** variables.

```
float x = Input.GetAxis("Horizontal");
float y = Input.GetAxis("Vertical");
```

These functions also **automatically** provide the use of the **arrow-direction keys** and the **A, S, D, and W** keys. This will **transfer** the **coordinate information** required for movement in the **horizontal plane**. However, the **depth** dimension is not included in this scope. For this, it will be necessary to assign a key (**vertical**) to a **float**-type variable to **ascend** and **descend**. For example, the **Q** key can be used for ascending and the **E** key for descending.

```
if (Input.GetKey(KeyCode.Q)) dikey = 1;
if (Input.GetKey(KeyCode.E)) dikey = -1;
```

Now, we can calculate the three parameters we will need in the **displacement** command **transform.Translate()**. The **x, y** and **dikey** (vertical) parameters are multiplied by the **velocity** variable defined as a **float** and **Time.deltaTime** for the time standard.

x *= Time.deltaTime * hiz;
y *= Time.deltaTime * hiz;
dikey *= Time.deltaTime * hiz;

Now, we can combine these three parameters into the **transform.Translate()** construct.

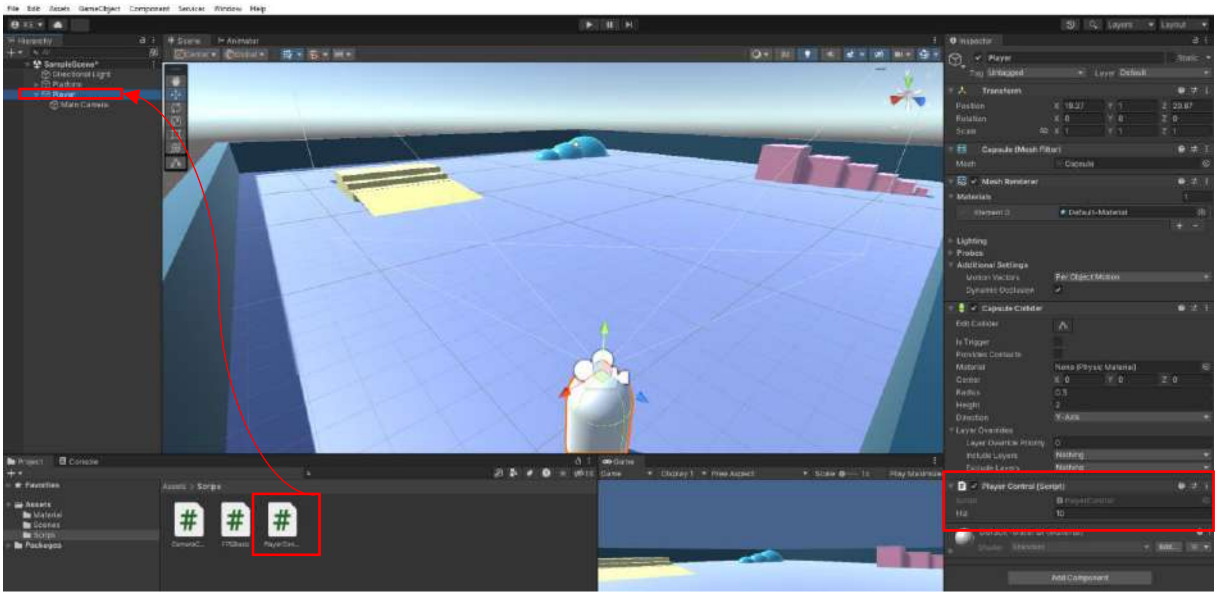
transform.Translate(x, dikey, y);

And coding in Visual Studio...

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

public class PlayerControl : MonoBehaviour
{

    public float hiz = 10;
    float dikey = 0;

    void Update()
    {
        float x = Input.GetAxis("Horizontal");
        float y = Input.GetAxis("Vertical");
        if (Input.GetKey(KeyCode.Q)) dikey = 1;
        if (Input.GetKey(KeyCode.E)) dikey = -1;
        x *= Time.deltaTime * hiz;
        y *= Time.deltaTime * hiz;
        dikey *= Time.deltaTime * hiz;
        transform.Translate(x, dikey, y);
    }
}
```

Then, drag and connect the **PlayerControl.cs** file to the **Player** object.

Do not forget that if the written codes are not connected to the relevant objects, they will **only** exist as a **non-functional** entity under Assets.



In case of adverse reaction to direction keys, **x** and **parameters** can be multiplied by **-**
transform.**Translate(-x, dikey, -y);**

Now, test in **Play** mode and see that movement is provided with **arrow** keys and **ASDW** keys.

Come to **camera** control and **CameraControl.cs** file.

Although we see three-dimensional movement on the screen, the third-dimensional coordinate of the objects is obtained by **perspective calculations** made on a **two-dimensional** plane. Therefore, both two-dimensional vector and three-dimensional vector calculations must be used. **Mathf.Lerp** and **Quaternion** functions will also be needed to avoid sharp movements and for **angle** calculations.

Use **float**-type variables for **precision** and smooth **transitions** in movements.

float hassas=5f;
float yumusak=2f;

We can also plan to use two **Vector2** to carry the camera and **new position information**, and a **GameObject** type variable for the **rotation transition** calculations.

Vector2 = newPos;
Vector2 = camPos;
GameObject player;

In the **Start()** method, assign the **first transform** value to the **game object** named **player**.

player=transform.parent.gameObject;

We can perform the other operations in the **Update()** method since there will be continuity.

Let's create a **Vector2** type **farePos** variable for the **mouse position** and assign the **X** and **Y** coordinates of the mouse with the **Input.GetAxis("X")** and **Input.GetAxis("Y")** commands. This process will be repeated 24 times per second or more.

```
Vector2 farePos = new Vector2(Input.GetAxis("Mouse X"), Input.GetAxis("Mouse Y"));
```

After **transferring** the mouse coordinates to this variable, a **new scale** calculation can be made using these coordinates and the sensitivity and smoothness variables.

```
farePos = Vector2.Scale(farePos, new Vector2(yumusak * hassas, yumusak * hassas));
```

Now, we calculate the axial coordinates of the mouse position changes. Assign values to the **X** and **Y** elements of the variable we defined above as **Vector2 newPos.** Do this with the real number math function **Mathf** and the **Lerp**, which smooths the transition with interpolation. Since the third parameter in the (**float a, float b, float t**) pattern of this function is the **float t** (**time**) type, we can specify it as **1f/yumusak** here.

```
newPos.x = Mathf.Lerp(newPos.x, farePos.x, 1f / yumusak);
newPos.y = Mathf.Lerp(newPos.y, farePos.y, 1f / yumusak);
```

With these two components, the new **camera position** can be calculated with a simple addition.

```
camPos+=newPos;
```

We can calculate the rotation locally with the command whose general form is **Quaternion.AngleAxis(float angle, Vector3 axis)**.

```
transform.localRotation= Quaternion.AngleAxis(-camPos.y, Vector3.right);
```

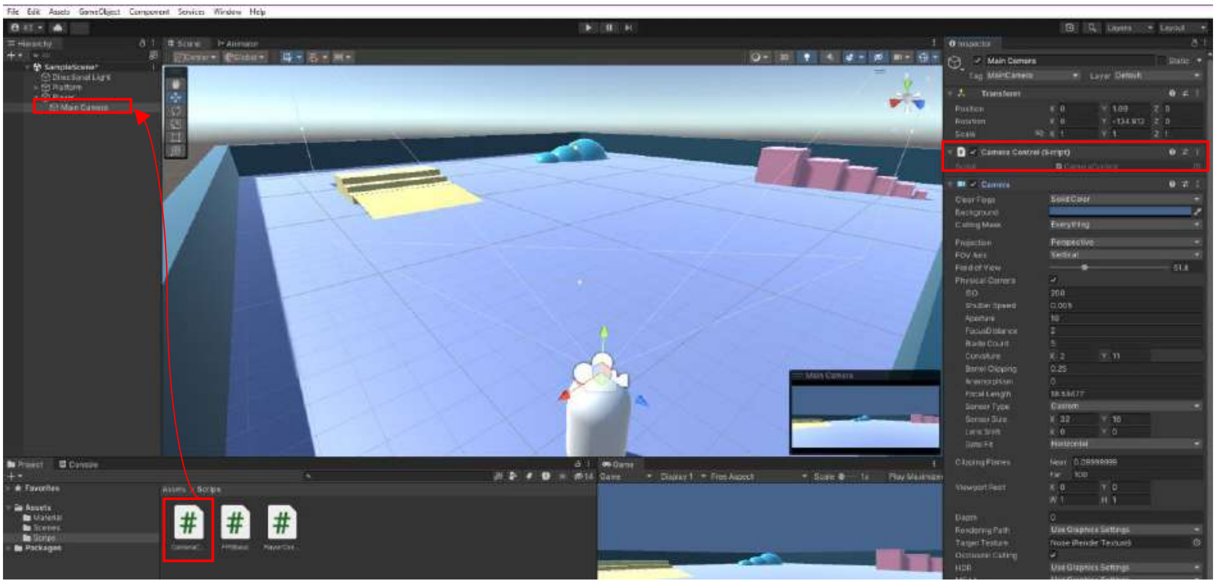
Here, the negative of the camera position is made for the **angle**, and the **right orientation** is made for the **3D vector** parameter. In some **local** and **global** coordinate differences, the keyboard and mouse can go in the opposite direction of the desired direction. In this case, a correlation should be made with **-/+** markings.
The last process is to calculate the game object **Player**.

```
player.transform.localRotation = Quaternion.AngleAxis(camPos.x, player.transform.up);
```

Here, the **local rotation angle** and **3D vector position** are calculated.

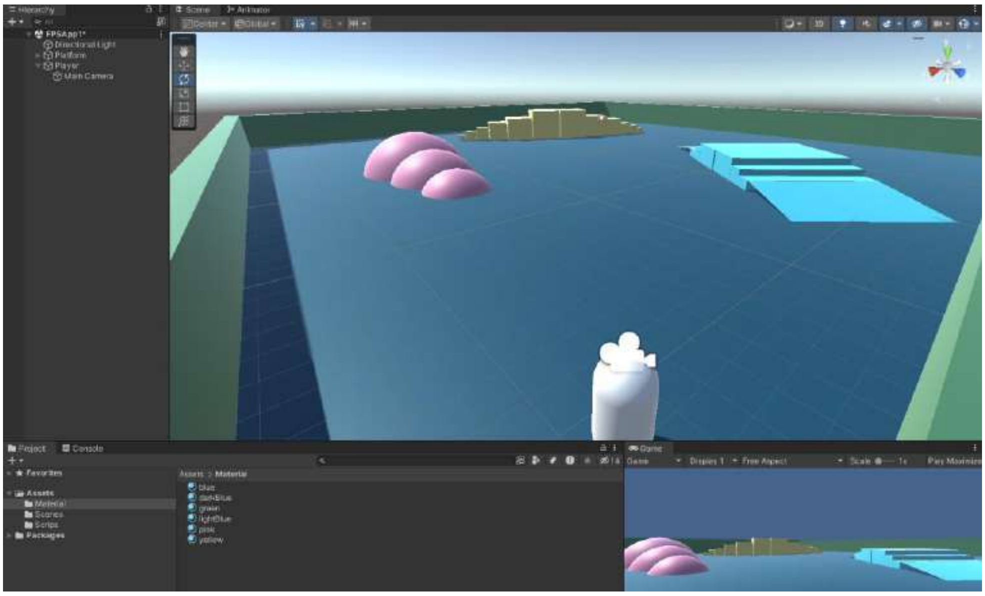
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CameraControl : MonoBehaviour
{
    private float hassas = 5f;
    private float yumusak = 2f;
    Vector2 newPos;
    Vector2 camPos;
    GameObject player;

    void Start()
    {
        player = transform.parent.gameObject;
    }


    void Update()
    {
        Vector2 farePos = new Vector2(Input.GetAxis("Mouse X"), Input.GetAxis("Mouse Y"));
        farePos = Vector2.Scale(farePos, new Vector2(yumusak * hassas, yumusak * hassas));

        newPos.x = Mathf.Lerp(newPos.x, farePos.x, 1f / yumusak);
        newPos.y = Mathf.Lerp(newPos.y, farePos.y, 1f / yumusak);
        camPos += newPos;
        transform.localRotation = Quaternion.AngleAxis(-camPos.y, Vector3.right);
        player.transform.localRotation = Quaternion.AngleAxis(camPos.x, player.transform.up);
    }
}
```

Now, drag and connect the script file to our camera on the scene and make the codes effective on this object. Run it in play mode.

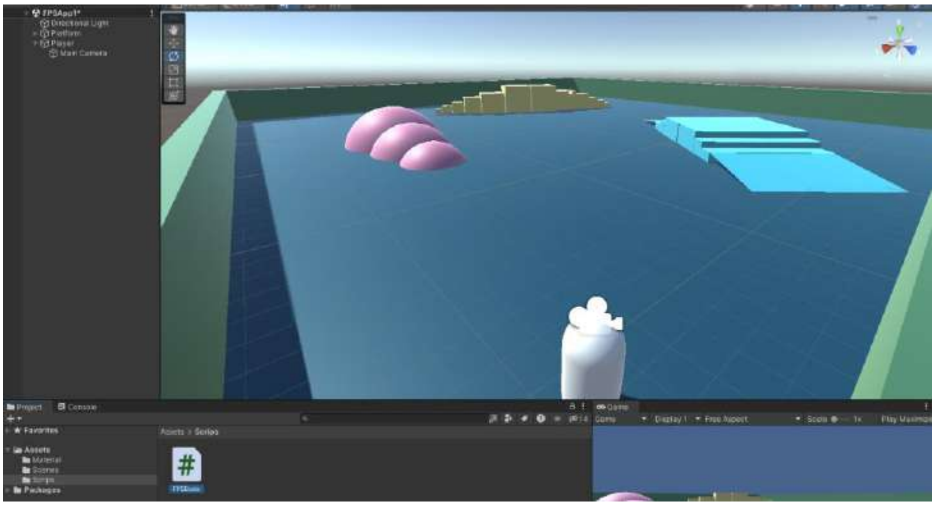


For a better application, the **Player** (**capsule**) **Mesh Renderer** check box can be unchecked to make the capsule **invisible**. In the camera's **Clipping Planes** settings, **Near** can be changed to 0.01 and **Far** to 1000 or greater.

For the **second approach**, let's create a new scene named **FPSApp1**. The scene design is similar to **SampleScene** with a small change; the object colors are different. In this application, unlike the previous one, the **FPS** body (**Player**) and camera controls will be combined in a C# file.



Now, create a file named **FPSBasic.cs** under the **Assets>Script** folder and open it in Visual Studio.



This scene utilizes the *User1 Productions (YouTube)* approach.

We can define **three public float-type** variables to control **walking**, **running** and **jumping** movements. Also, define a public **Transform** type variable for the camera transform reference and a **public float** type variable for the **camera control sensitivity**.

public float walkSpeed = 5f; // Speed of Walking
public float runSpeed = 10f; // Speed of running
public float jumpForce = 5f; // Force of jumping
public Transform cameraTransform; // Reference to the camera transform
public float mouseSensitivity = 2f; // Mouse sensitivity for camera control

Also, add a **Vector3** type variable for the **player speed** coordinates and a **bool** type variable for the **jump** control with the Unity variables **CharacterControl**.

```
private CharacterController controller; // character control variable

private Vector3 playerVelocity; // velocity vector
private bool isJumping; // jump - true or false
```

In the beginning, access the controller variable to the **CharacterController** component of the object that will be used in the character control (after connecting). Now, make sure that the **cursor position** is **centered** on the screen.

```
controller = GetComponent<CharacterController>();
Cursor.lockState = CursorLockMode.Locked;     // Lock the cursor to the center of the screen
```

If the continuity of the movement is considered, we should write the control codes of the **character** (**player**) and the **camera** to the **Update()** block.

Whether or not the running movement will occur can be linked to the **left shift key** being pressed. We can provide this information during **vectorial** movement by transferring the result to a **float**-type variable (*for example*, **moveSpeed**). While calculating **moveSpeed**, we can use the **?** condition **operator** means that if the left shift key is pressed, **walkSpeed** is **activated**; otherwise, **runSpeed** is activated. These variables were previously defined as **public float**. In addition, during the player movement, during **walking/running**, we can transfer the **horizontal** and **vertical axis** information to **float-type** variables with the **Input.GetAxis()** command.
The **axial data** and **speed data,** and the **3D vectorial information** of the control variable are transferred to the move variable.

```
float moveSpeed = Input.GetKey(KeyCode.LeftShift) ? runSpeed : walkSpeed;
float horizontal = Input.GetAxis("Horizontal");
float vertical = Input.GetAxis("Vertical");
Vector3 move = transform.right * horizontal + transform.forward * vertical;
controller.Move(move * moveSpeed * Time.deltaTime);
```

There are **ready-made functions** used for **CharacterController** type variables.

Let's set up an **if** conditional sentence to **reset** the **velocity** on the **y-axis** after the **jump** with the **isGrounded** function and to **stop** the **jump**.

```
if (controller.isGrounded)
{ playerVelocity.y = 0f;
isJumping = false;
}
```

When the jump is connected to a **key**, the **GetButtonDown** function connected to the data input command (**Input**) can be used. Here, the "**Jump**" keyword can be used, and **isJumping** is connected to the sentence by negating it. When the conditions are **true**, a formula is applied that adds the square root (**Sqrt**) of the jumping power and physical gravity to the **player** speed (**playerVelocity**). **isJumping** is set to **true**.

```
if (Input.GetButtonDown("Jump") && !isJumping)
{   playerVelocity.y += Mathf.Sqrt(jumpForce * -2f * Physics.gravity.y);
    isJumping = true;
}
```

Since the **y-axis** is **vertical/depth**, the physics **gravity** and the **time** regulator **Time.DeltaTime** are multiplied by the **playerVelocity.y** axis to control the **jump** with gravity. Then, the control is reflected in the movement of the variable with the **controller.Move** command.

```
playerVelocity.y += Physics.gravity.y * Time.deltaTime;
controller.Move(playerVelocity * Time.deltaTime);
```

After player control, let's come to camera control. Here, **two** separate calculations are needed **horizontally** and **vertically**.

Data assignment can be made with two **ready-made function templates Input.GetAxis("Mouse X")** and **Input.GetAxis("Mouse Y")** for two **float**-type variables. In order to prevent **sharp** movement, we multiply it with the **mouseSensitivity** we defined earlier. We use these two-coordinate data in three-dimensional **vector** transformation in **rotation**. Here, **Vector3.up** is a useful **ready-made function**.

```
float mouseX = Input.GetAxis("Mouse X") * mouseSensitivity;
 float mouseY = Input.GetAxis("Mouse Y") * mouseSensitivity;
 transform.Rotate(Vector3.up * mouseX);
```

For the camera's **vertical movement**, first, the current **angular** information is transferred to the **3D** vector variable, and then the desired new **angle** is calculated by taking the difference between the **x-axis** component of this information and the **y-axis** value of the mouse. If this **x-axis** value is more than 180 degrees, 360 degrees are **subtracted** from it. Otherwise, the **x-axis** value is calculated directly with the **Matf.Clamp** command as a **3D vector**.

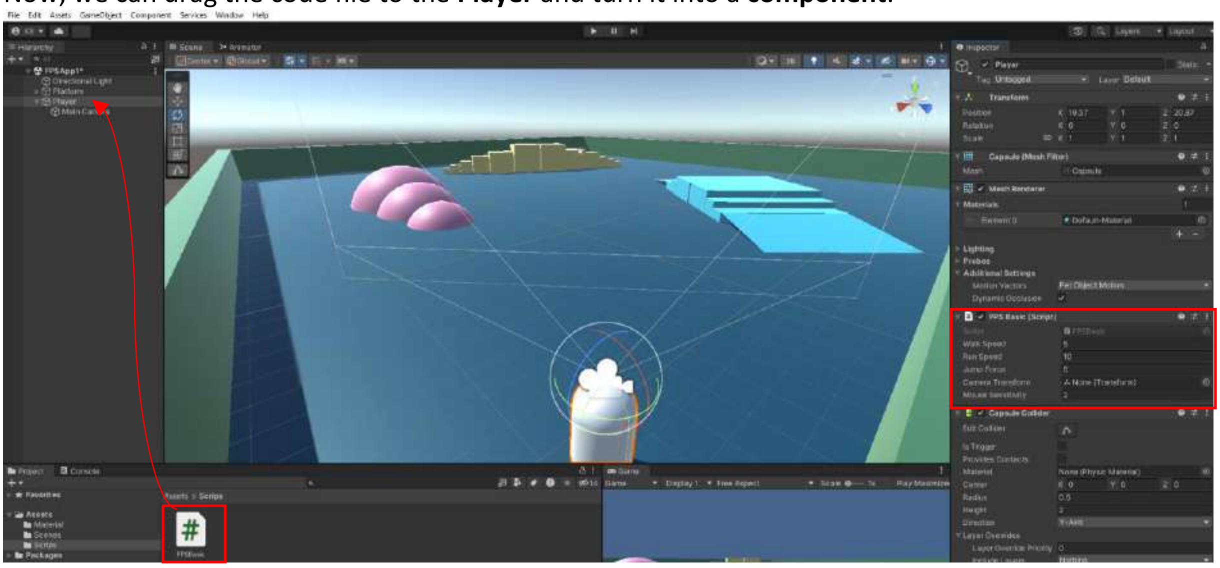
Camera rotation angles are also calculated with the **Quaternion.Euler()** command.

```
Vector3 currentRotation = cameraTransform.rotation.eulerAngles;
float desiredRotationX = currentRotation.x - mouseY;
if (desiredRotationX > 180)  desiredRotationX -= 360;
desiredRotationX = Mathf.Clamp(desiredRotationX, -90f, 90f);
cameraTransform.rotation = Quaternion.Euler(desiredRotationX, currentRotation.y, currentRotation.z);
```

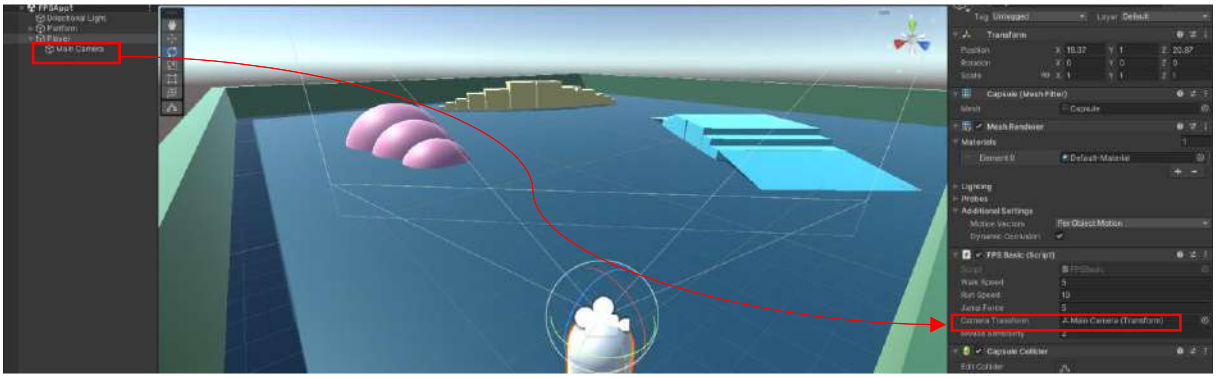
This step-by-step process sequence and its explanations are coded in **Visual Studio**.

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

public class FPSBasic : MonoBehaviour
{
    public float walkSpeed = 5f;         // Speed of walking
    public float runSpeed = 10f;         // Speed of running
    public float jumpForce = 5f;         // Force of jumping
    public Transform cameraTransform;    // Reference to the camera transform
    public float mouseSensitivity = 2f;  // Mouse sensitivity for camera control

    private CharacterController controller;
    private Vector3 playerVelocity;
    private bool isJumping;

    void Start()
    {
        controller = GetComponent<CharacterController>();
        Cursor.lockState = CursorLockMode.Locked; // Lock the cursor to the center of the screen
    }

    void Update()
    {
        // Player movement
        float moveSpeed = Input.GetKey(KeyCode.LeftShift) ? runSpeed : walkSpeed;
        float horizontal = Input.GetAxis("Horizontal");
        float vertical = Input.GetAxis("Vertical");
        Vector3 move = transform.right * horizontal + transform.forward * vertical;
        controller.Move(move * moveSpeed * Time.deltaTime);

        // Player jumping
        if (controller.isGrounded)
        {
            playerVelocity.y = 0f;
            isJumping = false;
        }

        if (Input.GetButtonDown("Jump") && !isJumping)
        {
            playerVelocity.y += Mathf.Sqrt(jumpForce * -2f * Physics.gravity.y);
            isJumping = true;
        }

        // Apply gravity to the player
        playerVelocity.y += Physics.gravity.y * Time.deltaTime;
        controller.Move(playerVelocity * Time.deltaTime);

        // Player camera control
        float mouseX = Input.GetAxis("Mouse X") * mouseSensitivity;
        float mouseY = Input.GetAxis("Mouse Y") * mouseSensitivity;
        transform.Rotate(Vector3.up * mouseX);

        // Rotate the camera vertically
        Vector3 currentRotation = cameraTransform.rotation.eulerAngles;
        float desiredRotationX = currentRotation.x - mouseY;
        if (desiredRotationX > 180) desiredRotationX -= 360;
        desiredRotationX = Mathf.Clamp(desiredRotationX, -90f, 90f);
        cameraTransform.rotation = Quaternion.Euler(desiredRotationX, currentRotation.y, currentRotation.z);
    }
}
```

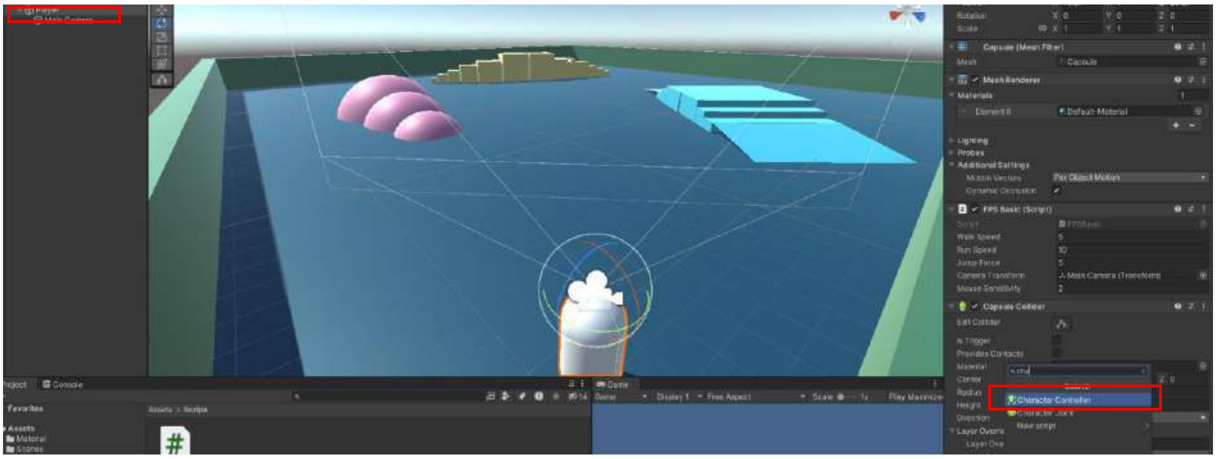
Now, we can drag the code file to the **Player** and turn it into a **component**.



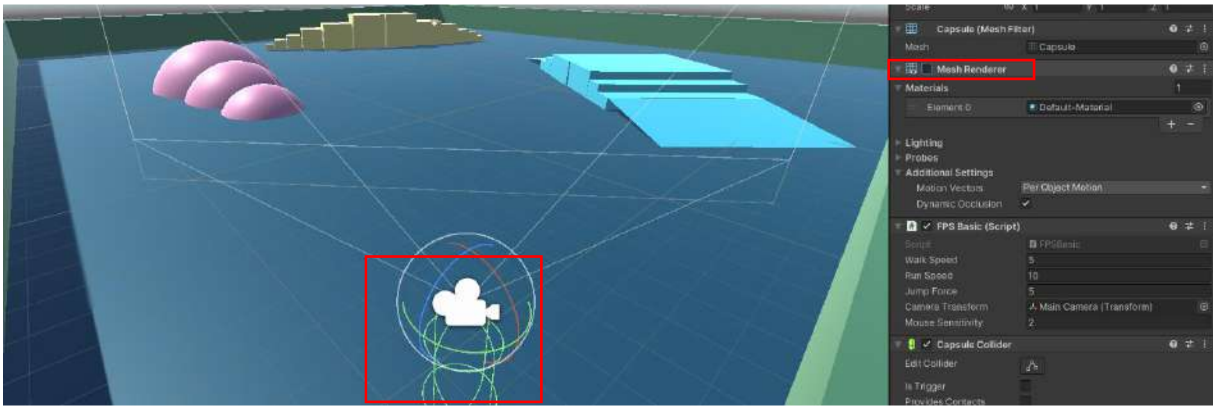
The **Camera Transform** section of the **FPSBasic** script is **empty,** and the camera to be controlled needs to be **connected** here. Drag the **Main Camera** here and connect it.



Another issue is the addition of the **CharacterControl** component defined and used in the codes to the **Player** (**capsule**) object.



If we turn off the **Mesh Renderer** feature of the **capsule**, we will not see the capsule when looking **down,** and its shadow will not appear.

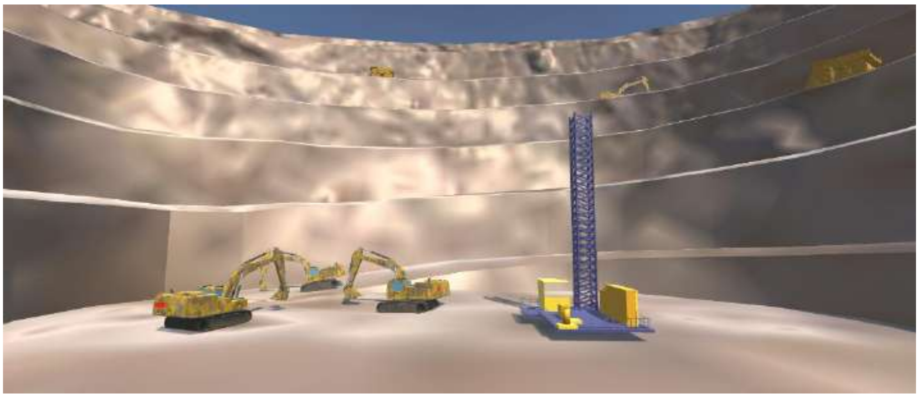


Experience that we move with the **arrow keys** in **play** mode, that we can **look** in every direction with the mouse, and that we can determine the **direction** of movement. While moving, the **left shift key** simultaneously **switches** to **running** mode, and when the **space bar** is pressed simultaneously, the **player** will be seen to **jump**.

As can be seen, we have made coding that is different from the previous approach but ultimately works at a similar level.

Note: The codes can be used on our projects one-on-one using various written and visual sources. In some sources, the codes are provided ready-made. It is useful to use them **even without** understanding them in the beginning. Using **ready-made codes** is a **natural** part of application development. As a result of doing and writing many applications with coding, the developer can start to make his comments. This requires **time** and **effort**.

Similarly, locomotion in a mine area is given below.

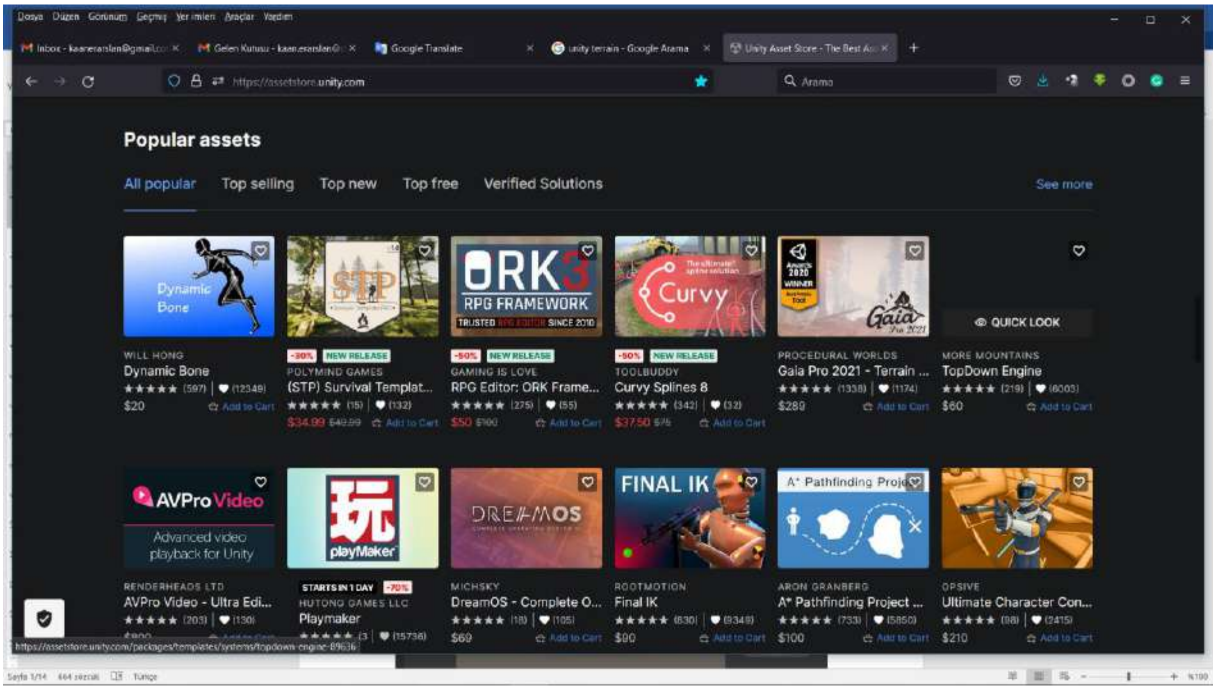


5. ADDING SCENE OBJECTS AND ASSET RESOURCES

Before we continue adding **GameObjects** to our scene, it is useful to give brief information about a few websites: **Unity Asset Store**, **Sketchfab,** and **others** such as **Rigmodels**, **GrabCad**, and **3D Warehouse**.

5.1.Unity Asset Store

When you enter the address **assetstore.unity.com** and examine it, a site awaits us with thousands of free and paid 2D and 3D objects, packages, ready-made games, etc., that we can add to our scenes. With this content, which makes many of our jobs easier, avoiding having to do everything from scratch and not having to write long codes is possible.



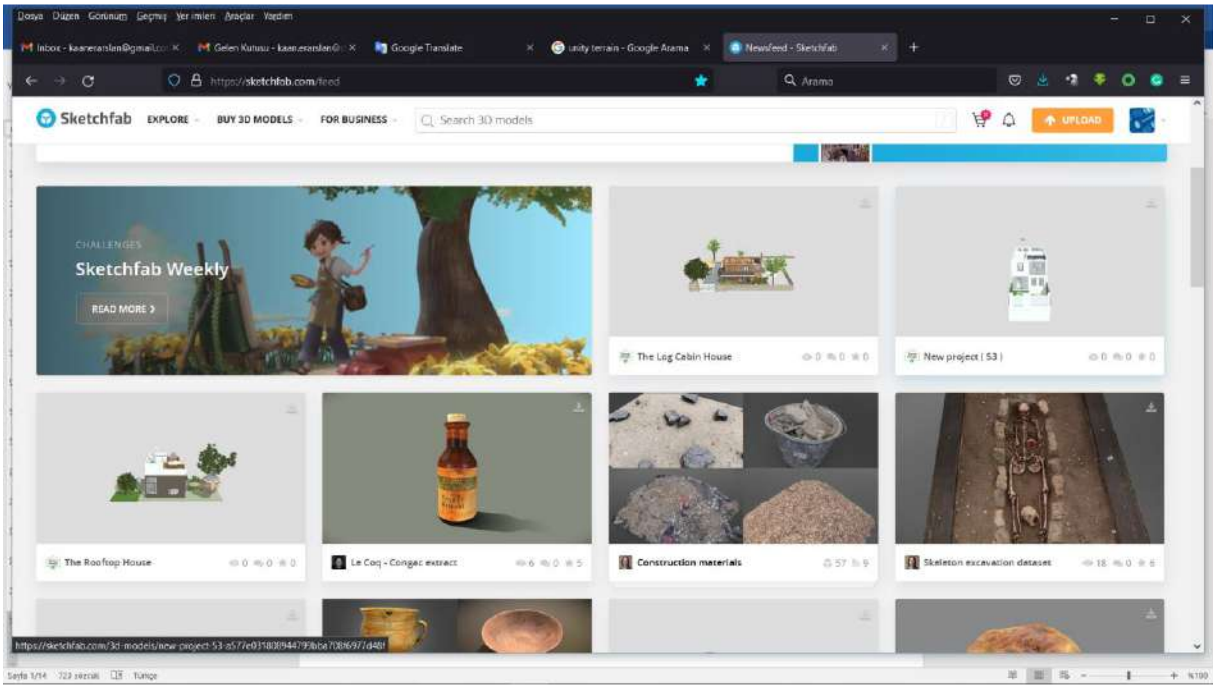
Here, by opening the account we opened on **Unity Hub**, we will be directly connected to our project. By accessing this site from within the project, we will be able to add assets to our Package Manager.

5.2.Sketchfab

Another external source is the website called **Sketchfab.com**. There are thousands of ready-made objects, animated designs, and stage materials that belong to many different professions and disciplines. We can add files to our scene by opening an account and downloading/adding them to our Asset window/folder.

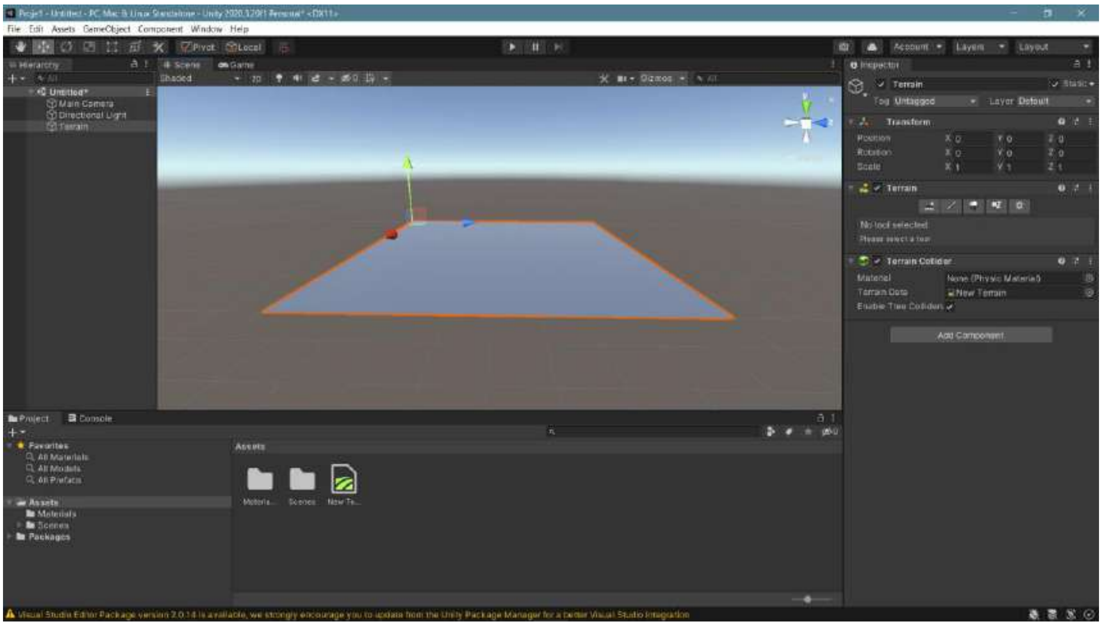
5.3.GrabCAD, Rigmodels and 3DWarehouse-Sketchup

GrabCAD.com, **Rigmodels.com**, and **3DWarehouse.com** (**Sketchup**) are other very important resources and websites that provide a huge asset library that is mostly free.

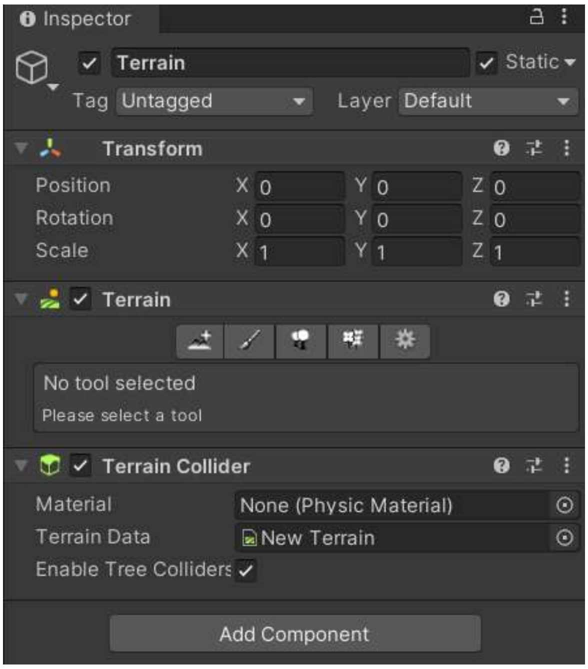


5.4.Terrain

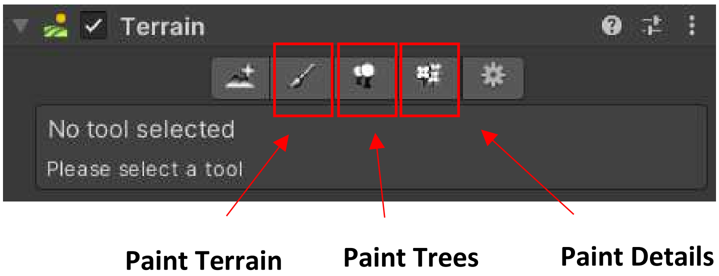
Unity's tool for creating impressive terrain design quickly is **Terrain**. Adding **Terrain** to the **Hierarchy** creates a flat area on the stage that is dozens of times larger than a normal plane.



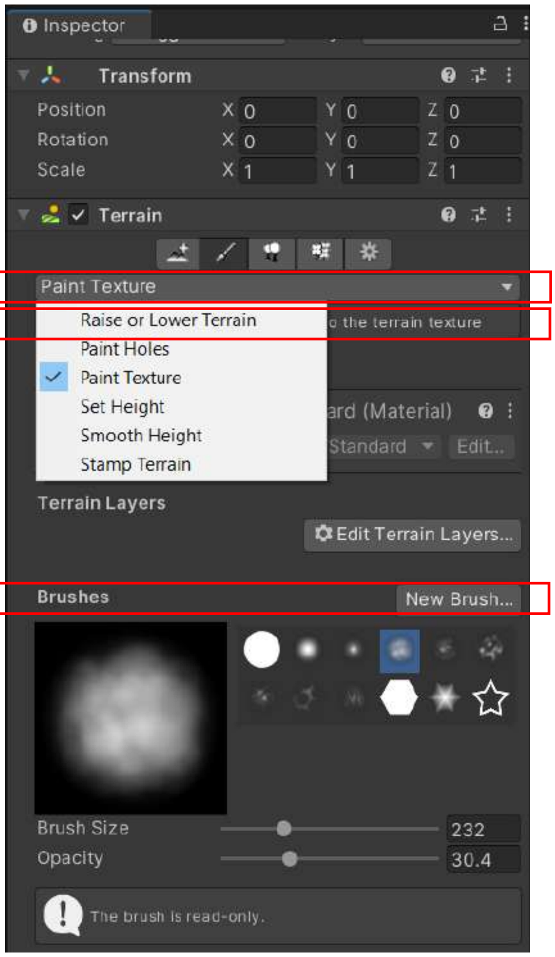
Terrain and **Terrain Collider** will be seen when examining Terrain's **Inspector**, apart from the **Transform** features.



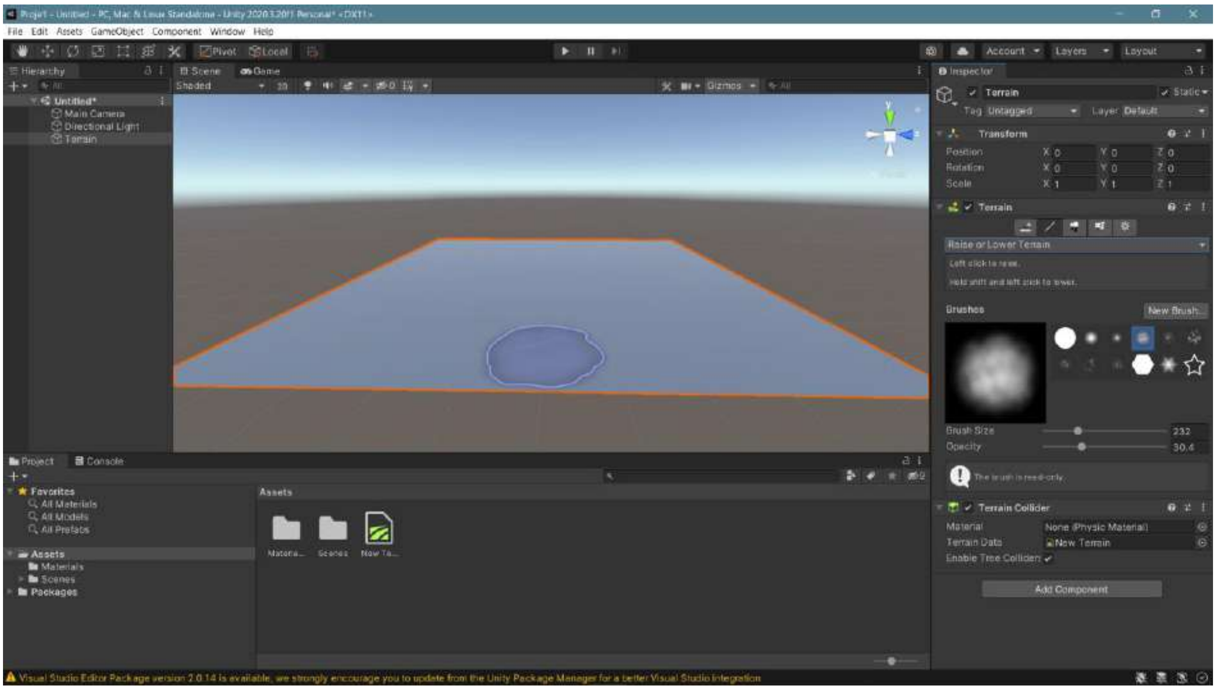
The most important functions for Terrain: **Paint Terrain, Paint Trees** and **Paint Details**.

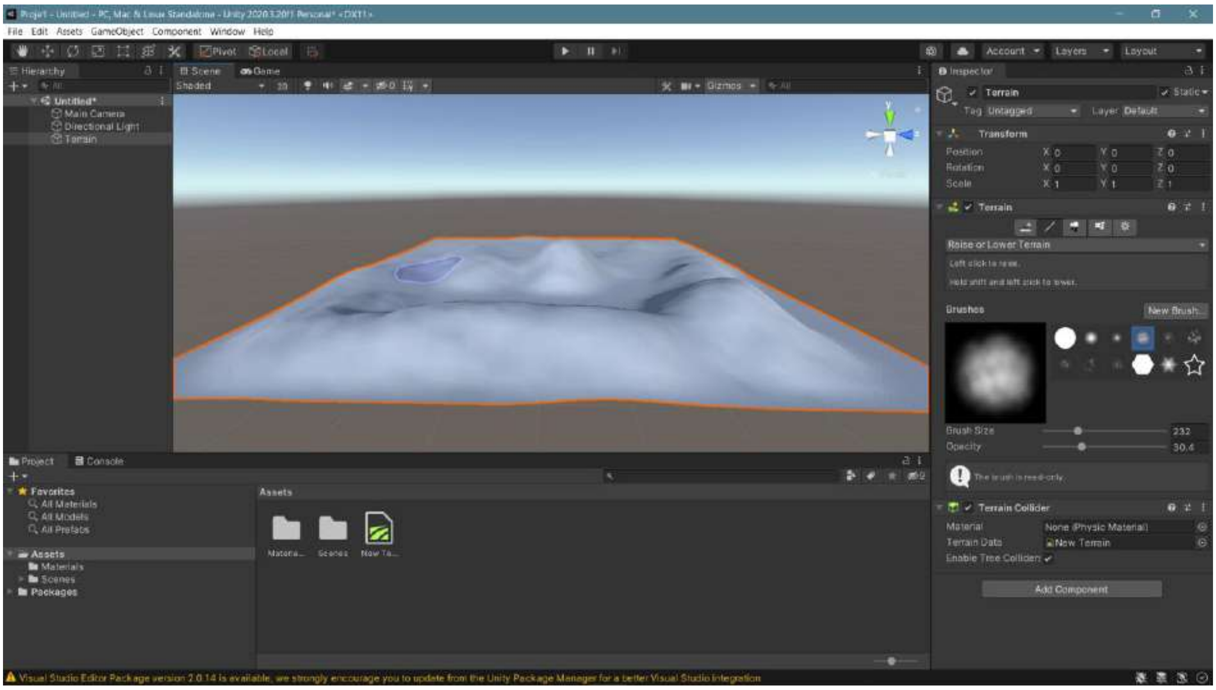


Paint Terrain is used to create **height** and **depth** in the field. When this feature is selected and the related menu is opened, its sub-parameters appear. **Raise or Lower** must be selected to obtain height. The same selection is made by lowering the ground with the **Shift** key. Another selection to be made is to select a brush head from the **Brushes** section. When our mouse is moved on the Terrain, the ground will rise or fall according to this pattern.



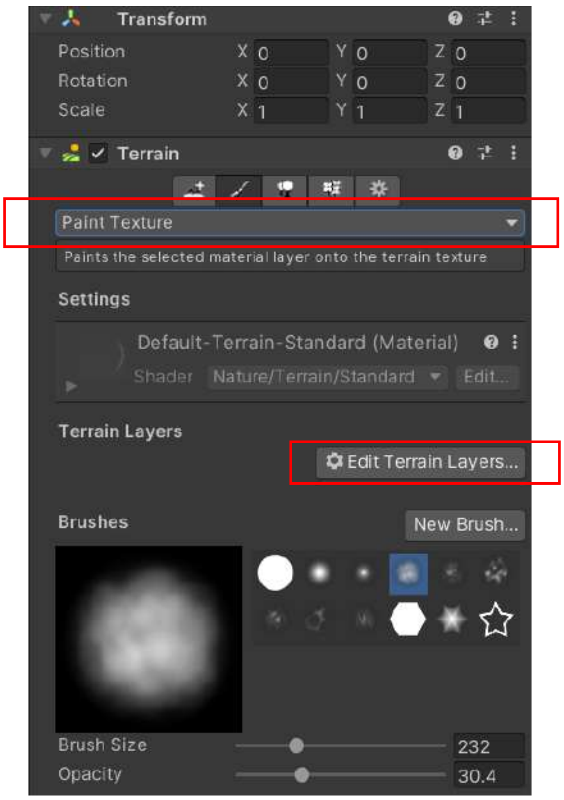
In the following figures, the shape left by the brush on the field and the heights created by moving the mouse are seen.

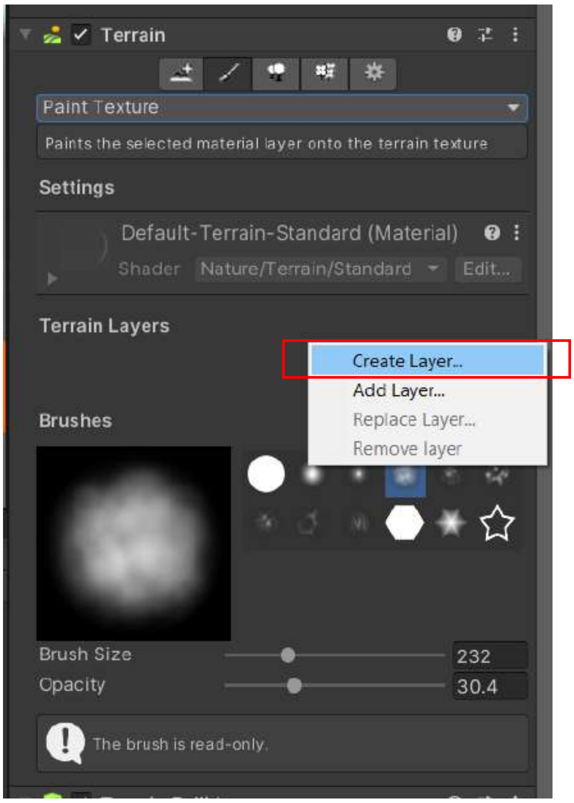


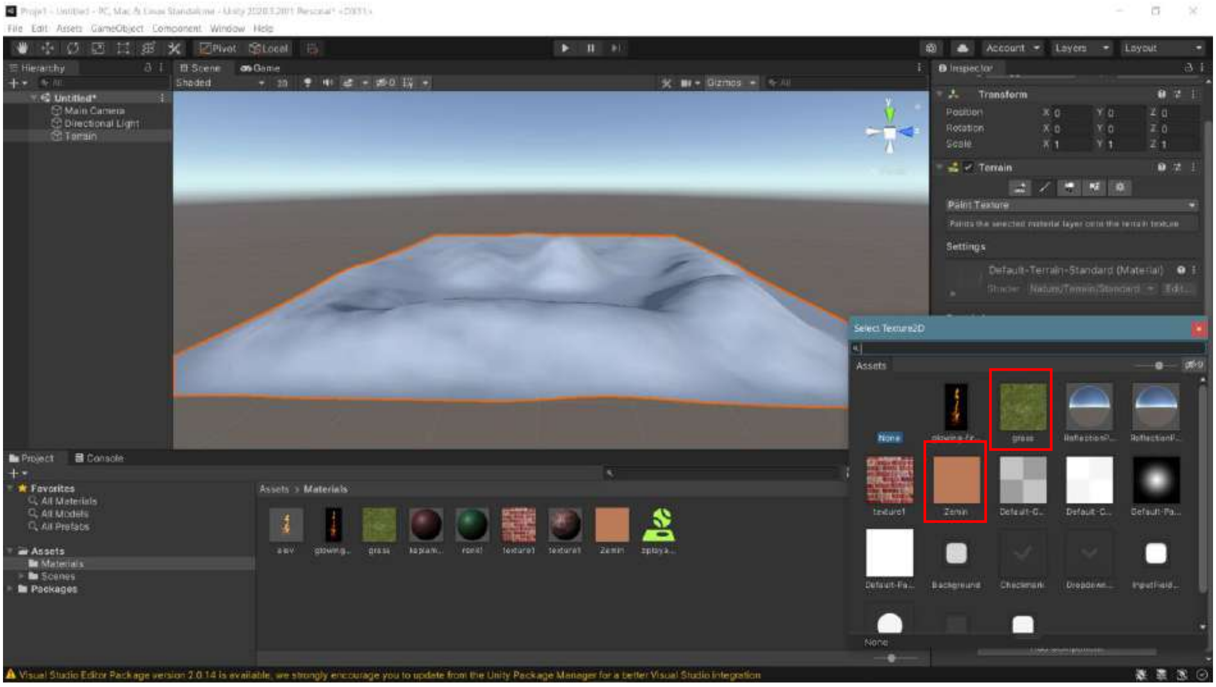


As you can see, a terrain topography was quickly obtained. The brush selected in the **Inspector** also has settings for brush size and solidity-effect degree, such as **Brush Size** and **Opacity**. The effects of changes in these can be observed when creating a field.

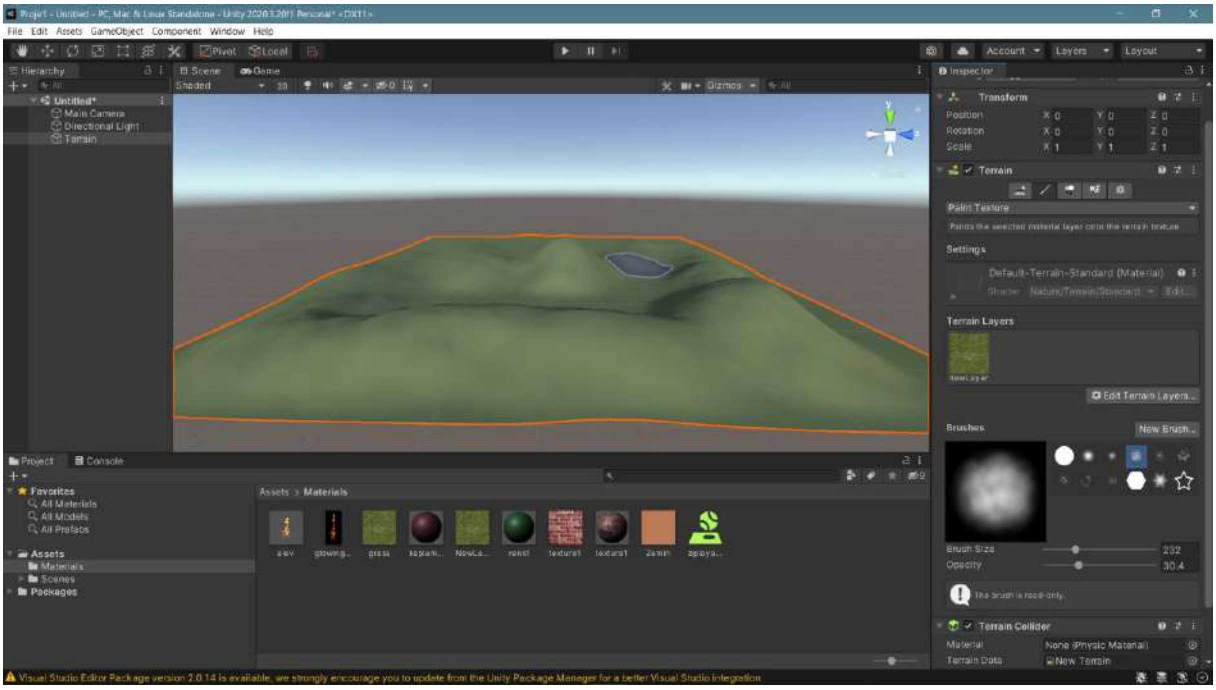
To add texture to the field, select **Paint Texture** from the list under **Paint Terrain**. Under **Paint Texture**, click **Edit Terrain Layers-> Create Layer** to open a selection window for a new texture layer.



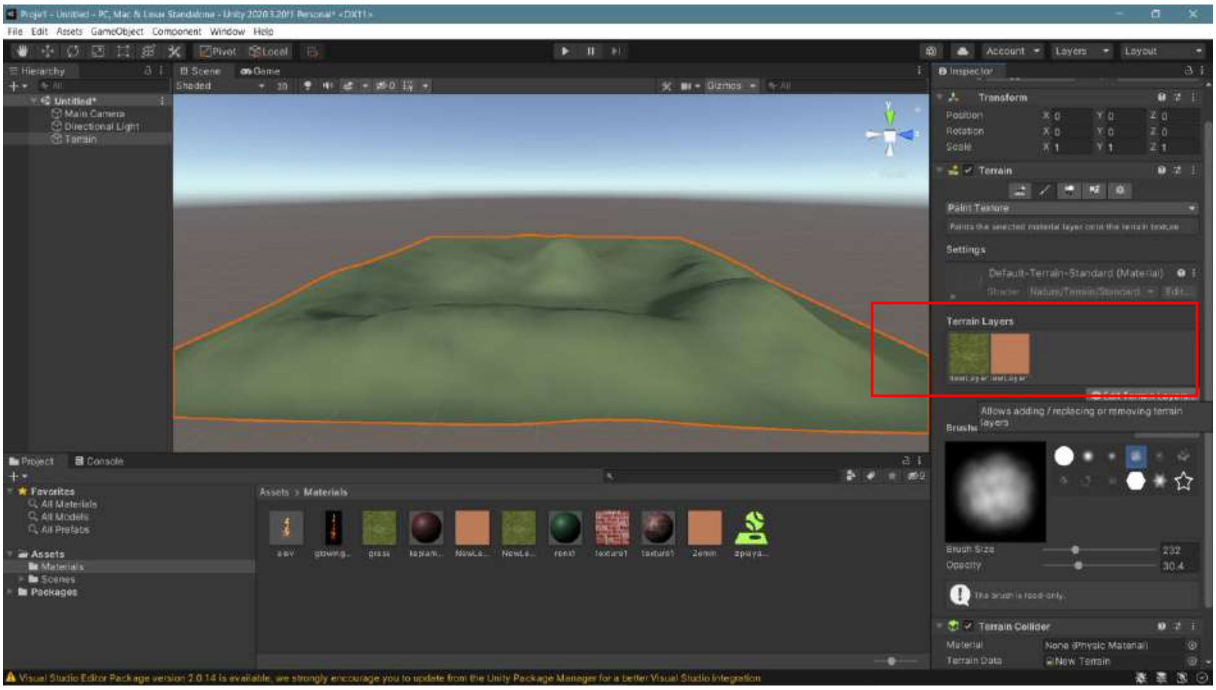


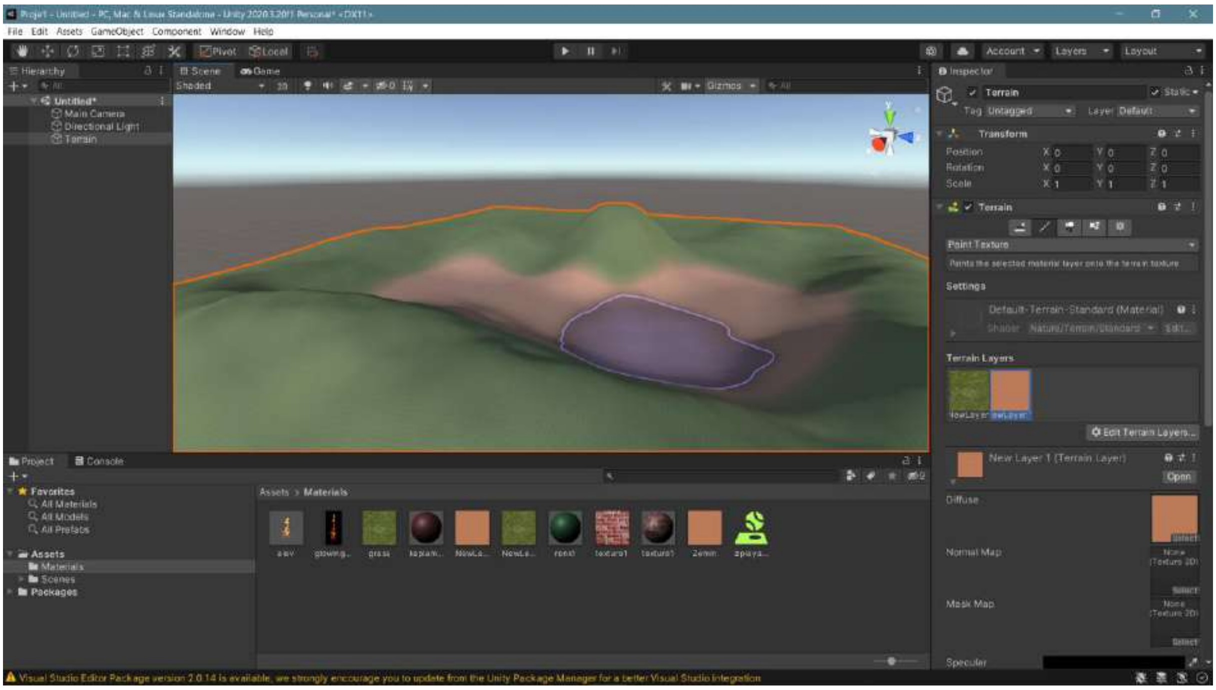


This window shows the files that can be used under the project. In the example, two JPEG files have been added to be used on the ground. One is a plain soil color made in **Paint Brush**, the other is a file found by searching the internet for "**ground texture**" or "**grass texture**". When the "**Grass**" texture is selected, the field is automatically covered with this texture because it is the first and only selection.

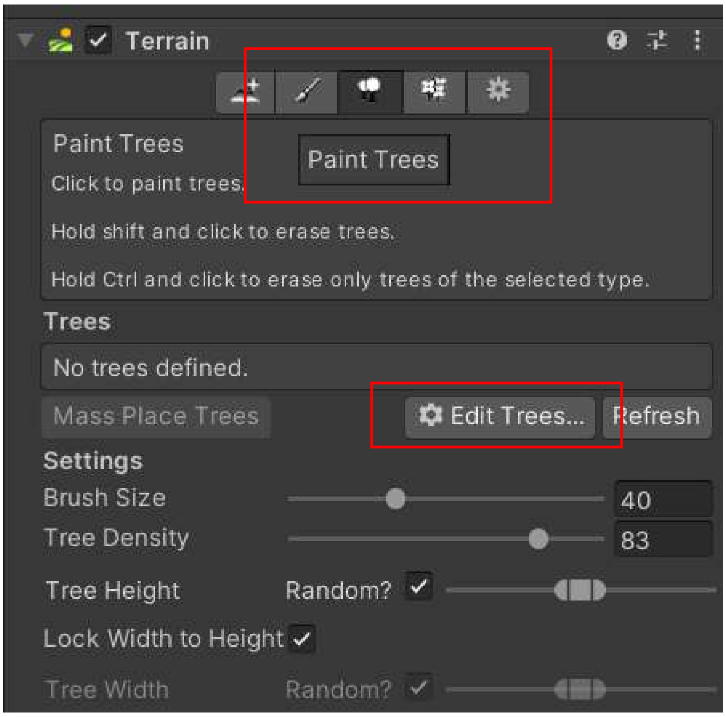


Once again, **Edit Terrain Layer > Create Layer** ground selection will appear in the **Terrain Layers** section; the second texture will appear. After that, the desired areas will be painted with the ground color with the brush.

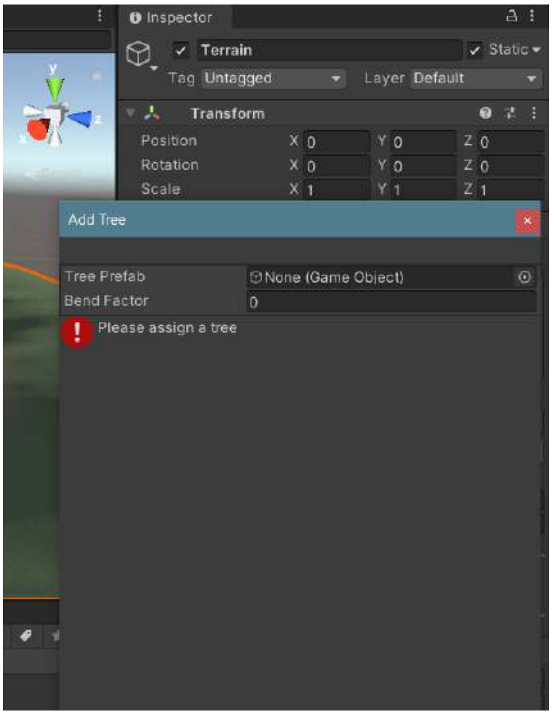


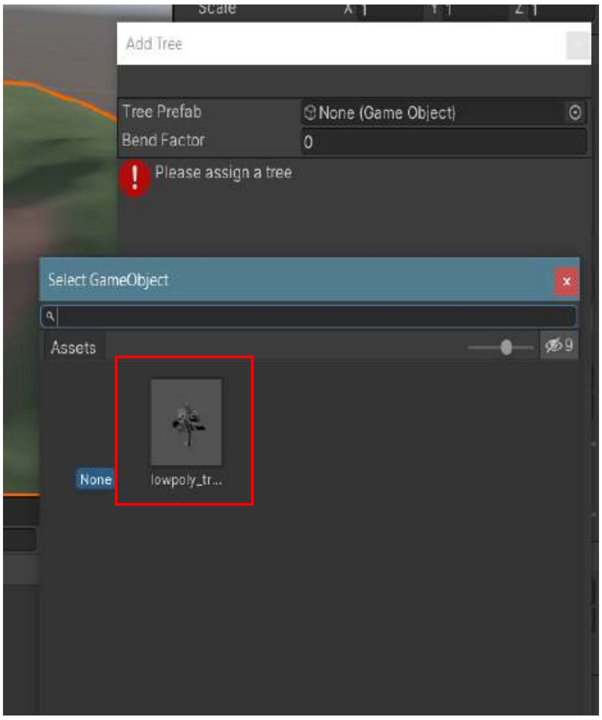


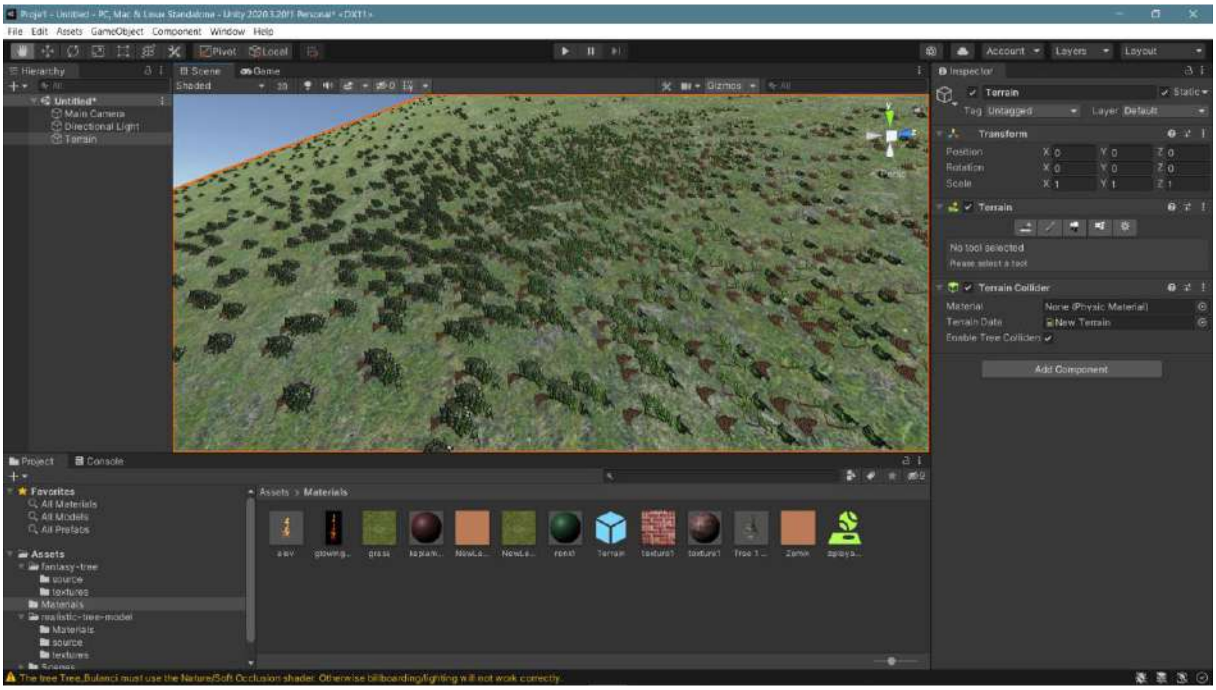
To **add trees** to the field, we first selected a low poly model from the **Asset Store** or **Sketchfab.com** and added it to our project. Now, from the **Terrain** menu, **Paint Trees** and **Edit Trees** can be selected.



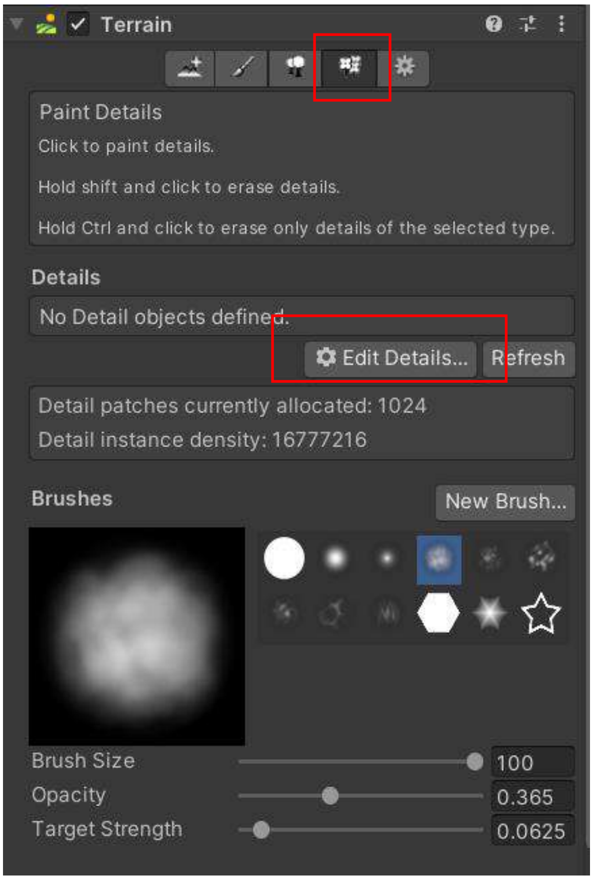
We add the tree we took to our project from the window that opens with the consecutive **Edit Trees >Add Tree > No (Game Object) > Select GameObject > lowpoly_Tree > Add** selections. In the areas where we move our brush, trees appear in a short time and a forest is formed.





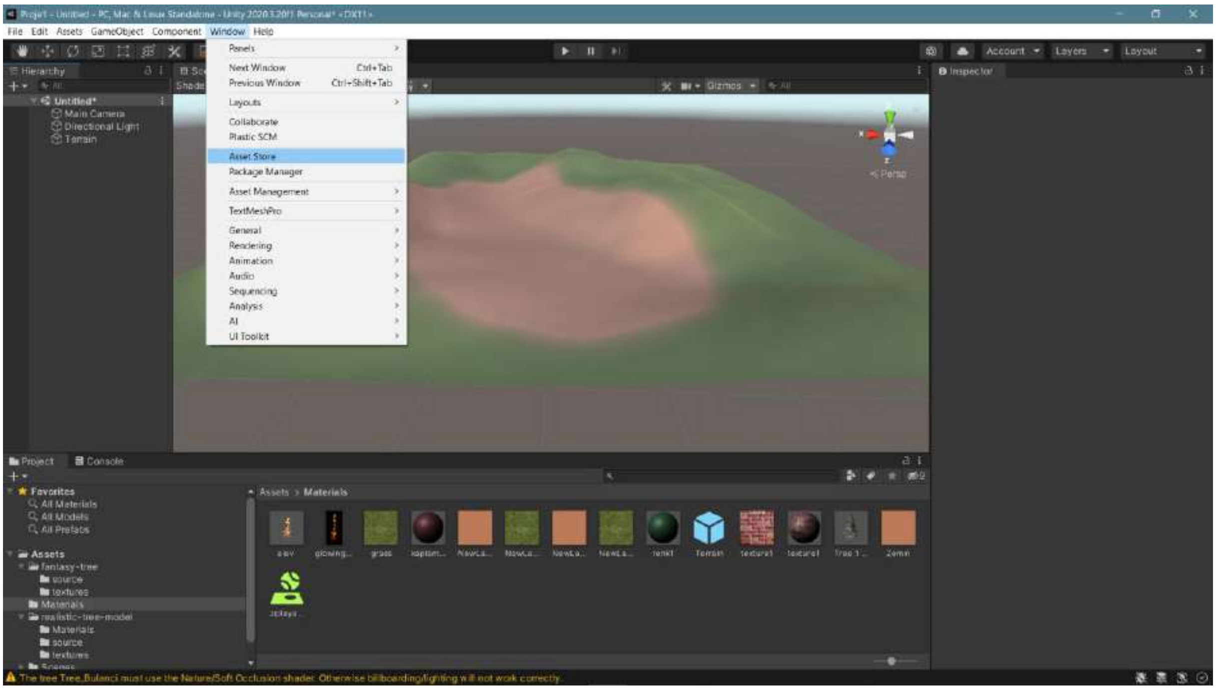


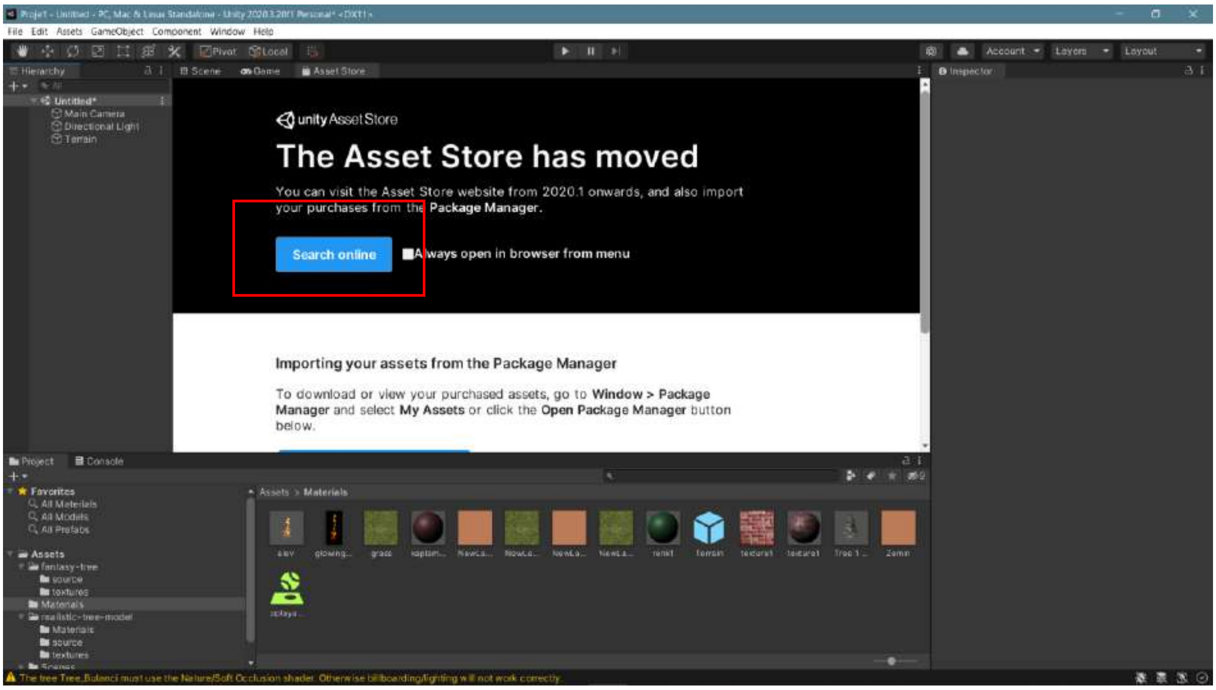
It is possible to create a real nature image on the field by doing the same process using grass or flower models with **Terrain > Paint Detail > Edit Detail > Add Grass Texture** file selection and then, **Add**.



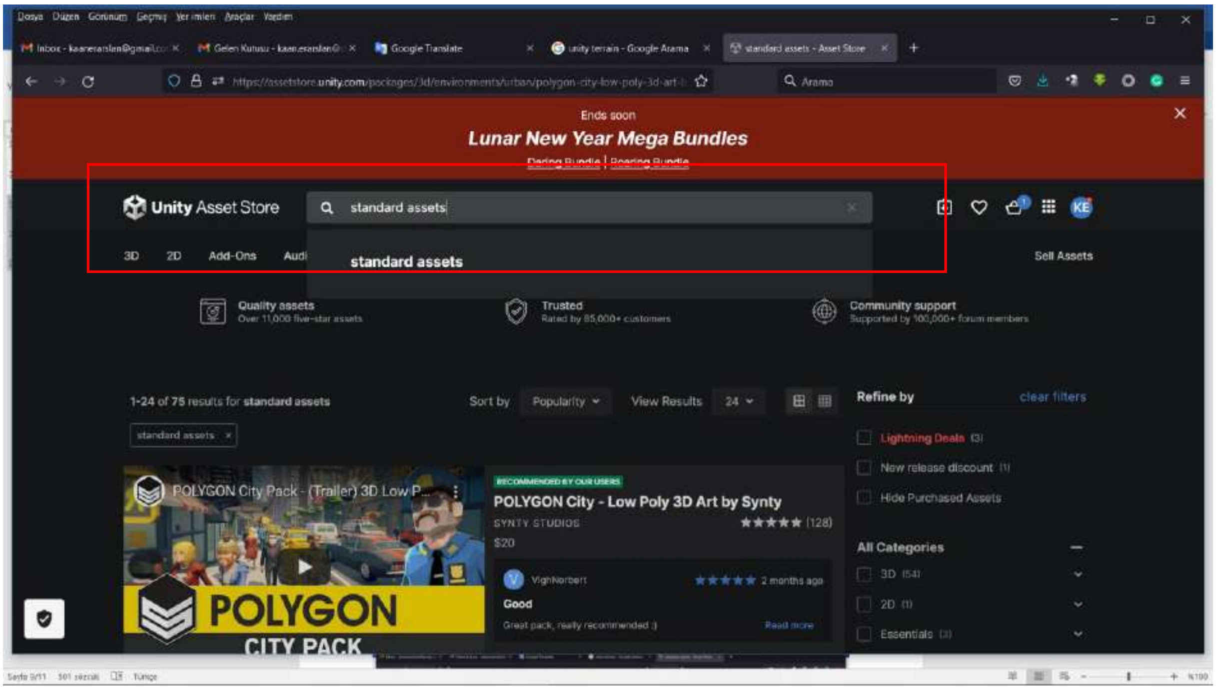
5.5.Terrain + Standard Assets Using Unity Asset Store

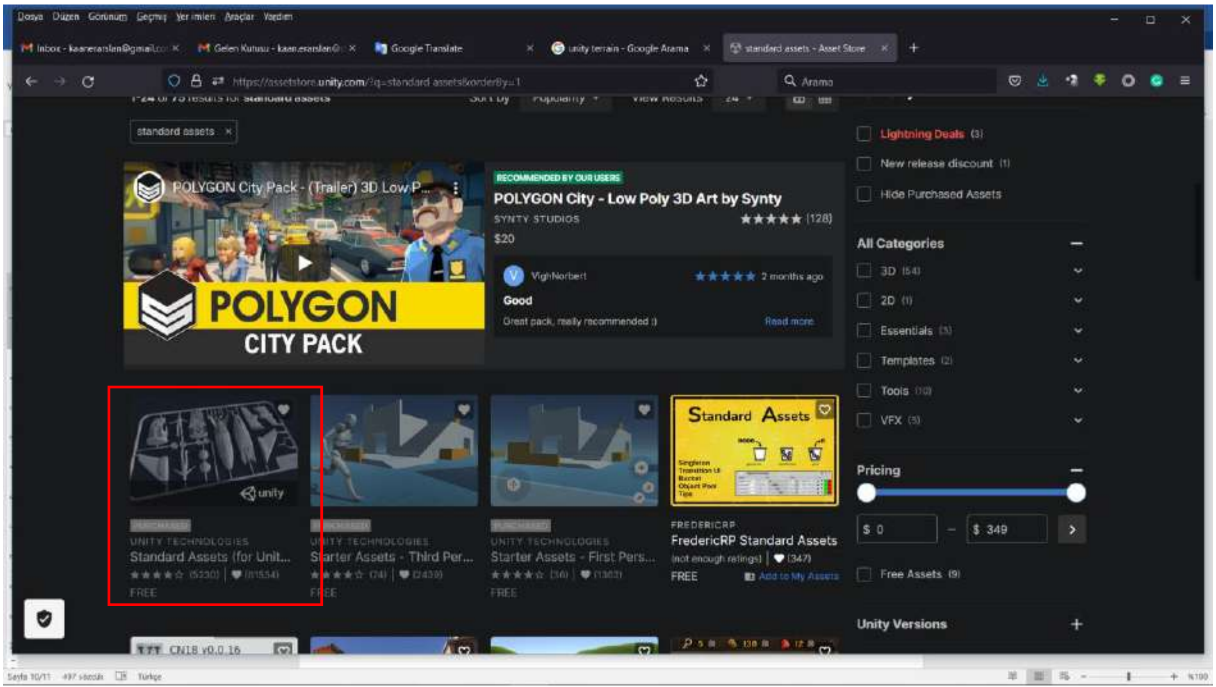
There are Unity Assets with very rich content to create natural landscapes. For this, the Unity Asset Store can be opened, and many ready-made assets, both paid and free, can be downloaded. For this, **Window>Asset Store** should be selected. Then, we connect to the website with the **Search Online** selection. It should not be forgotten that the **same account ID** should be used in both **Unity Hub** and **Unity Asset Store**. In this way, it is possible to download assets directly from the internet to our project.



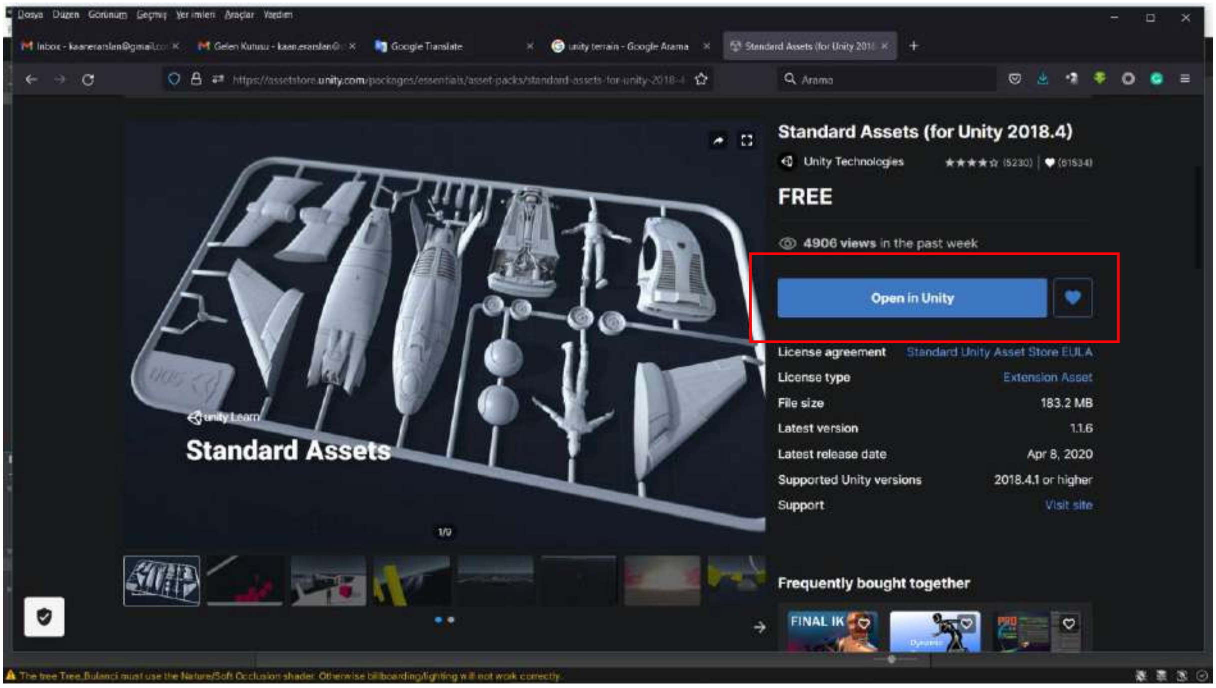


Here, **Standard Assets**, which is free and belongs to **Unity Technologies**, should be searched.

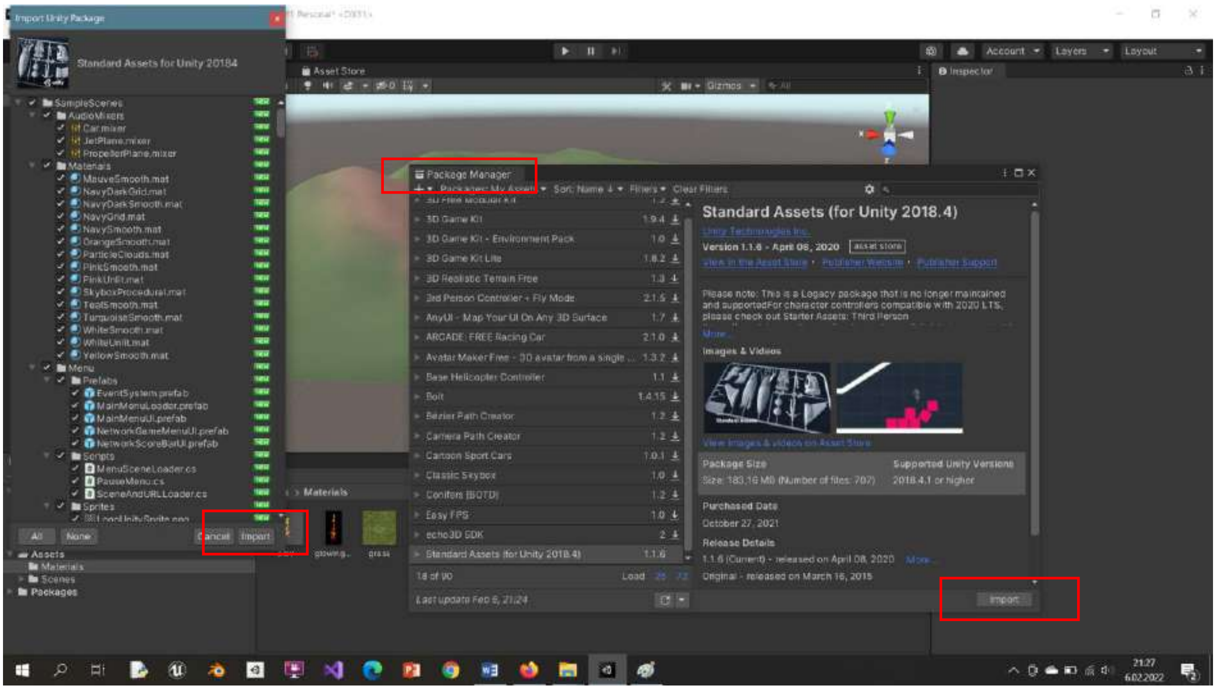




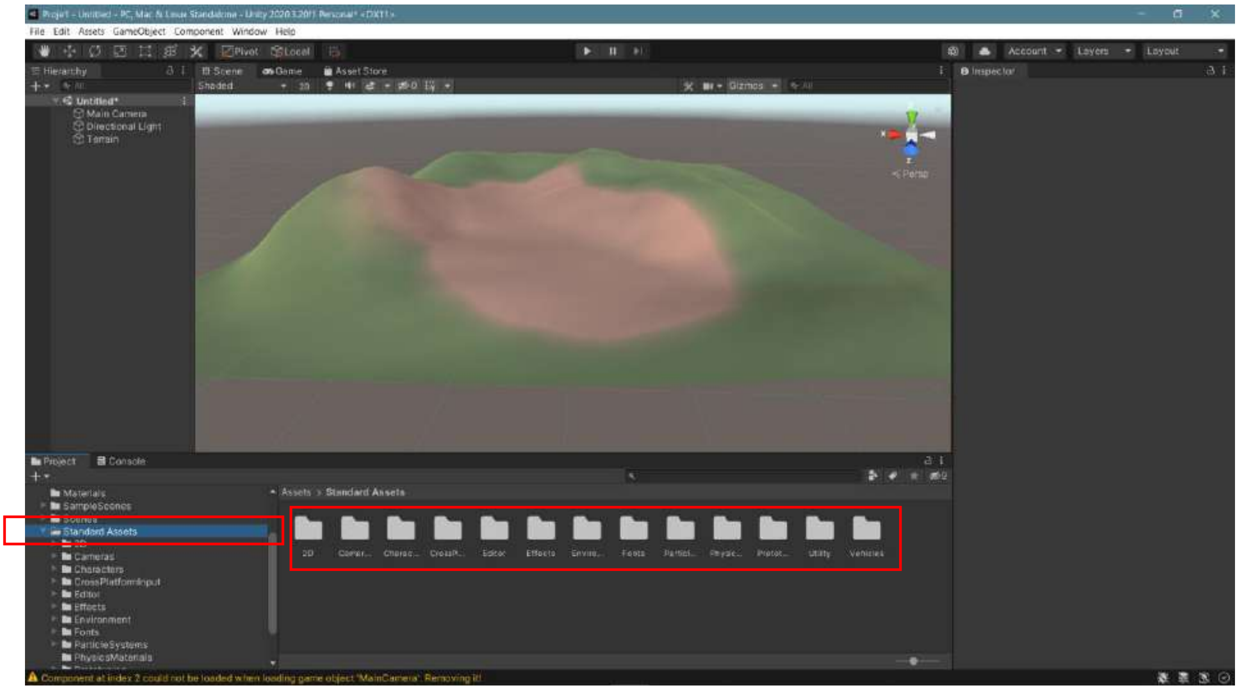
With this selection, we will first add the asset, which contains very rich materials, textures, scenes, C# codes, etc., to our **Assets** archive, and then download it directly to our **Package Manager** with Open in Unity.



Standard Assets will be visible and importable in the **Package Manager**. After import, another window will open and show all package content. With the **Import** key in this window, the package is added to our Assets window.



When we follow the path we followed before in **Terrain** under **Standard Assets**, we will see that new trees, textures, grass and flower models, vehicles, features, and **First** and **Third Person Shooter** (**FPS** and **TPS**) codes and features have been added.



With the terrain components offered by Standard Assets, much more successful and realistic field designs can be made. An application environment can be created with houses, transportation vehicles, people, animals, etc. added to this scene.

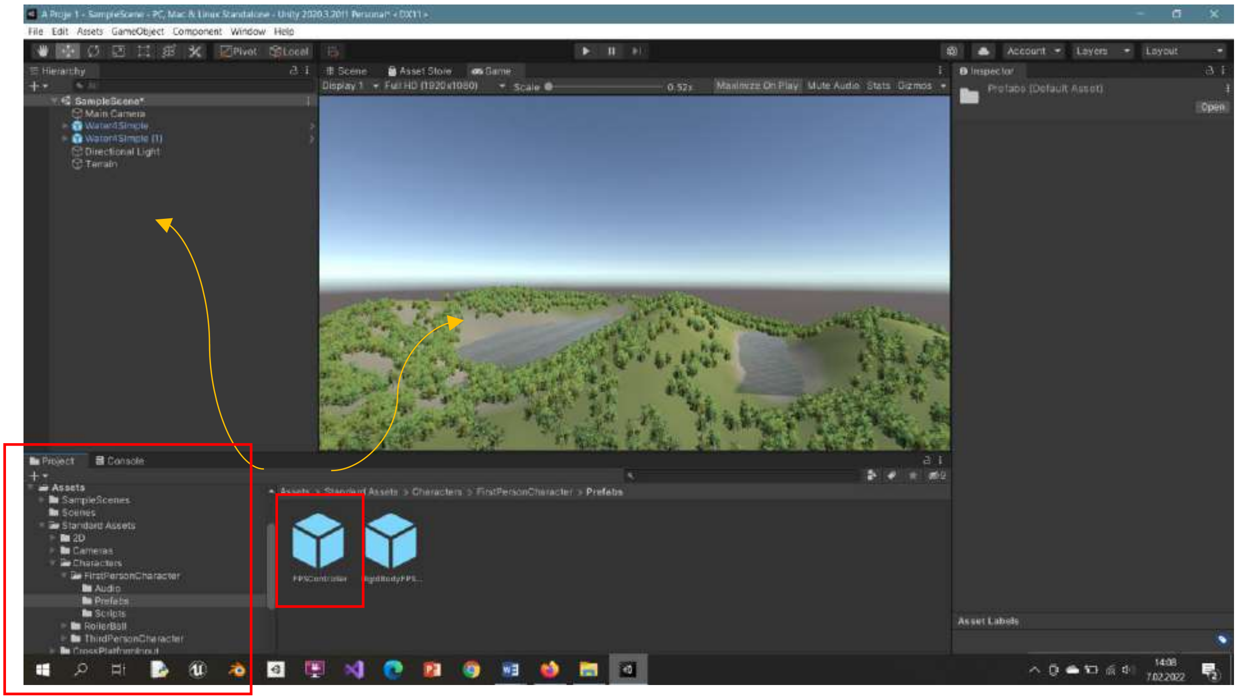
Moreover;
By positioning **Assets>StandardAssets>Environment>Water>Water4>Prefabs>Water4Advanced**, prefabs in the scene, lake, sea, and stream can be added.

Another feature offered by **Standard Assets** is the **FPS** and **TPS** character options that allow movement within the scene.

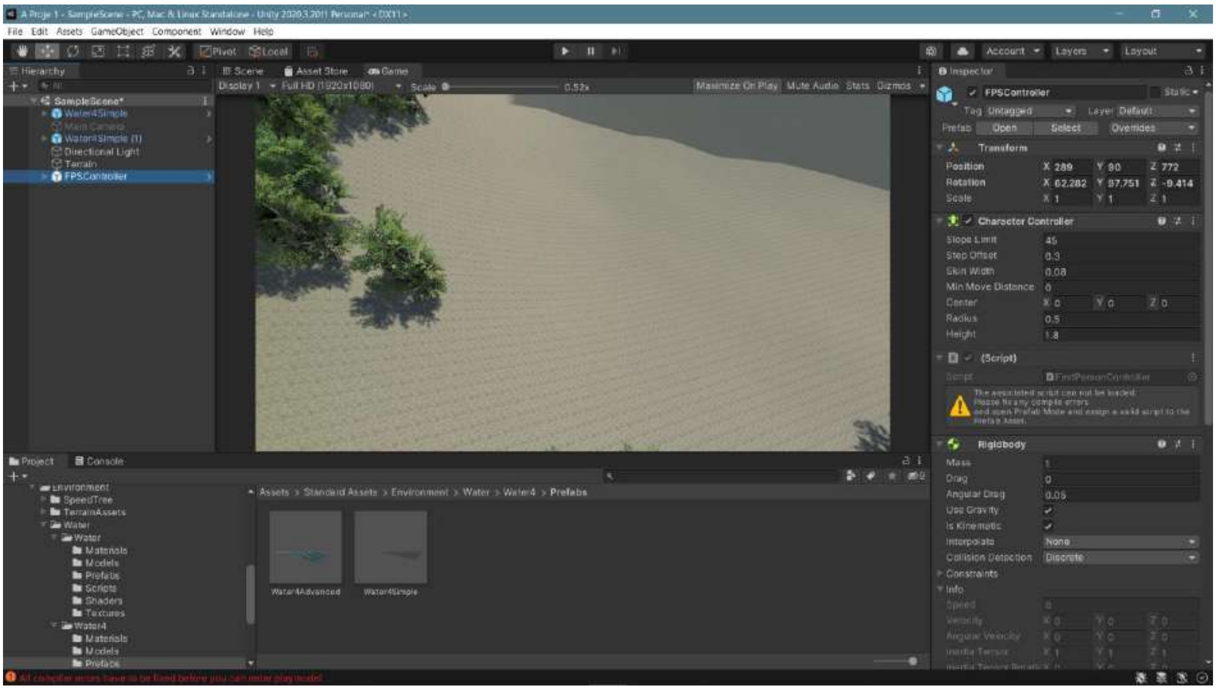
For an FPS scene, the **Main Camera** needs to be cleared (or inactive). Afterwards;

Assets>Standard Assets>Character>FirstPersonCharacter>Prefabs>FPSController

The prefab control is added by dragging it to the scene or **Hierarchy**, following this order.



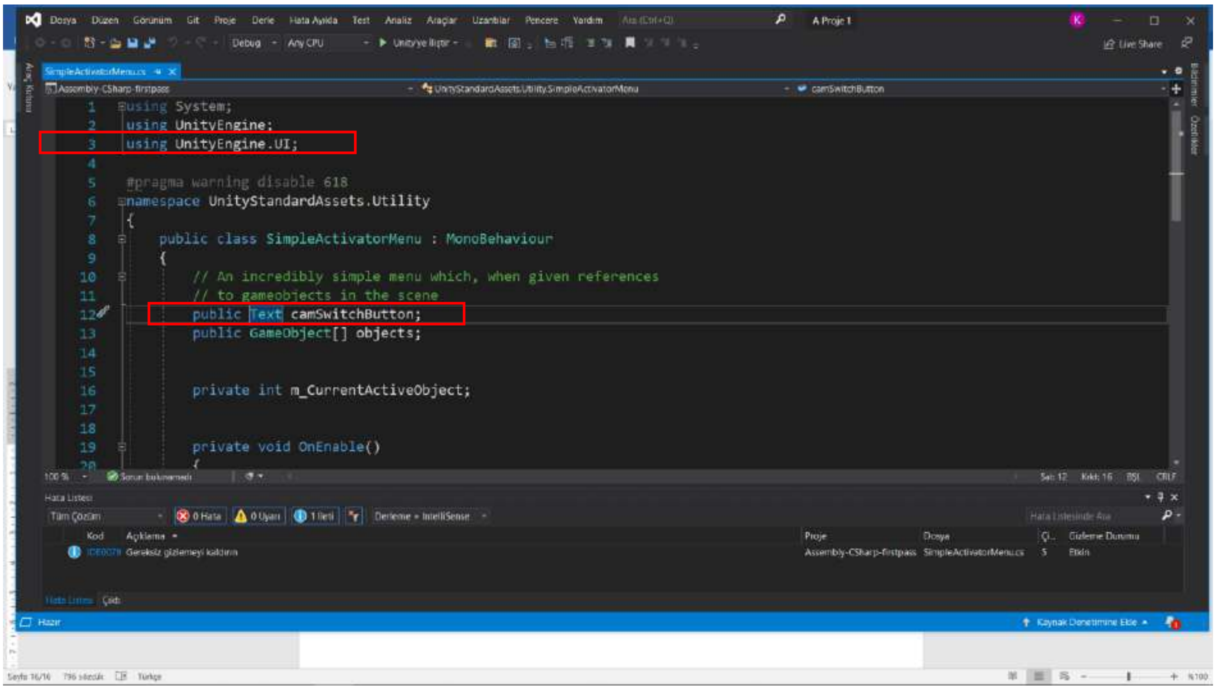
FPSController has its camera. Therefore, the position and viewpoint of the **FPSController** in the scene are adjusted.



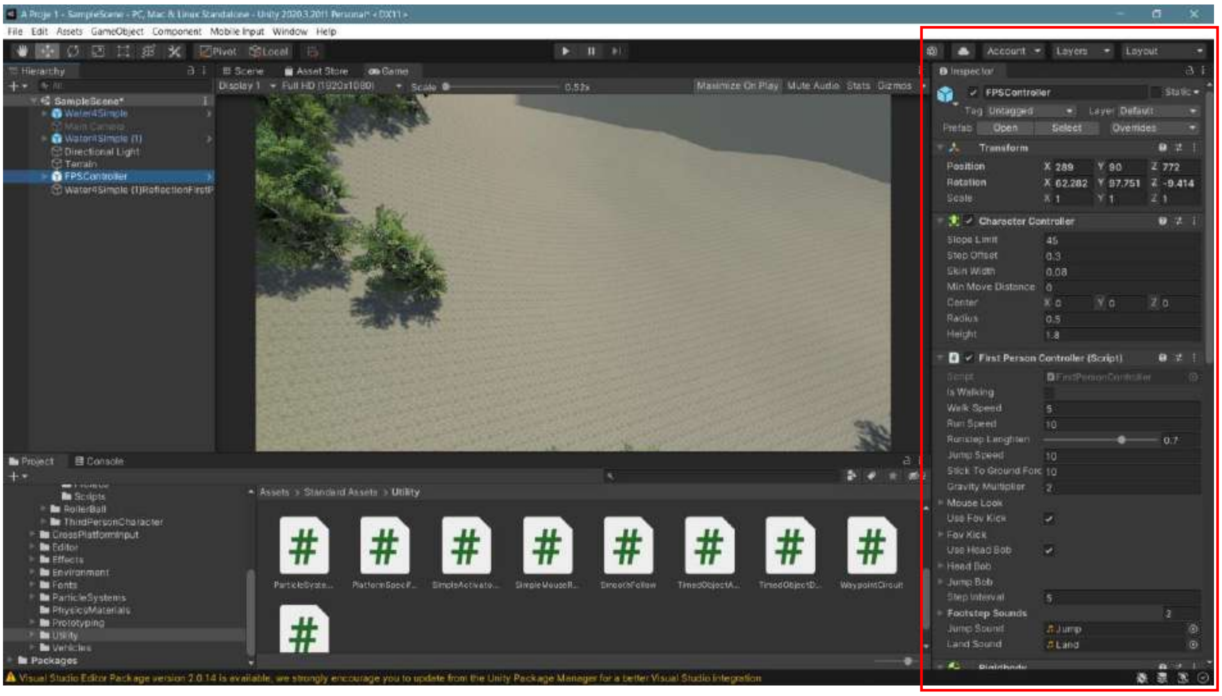
In Unity 2022, a line of code under **Standard Assets** needs to be added, and a small change needs to be made to a line of code. To do this, in the **Assets>Utility>SimpleActivatorMenu.cs** file, add

using UnityEngine.UI; and change the **public GUIText camSwitchButton;** line to **public GUIText camSwitchButton;** as

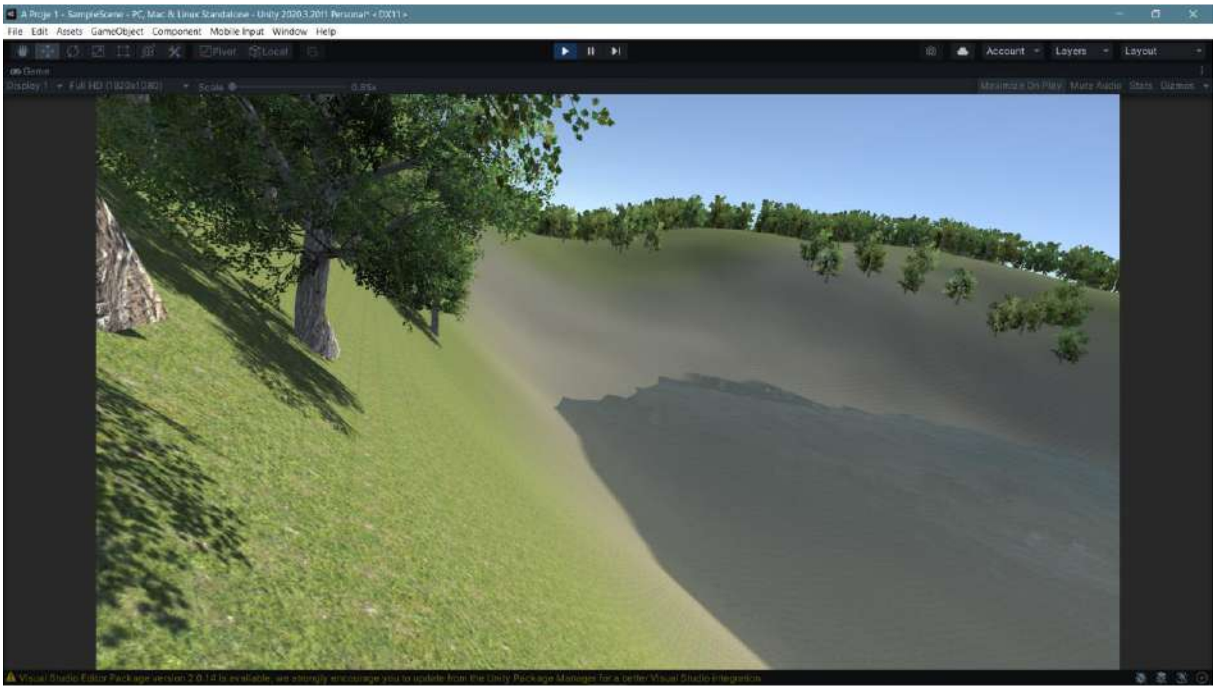
GUIText >Text:



Settings in the Inspector of **FPSController** can be changed depending on personal preferences.



As a result, an application is run that can be navigated with the **arrow keys, w, a, s,** and **d keys**, jumped with the **space bar** and controlled 360-degree viewing movements with our **mouse**.

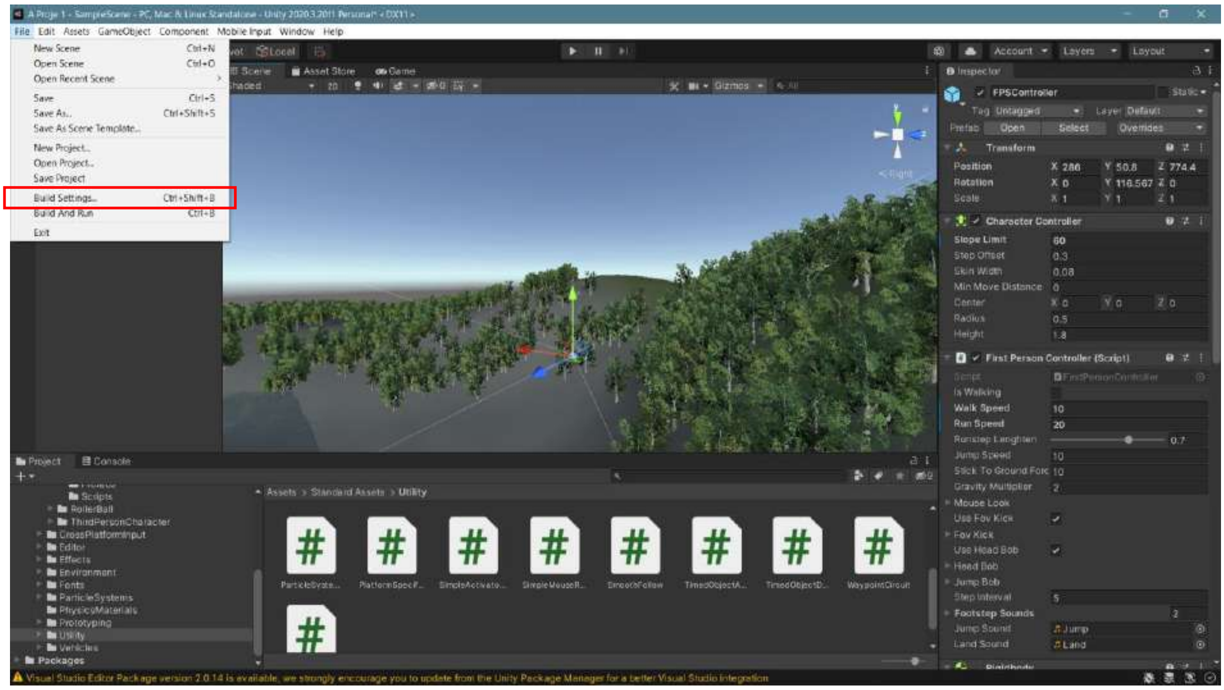


6. DEPLOYING PLATFORMS

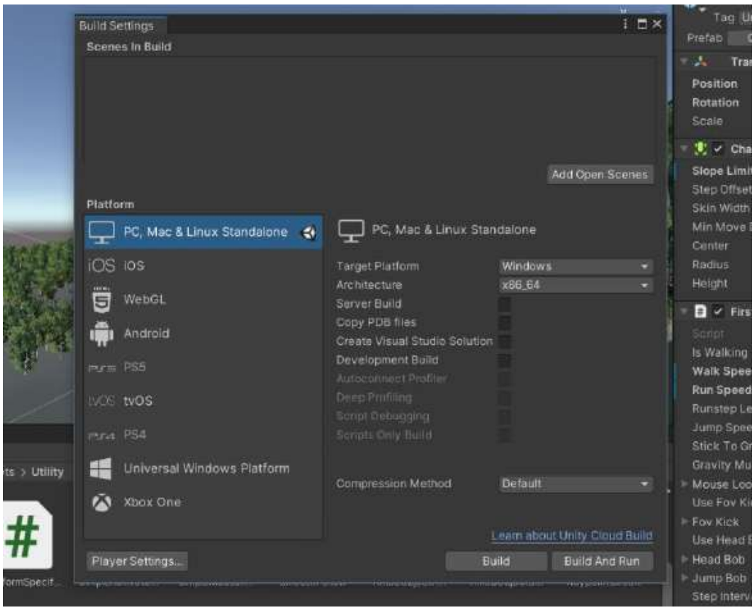
6.1.Build Settings

It was stated that **Unity** is a **cross-platform**. This means that projects produced in Unity can be adapted to computers, web, mobile devices, VR (virtual reality), AR (augmented reality), and game console platforms. Let's see how the FPS application we developed for Terrain can be converted into a computer application in **EXE** format.

After the project is completed, **Build Settings** should be selected from the **File** menu.

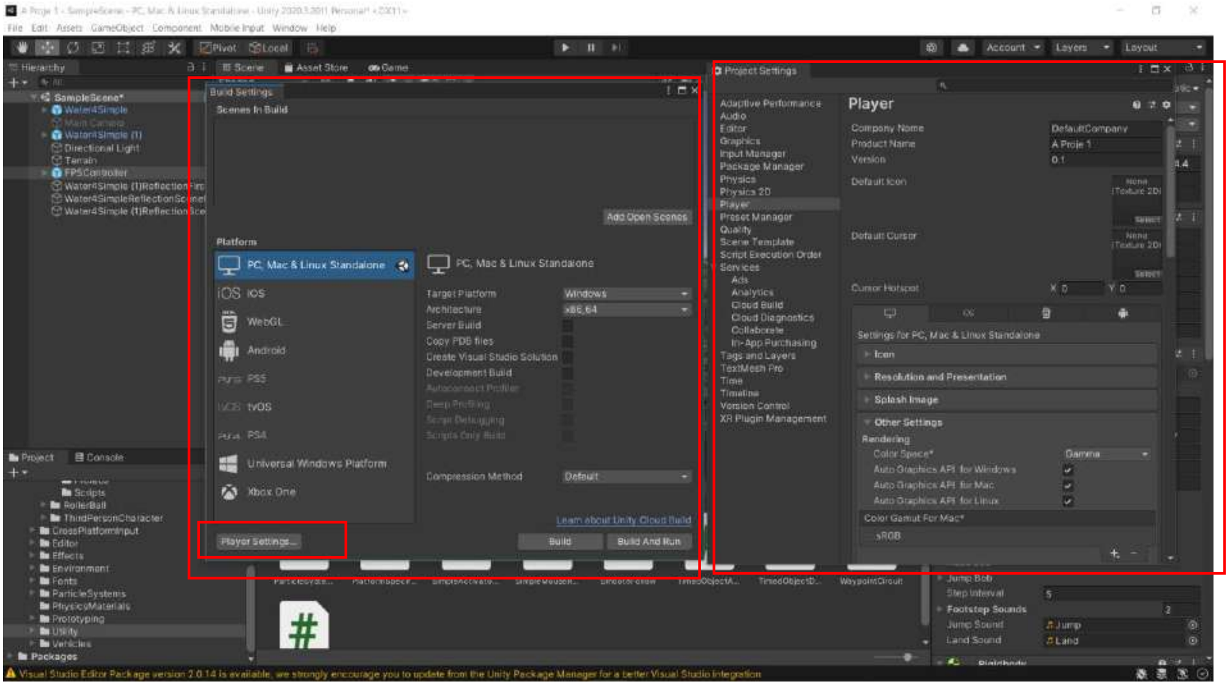


The window that opens contains the scene, platforms and other settings.

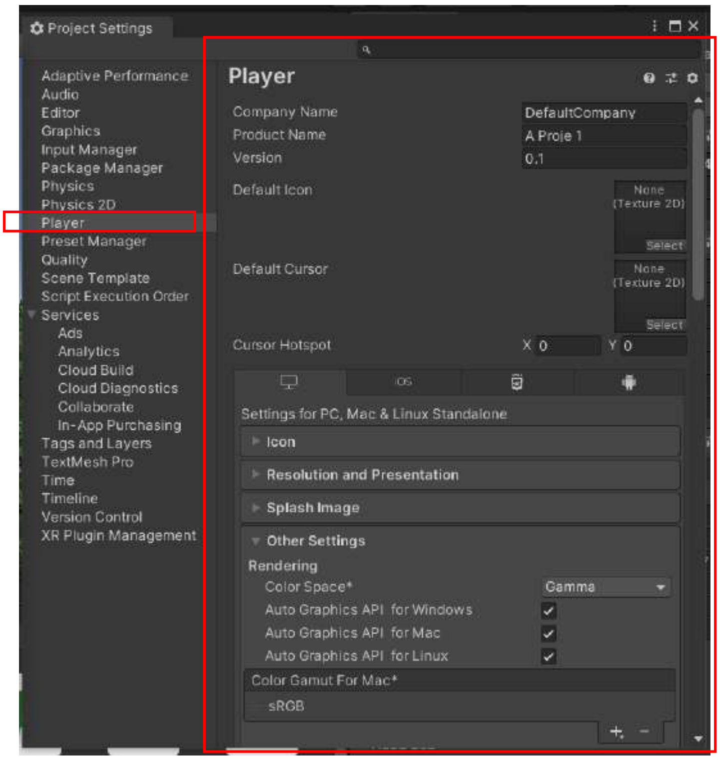


Our scene must be added to the list with the **Add Open Scenes** button. Since we will not be switching to another platform, changing the **Platform** section is unnecessary. However, another platform should be selected according to the type of application, and the **Switch Platform** button that will appear after the selection should be used to switch.

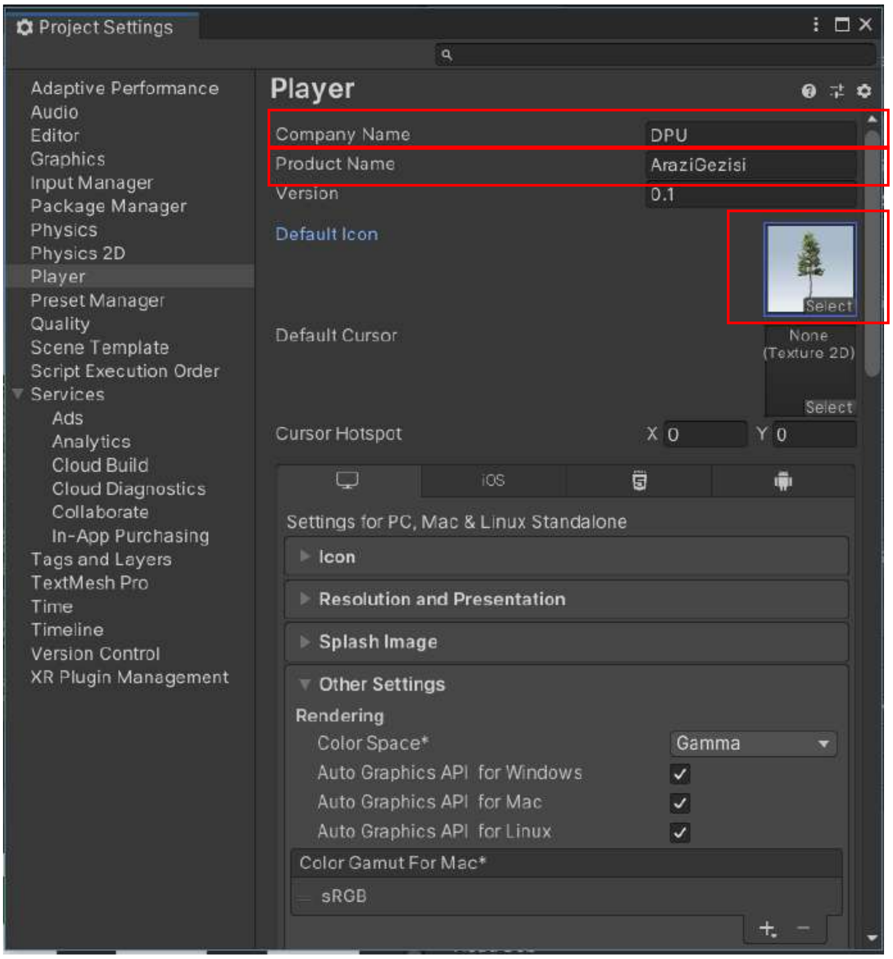
One of the most important buttons under **Build Setting** is **Player Settings**.



Player Settings are in the **Project Settings** window, along with many other settings.



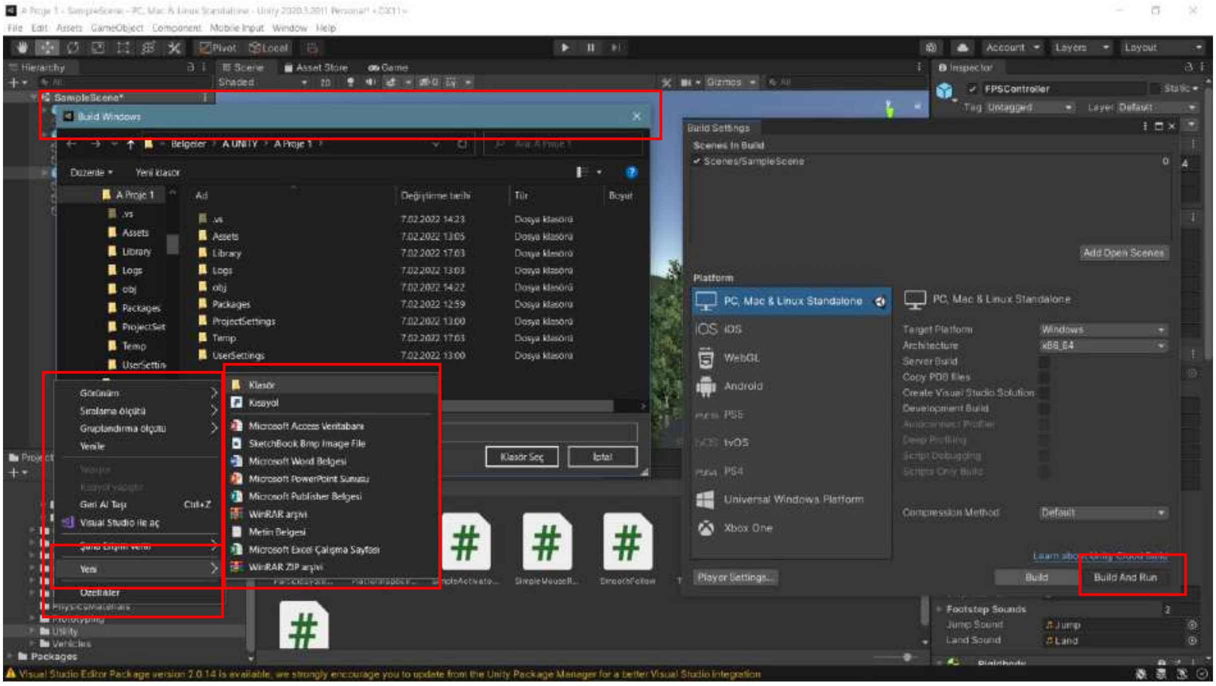
Company Name and **Product Name** information must be entered for the project title that will be converted into an application. An image file prepared for the icon image that will appear in the folder is placed in the **Default** Icon section.

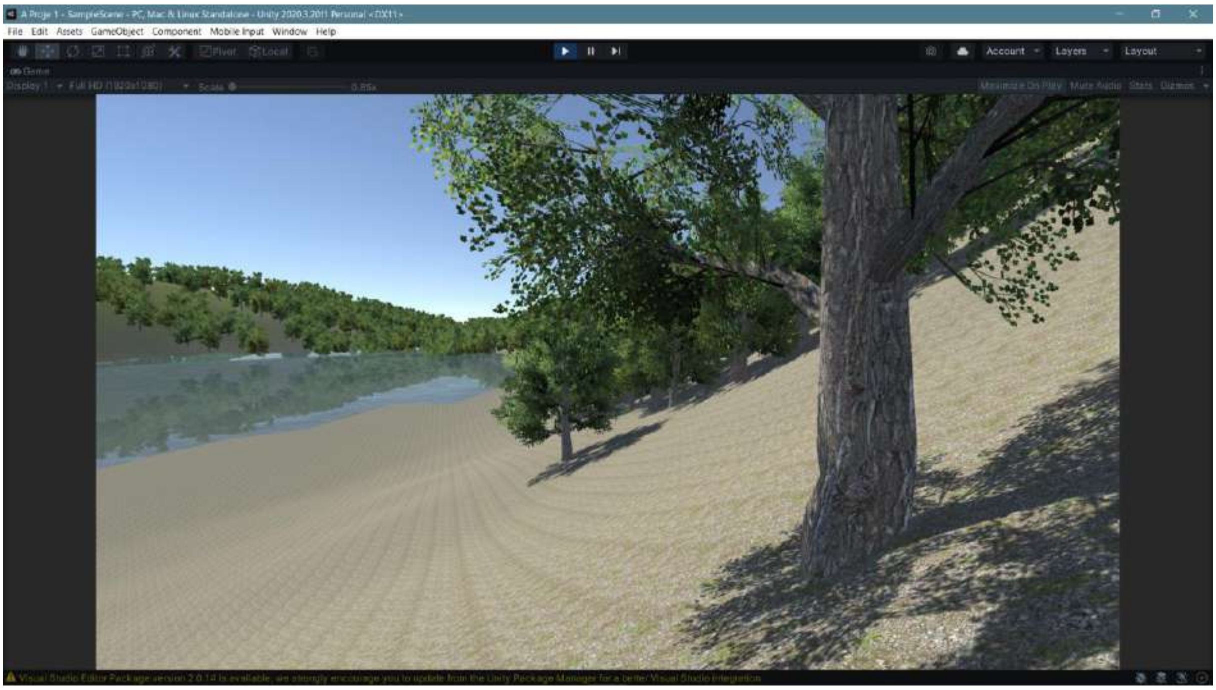


These definitions are sufficient for the PC application. Now we can proceed to convert the scene to an application (**EXE program**). To do this, click the **Build and Run** button in the **Build Settings** window. With this selection, the program asks us to create/open the folder where the application will be created.

In the open window, right-click on the mouse and select **Folder** from the **New** option. Save the folder name of the application and select it. The program will automatically start compiling and creating **EXE**. It is normal for this process to take a certain amount of time, depending on the computer's hardware power.

After the application is compiled, the game starts immediately. After visiting the field, press **Alt+F4** to **exit** the game. The **EXE** file and other component files and folders are created under the specified folder.

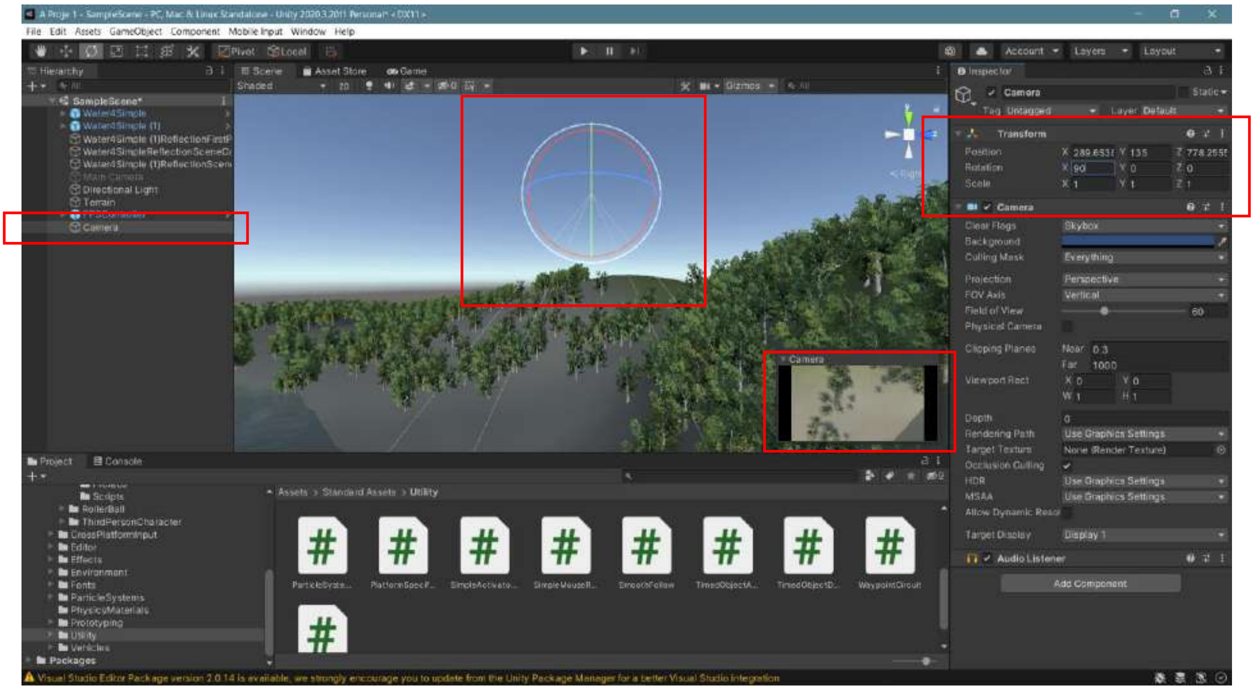




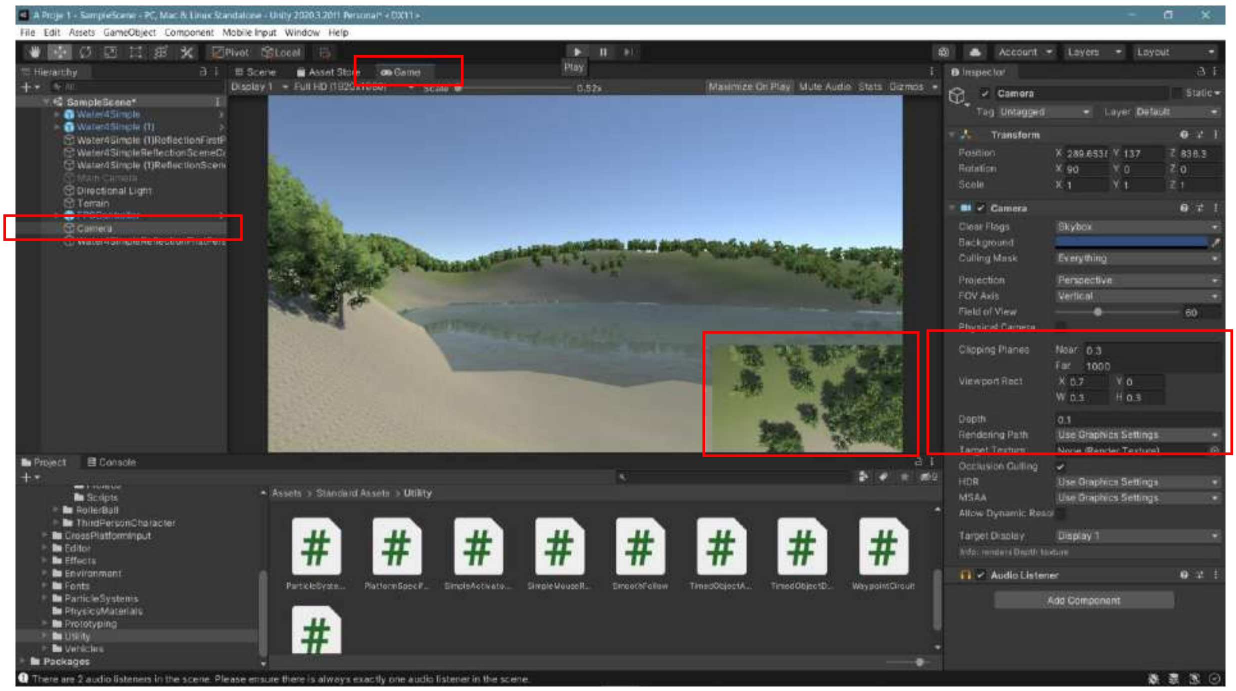
6.2.Cameras

Other **cameras** can be added next to the current camera, including their viewing angles, screen sizes, and locations. To do this, right-click on our mouse under Hierarchy, create a camera, determine its angle, and position it in place.

In the example, a camera is added, positioned to see the field from above, and its angle is adjusted.



To get results from the **Inspector>Viewport Rect** setting, the positions on the screen are set with **X** and **Y**, and the dimensions of the new camera on the screen are set with **W** and **H**. If the **Depth** setting is a number greater than zero, it will be seen together on top of the **FPS** camera. It is possible to see these settings more clearly in **Game** mode.



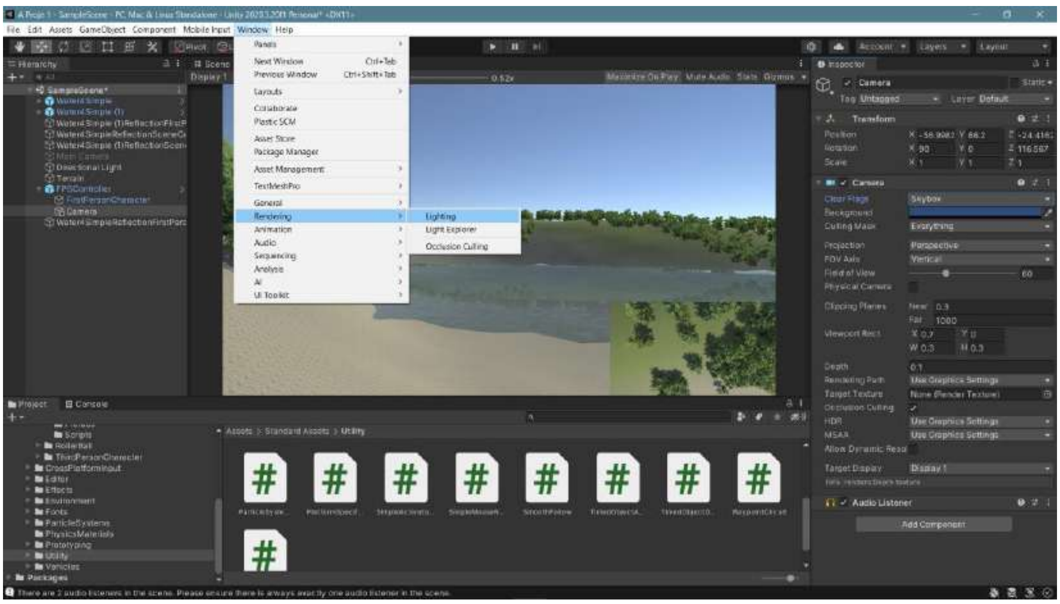
However, finally the **Camera** needs to be dragged under the **FPSController**. It is possible to add other cameras and different angle views.

6.3.Parent Child Relation

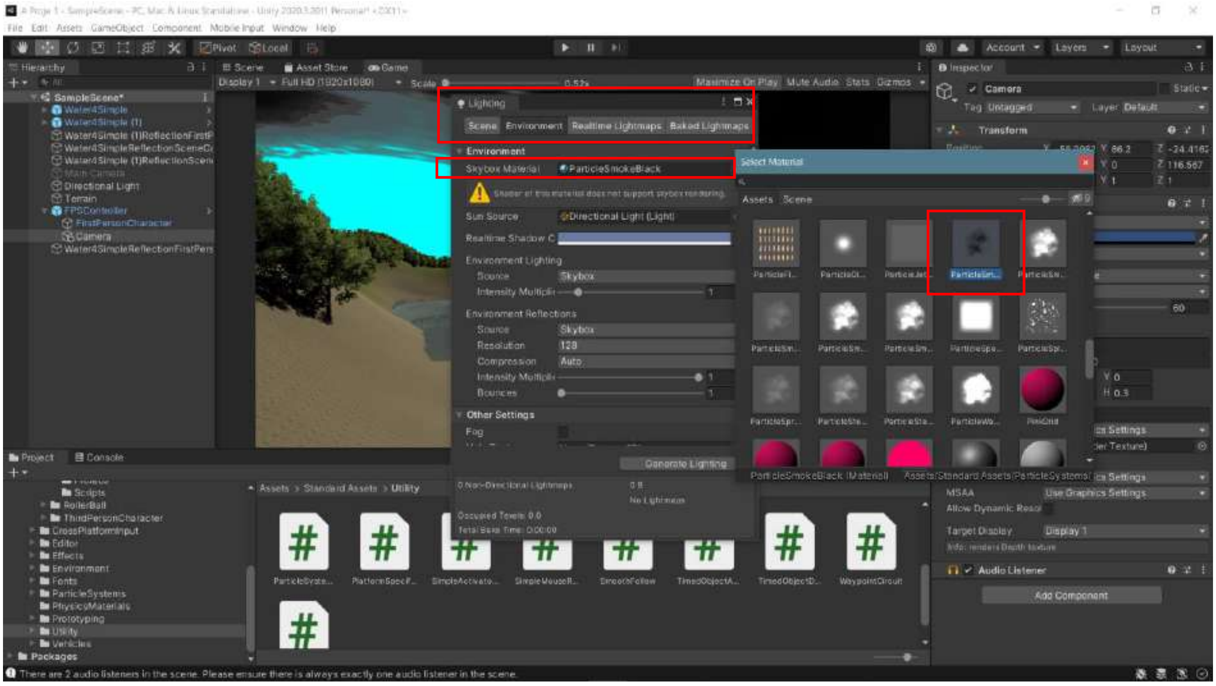
By dragging the camera under the **FPSController**, a **Parent-Child** connection is created between them. The camera becomes the child of the **FPSController** and is subject to the movements of the **FPSPlayer**. In game mode, when the **FPS** moves, the other camera will follow it, and the image above will follow the image below.

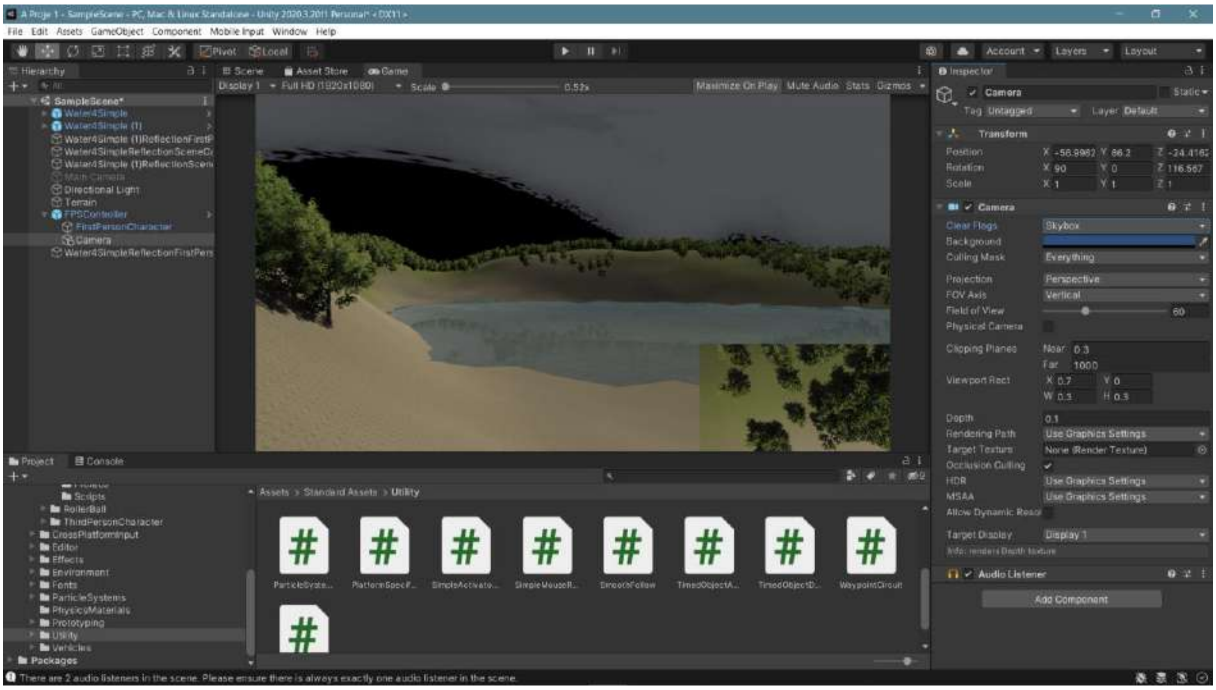
6.4.Skybox

The **sky** seen behind the scenes is a standard application. If desired, it can be changed to a fixed color or different sky backgrounds. Skybox assets can also be found in the Asset Store. For the Skybox application, the relevant window is opened via **Window>Rendering>Lighting**.



In the **Lighting** window settings, **Environment>Skybox Material** opens the window with the image files. The selected image file is assigned as the new sky.

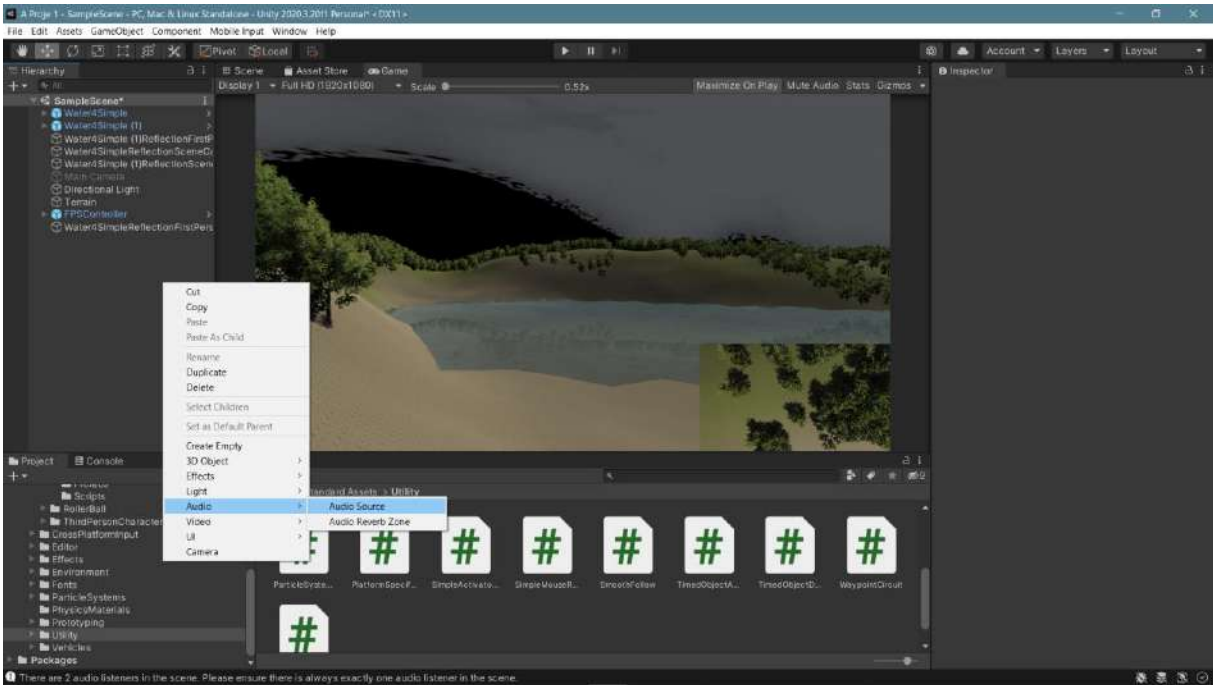




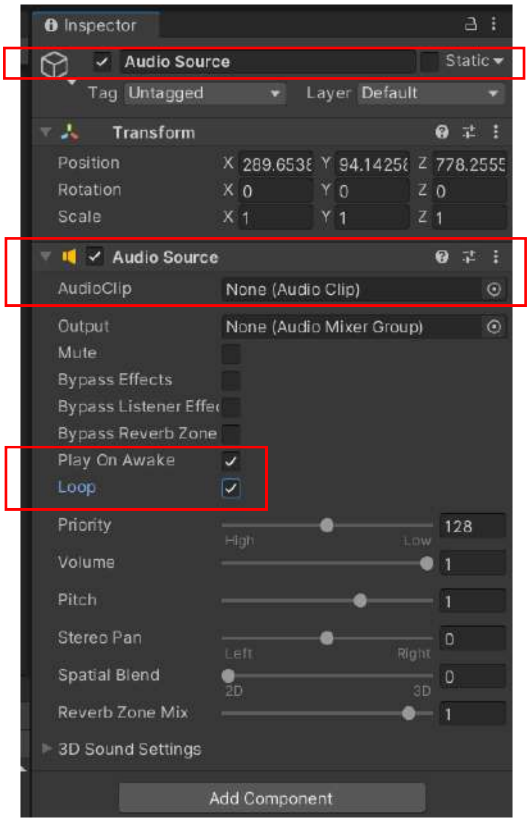
In a normal project where Standard Asset is not used, **Skybox** change is simpler. **The main Camera is selected,** and a fixed color can be selected instead of Skybox in Inspector.

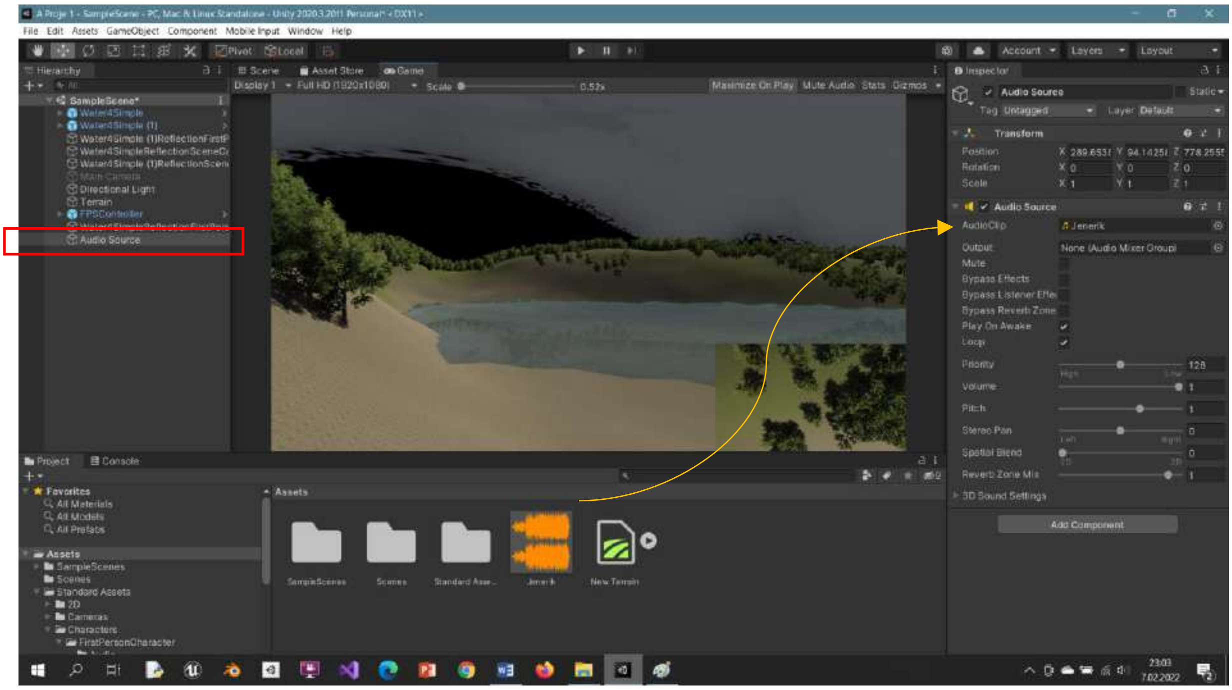
6.5.Adding Audio and Video files

An audio file is added to the scene via **Hierarchy>Audio>AudioSource**.

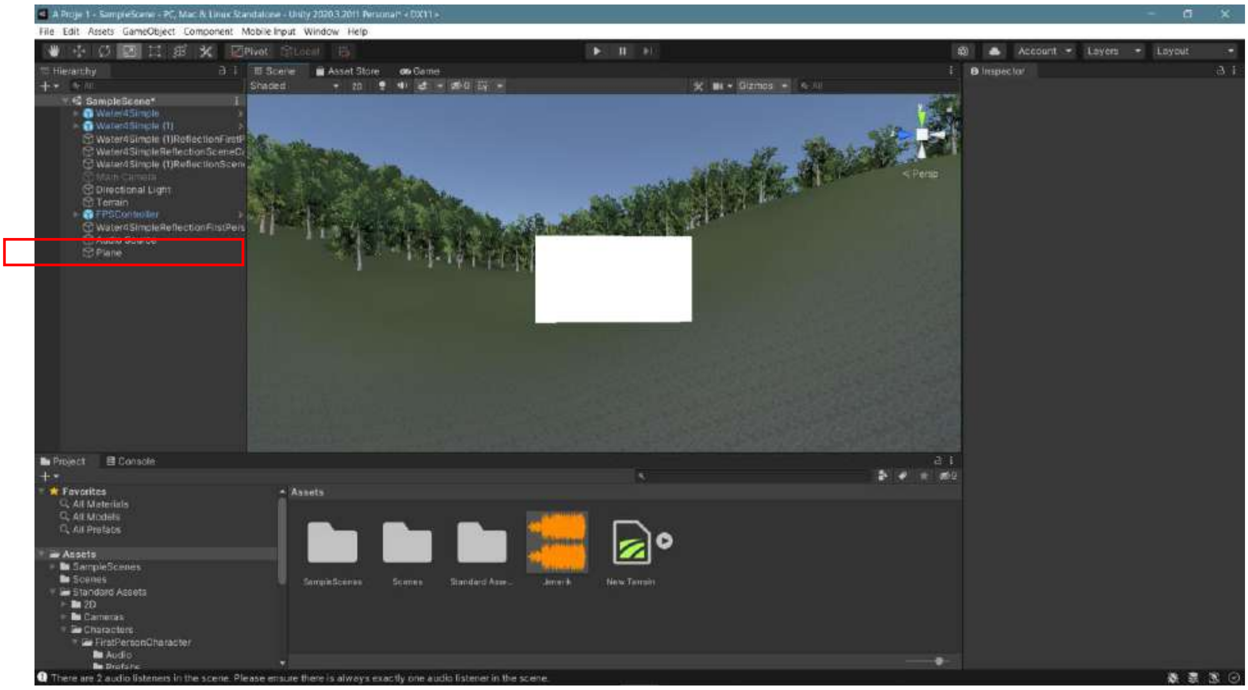


Afterward, Audio is selected, and the previously prepared audio file in the **Inspector** is dragged to the **AudioClip** area and connected. The **loop** check box is checked to make the audio file loop continuously.

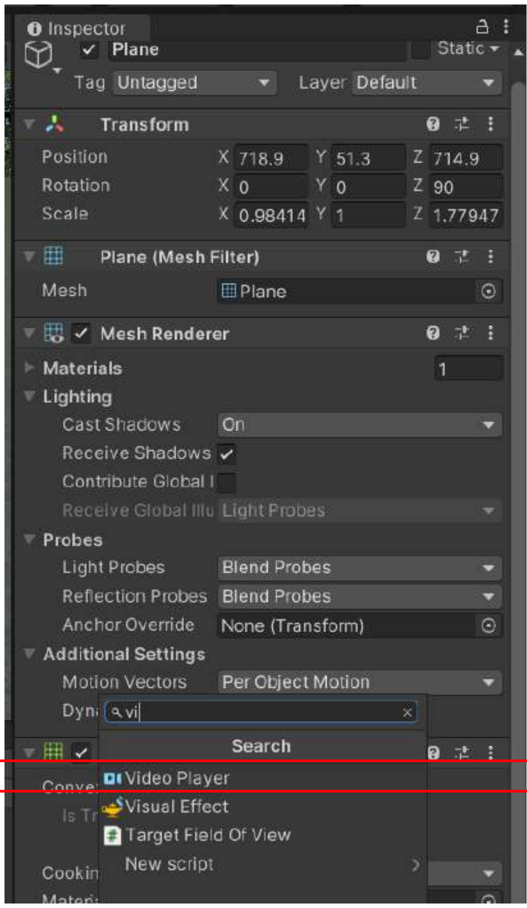




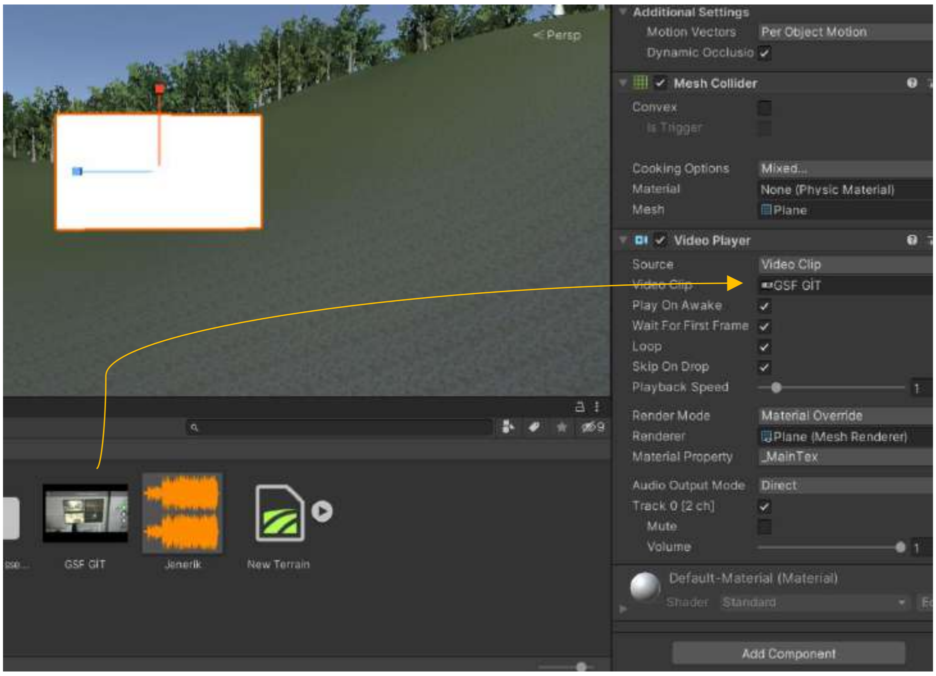
To add an image file to the scene, it is possible to add a **Plane** to which the file will be assigned. The **Plane** is positioned on the scene and its dimensions are adjusted.



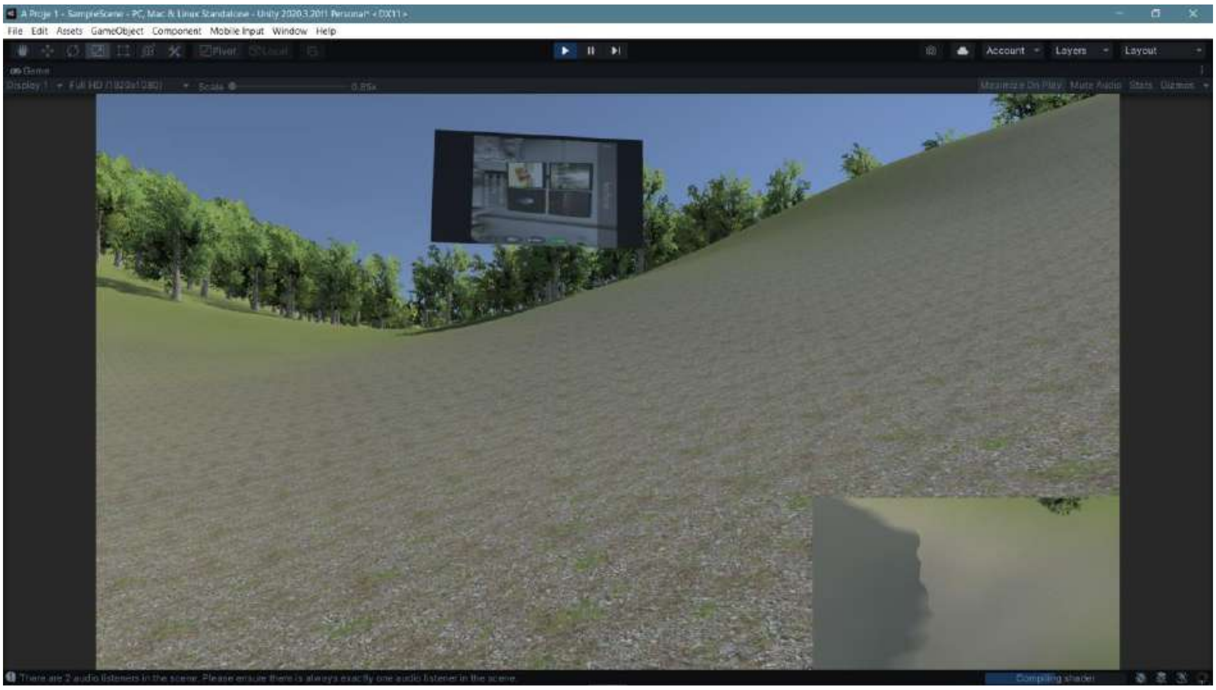
To add the prepared **video film**, first select **Plane**. The component is added with the **Add Component>VideoPlayer** selection.



Video Player has been added to the **Inspector** connected to the **Plane**. The **video file** that was previously prepared and added to the Assets section is dragged to the **Video Clip** area and connected here. If the **Loop** box is checked, it will loop.



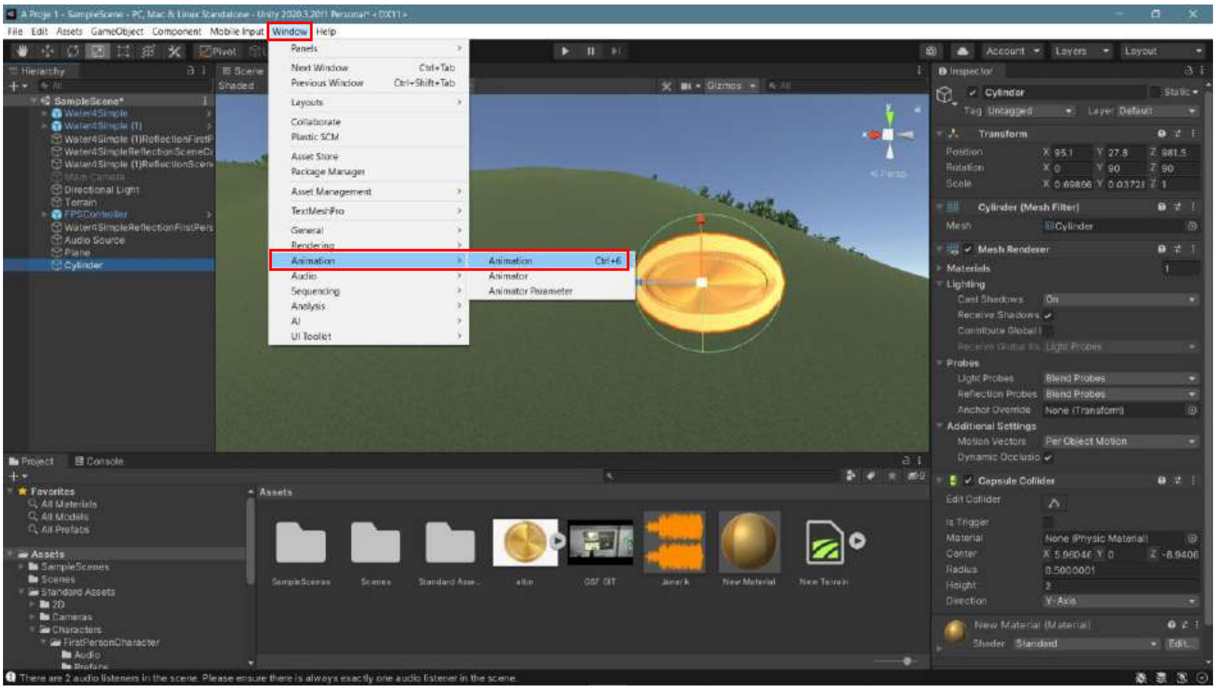
The **Plane** added to the field now has a video player feature.



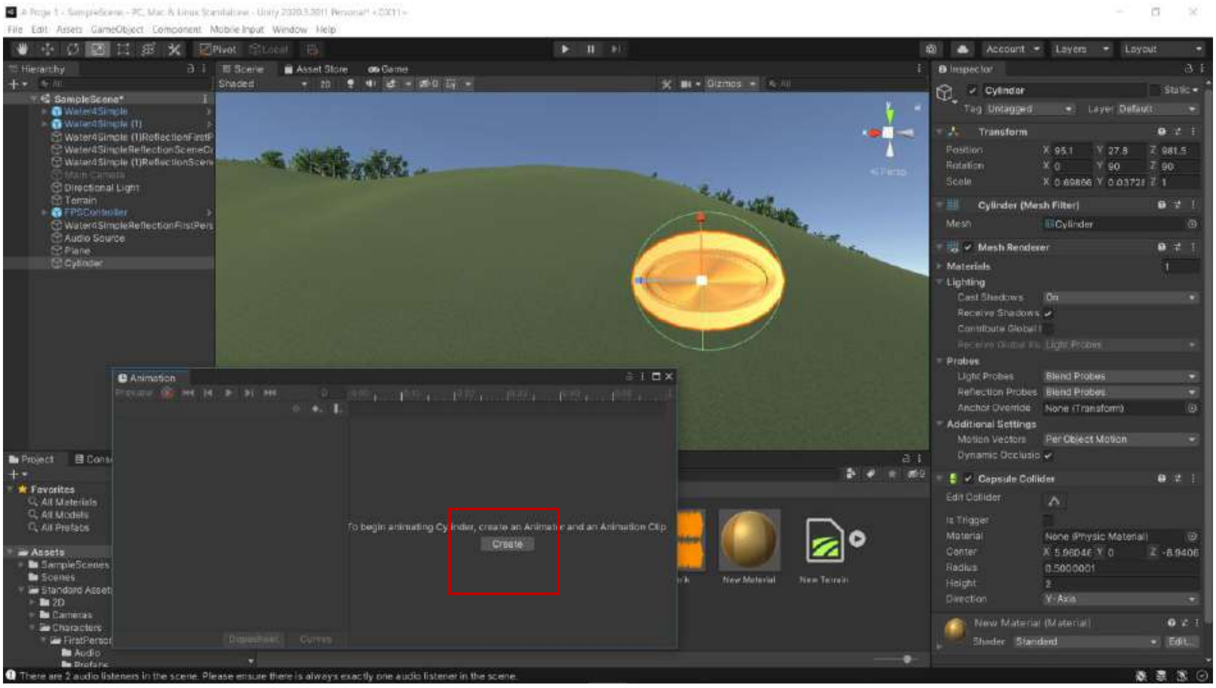
6.6.Animation

Let's add an **animation** without changing the scene. Animations can be added to a certain extent in Unity. The relevant object is selected and applied to it.

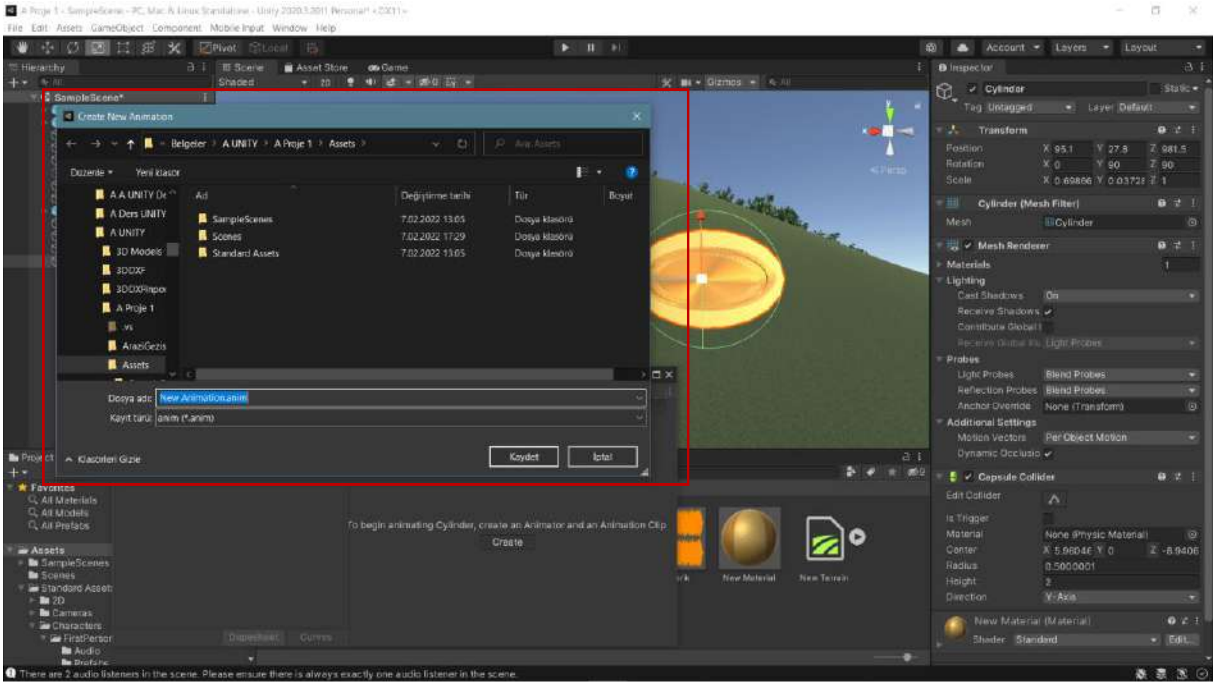
Let's add any 3D object or a cylinder to our scene and scale it like the gold image. Let's choose **Window>Animation>Animation**.



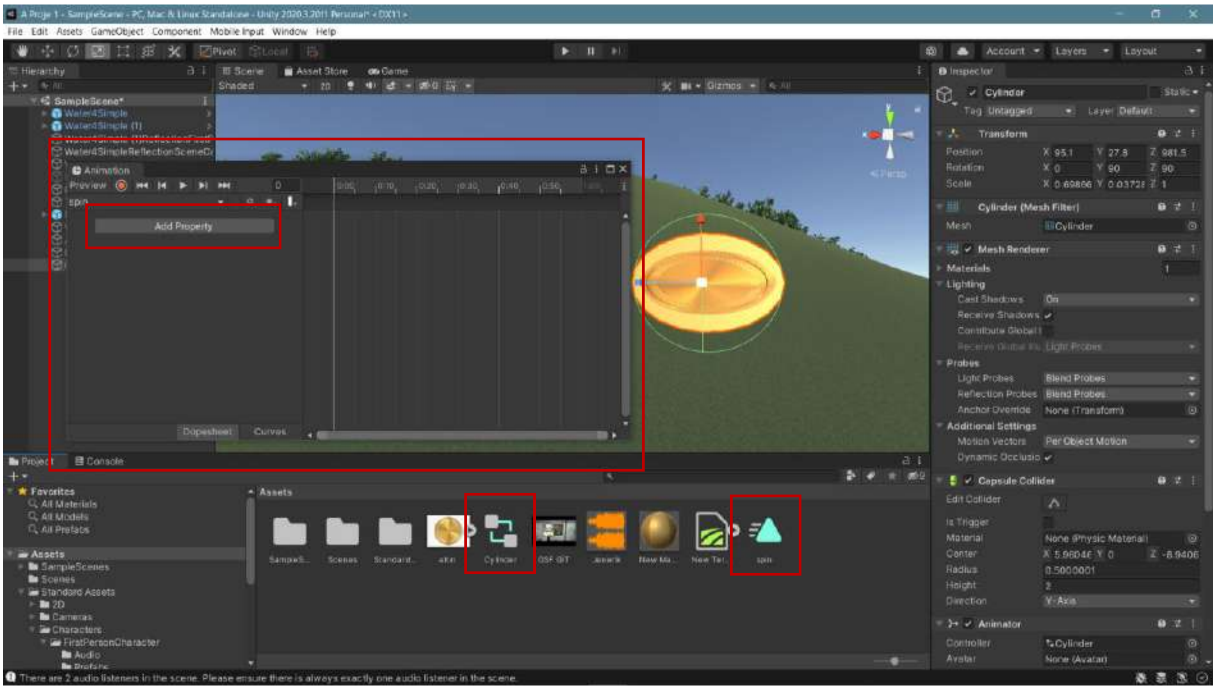
A window opens for the animation. Here, a new animation is started with the **Create** button.



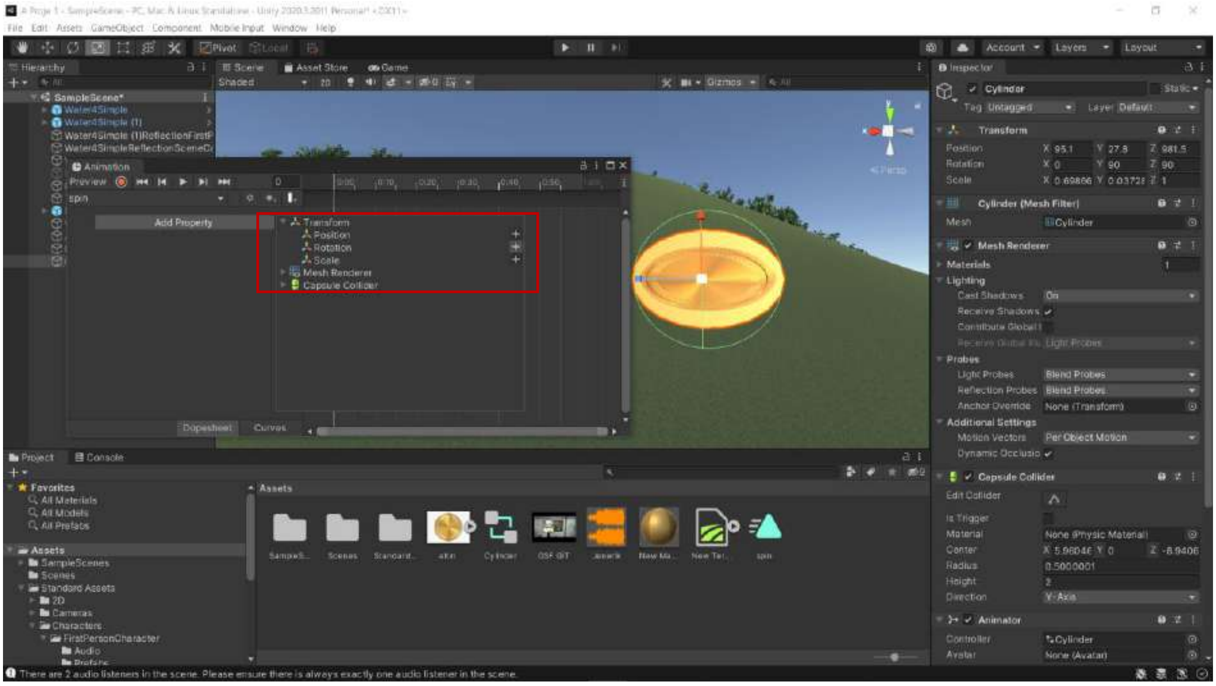
First, a file name is given to create the file related to the animation.



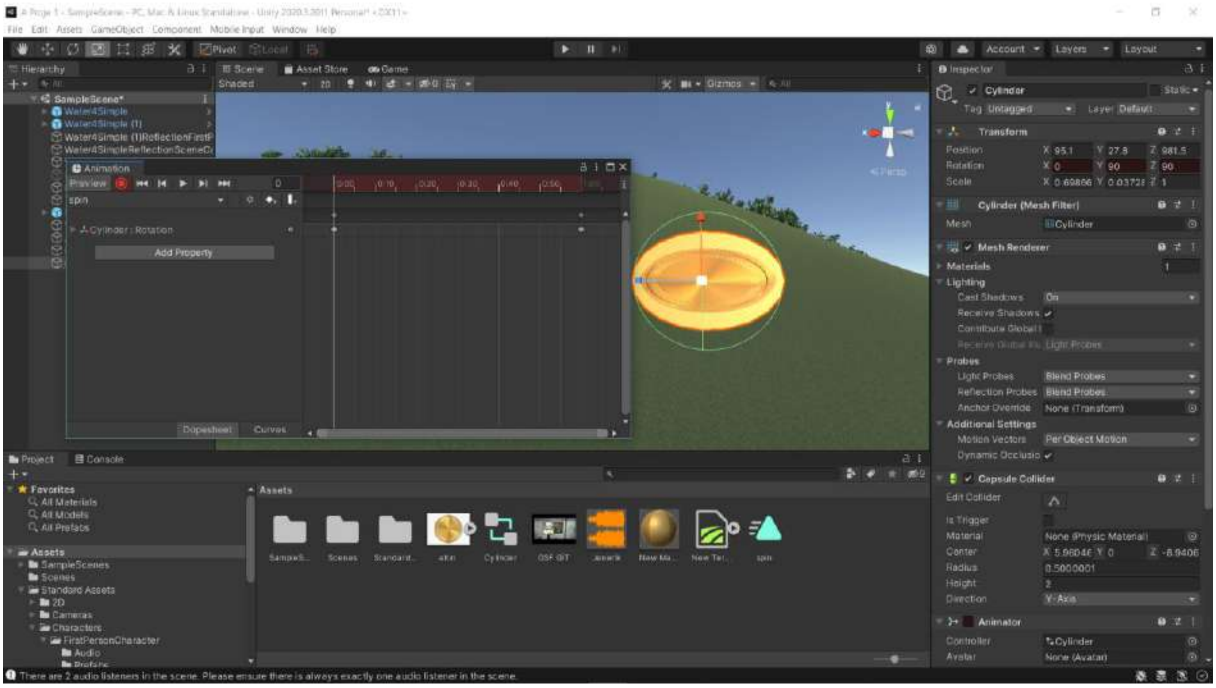
We named the animation **spin**. After that, a new window will open where we will create the animation.



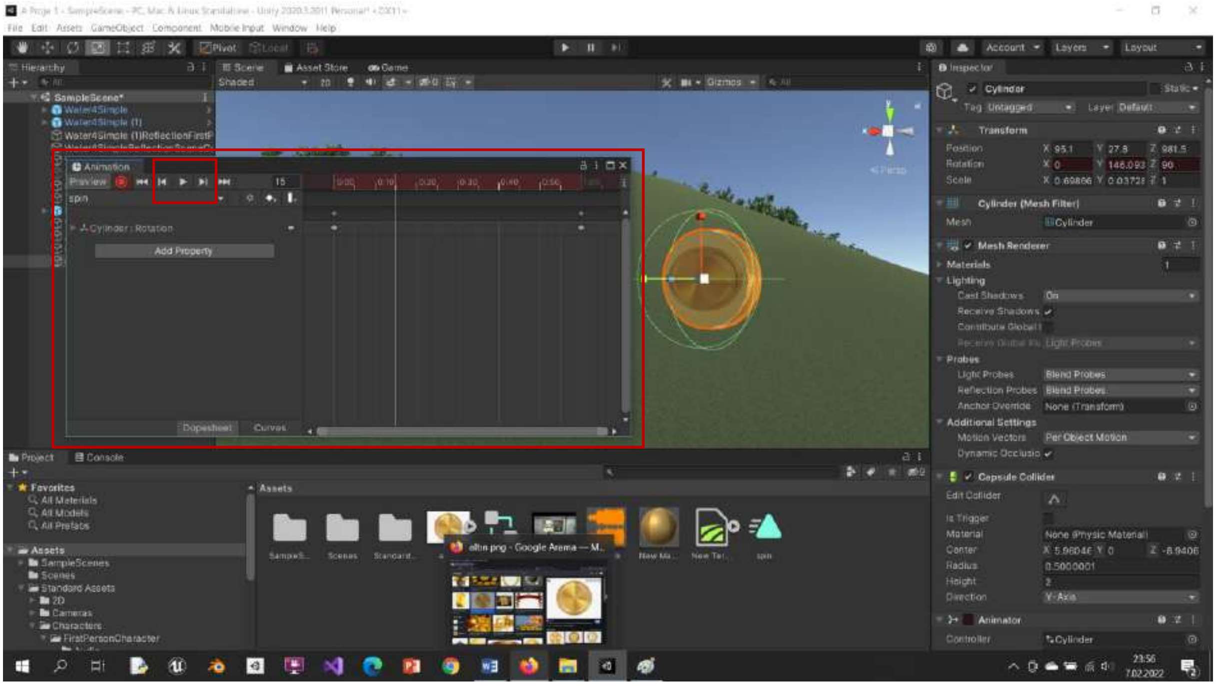
With **AddProperty**, animation types will be displayed. **Rotation** is selected under **Transform**.



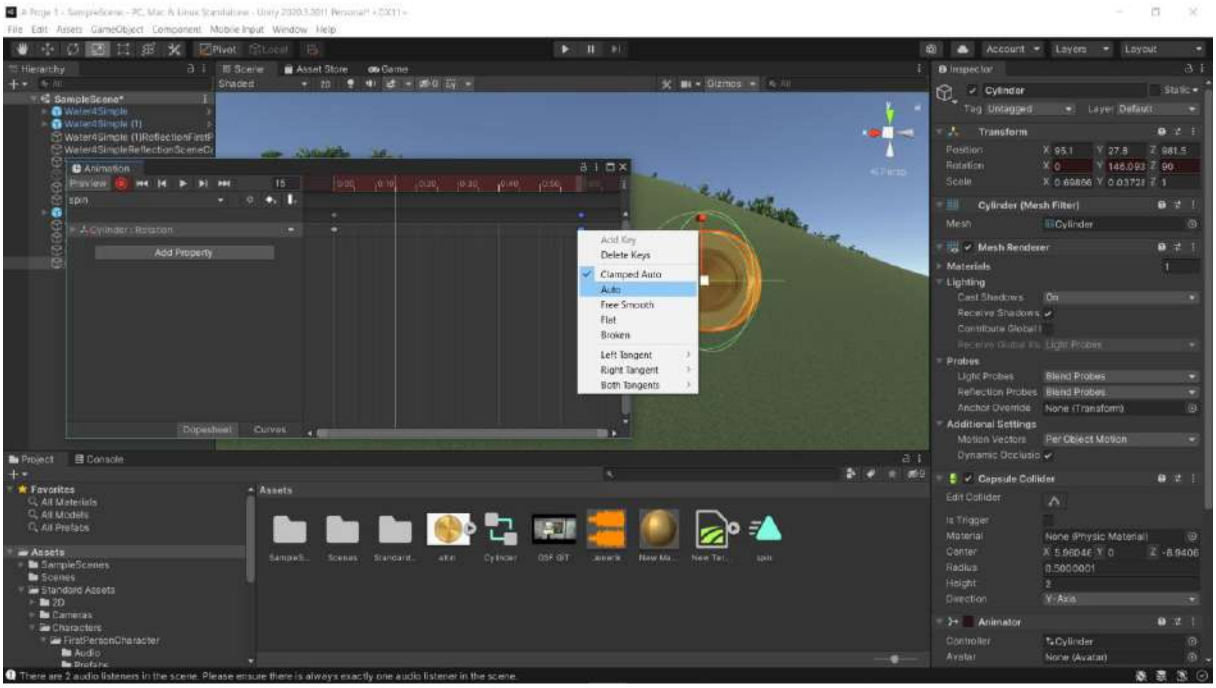
The animation determination process starts by pressing the **red button** - Enable/Disable. Here, the **timetable** and the **Rotation** line in the **Inspector** turn red.



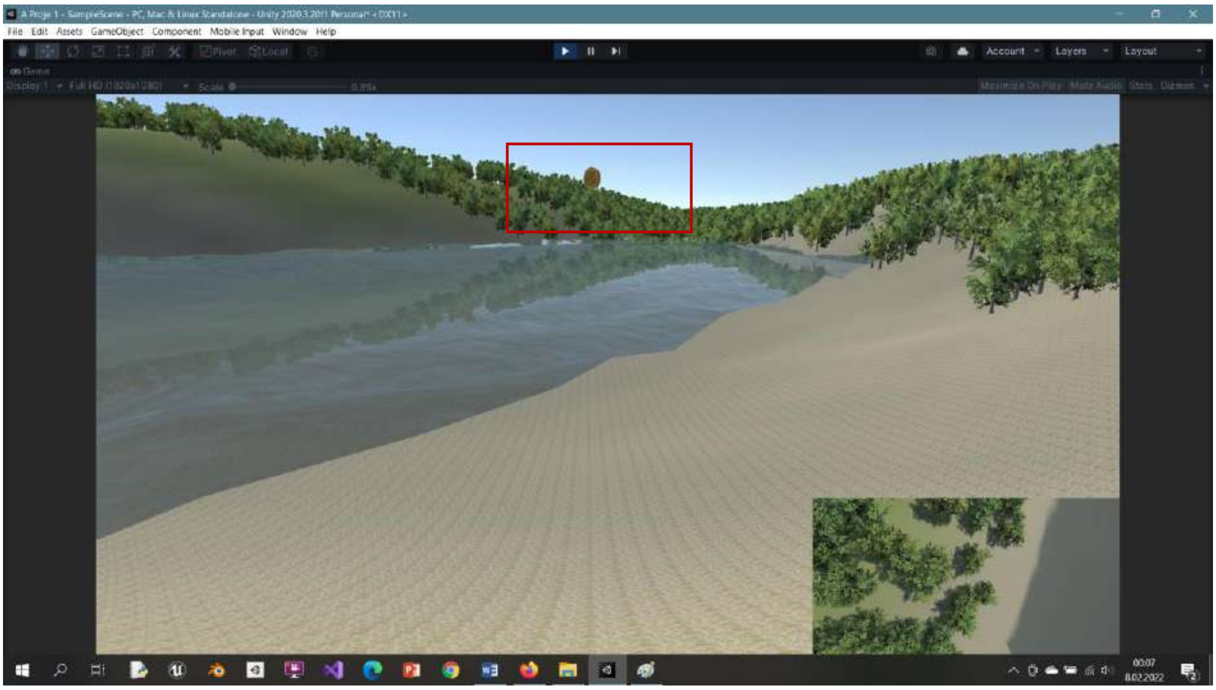
Let's hold the line on the **timeline** and bring it to 60'. 60 frames are equal to one second. The aim is to determine which position it will be in the 60th frame. In the example, it is planned to rotate around 360 degrees. For this reason, a rotation value of 359 degrees is written around the **Y-axis**. The animation can be seen by pressing the play button.



There is one last setting to achieve a continuous rotation instead of a **choppy** movement. This is achieved by **right-clicking** on the **anchors** in the first and last frames and selecting **Auto** in the window that opens.



In game mode, our gold will rotate around itself. You can find other types of animations by trying them.

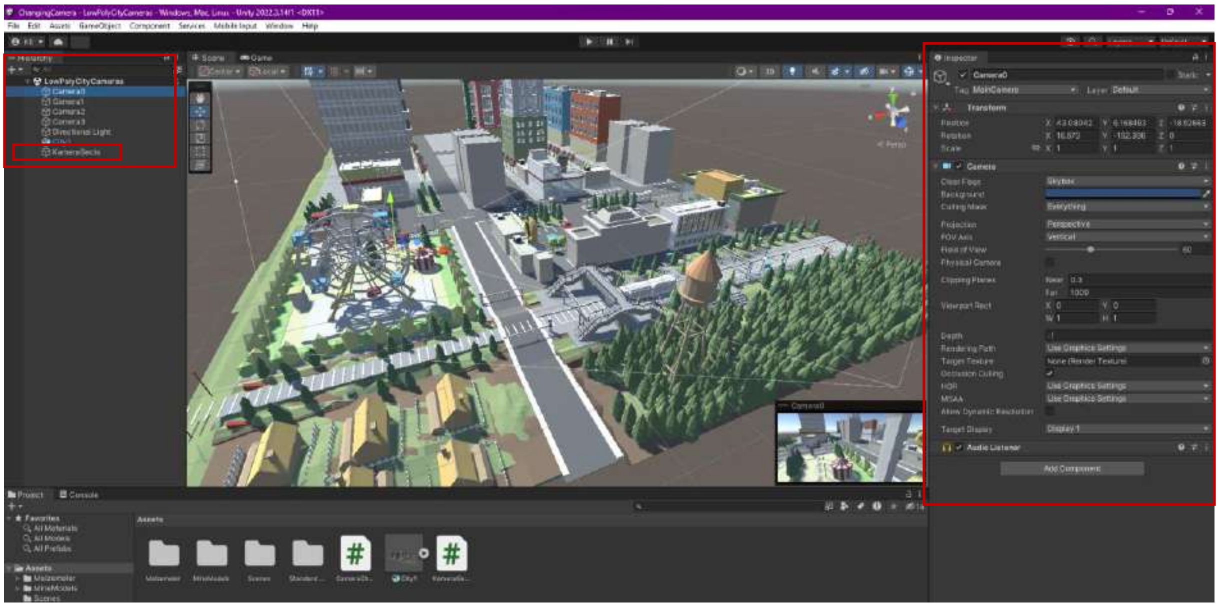


6.7.Switching Between Cameras in Unity

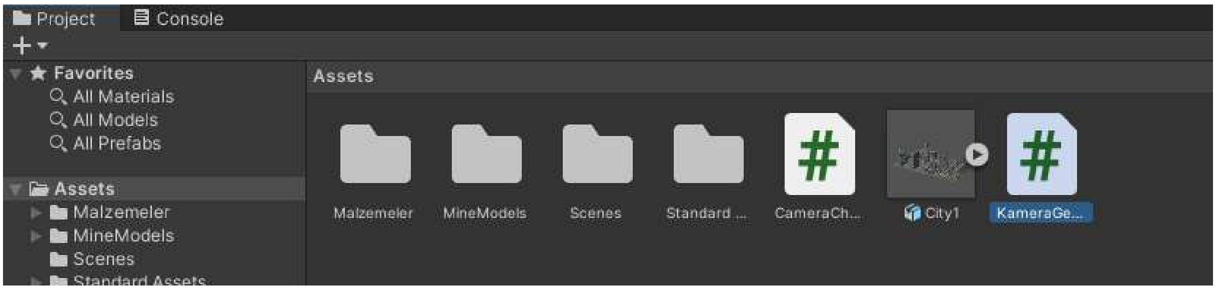
In scenes with multiple cameras, various methods can be applied to switch between cameras; **UI Button** applications, keying in camera numbers, navigating between cameras with a keyboard character, etc. In this tutorial, two methods and **two C# code files** will be shared.

6.7.1. 1st Method

Let's drag the low poly city file we got from Sketchfab to our project. We named the main camera **Camera0** (not required but for clarity). Let's add **three** additional cameras to look at the city from different angles by selecting Camera with the right mouse button in Hierarchy. In this application, the cameras are named **Camera1, Camera2,** and **Camera3**. Also, to open an **empty game object** in the same place, let's right click the mouse and select **Create Empty**. This object is named **KameraGecis**.



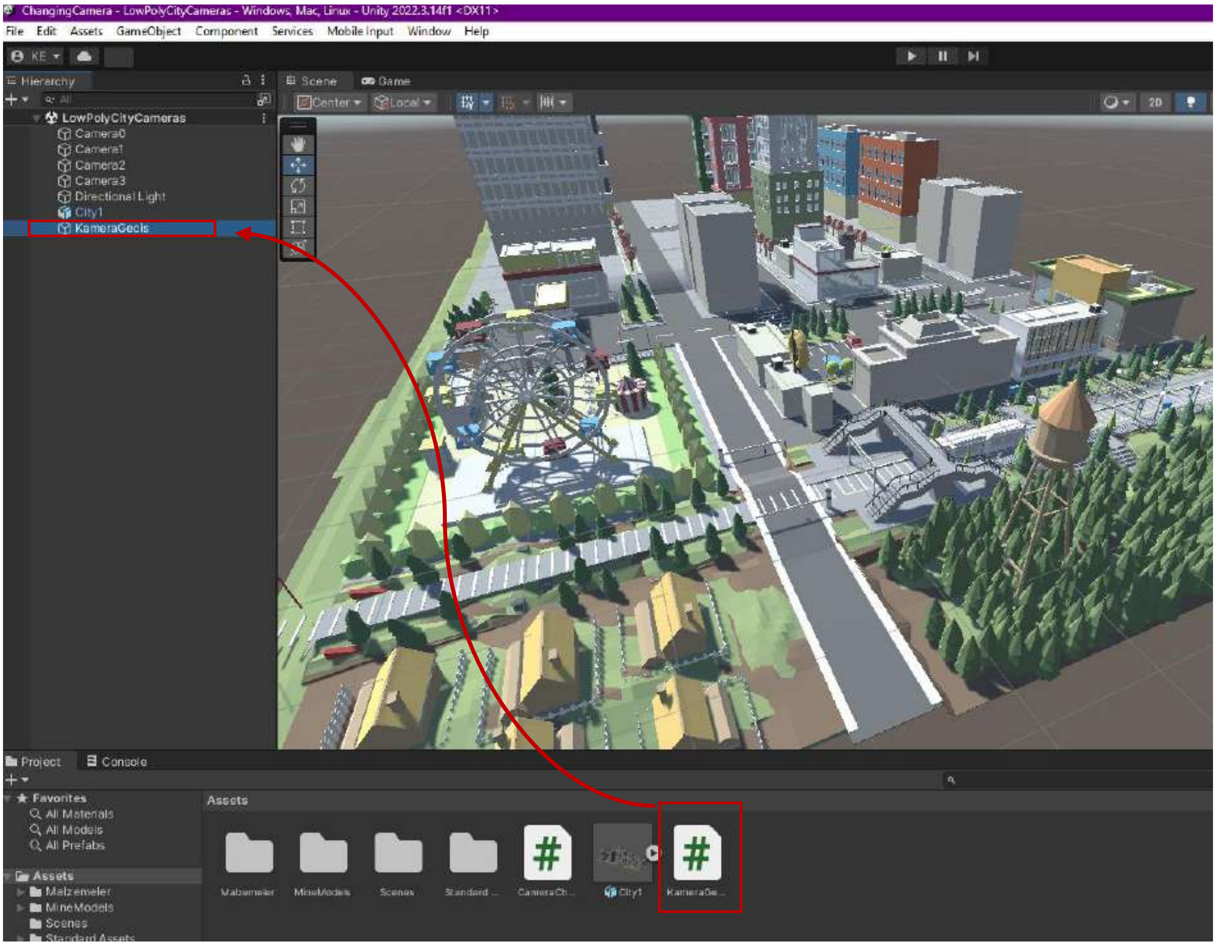
Let's create a C Sharp (C#) file named **KameraGecisleri.cs** with **Create>C# Script** in the Assets section. Now, write the following codes to this file.



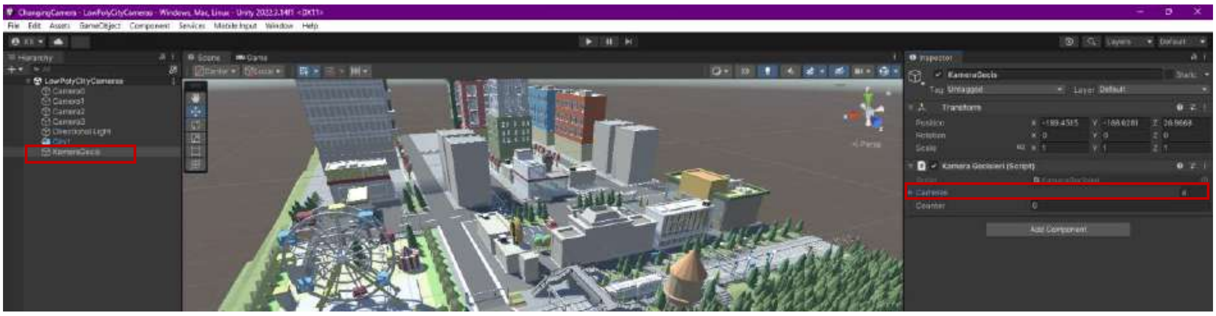
As an editor, if **Visual Studio** is installed with Unity installation, this editor will automatically open when the file is **double-clicked,** and codes can be written. If Visual Studio is not installed, you can open the **KameraGecisleri.cs** file with Notepad, WordPad, etc. and perform the writing process.

```
using UnityEngine;

public class KameraGecisleri : MonoBehaviour
{
   public GameObject[] Cameras;               // cameras array
   public int counter = 0;                    // the counter variable

   void Start()
   {
      foreach (var item in Cameras)
      {
         item.SetActive(false);   //make the cameras passive
      }
      Cameras[0].SetActive(true);
      counter++;
   }

   void Update()
   {
      if (Input.GetKeyDown(KeyCode.Tab))   // change the cameras with Tab key
      {                                      // make active the current camera
         foreach (var item in Cameras)
         {
            item.SetActive(false);   // make the camera passive
         }
         Cameras[counter].SetActive(true);   // activate the current camera
         counter++;                          // increase the counter by one

         if (counter == Cameras.Length)   // if the total number of cameras are reached
         {
            counter = 0;   // initialize the counter
         }
      }
   }
}
```

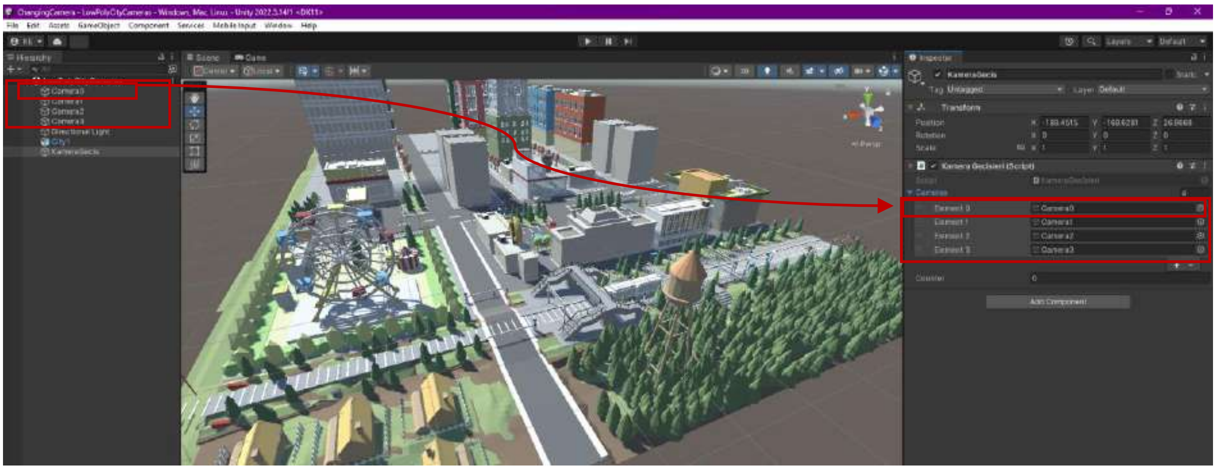
After the **KameraGecisleri.cs** file is written, let's drag and connect this file to the object we opened as **GameObject** and named **KameraGecis**.



According to the code content, the initial value for **KameraGecis>Inspector>Cameras** is zero. Let's make it **4**.



When the number is set to 4, a camera list will open under **Cameras**. Let's drag and match all our cameras in the **Hierarchy** here so that they match exactly.

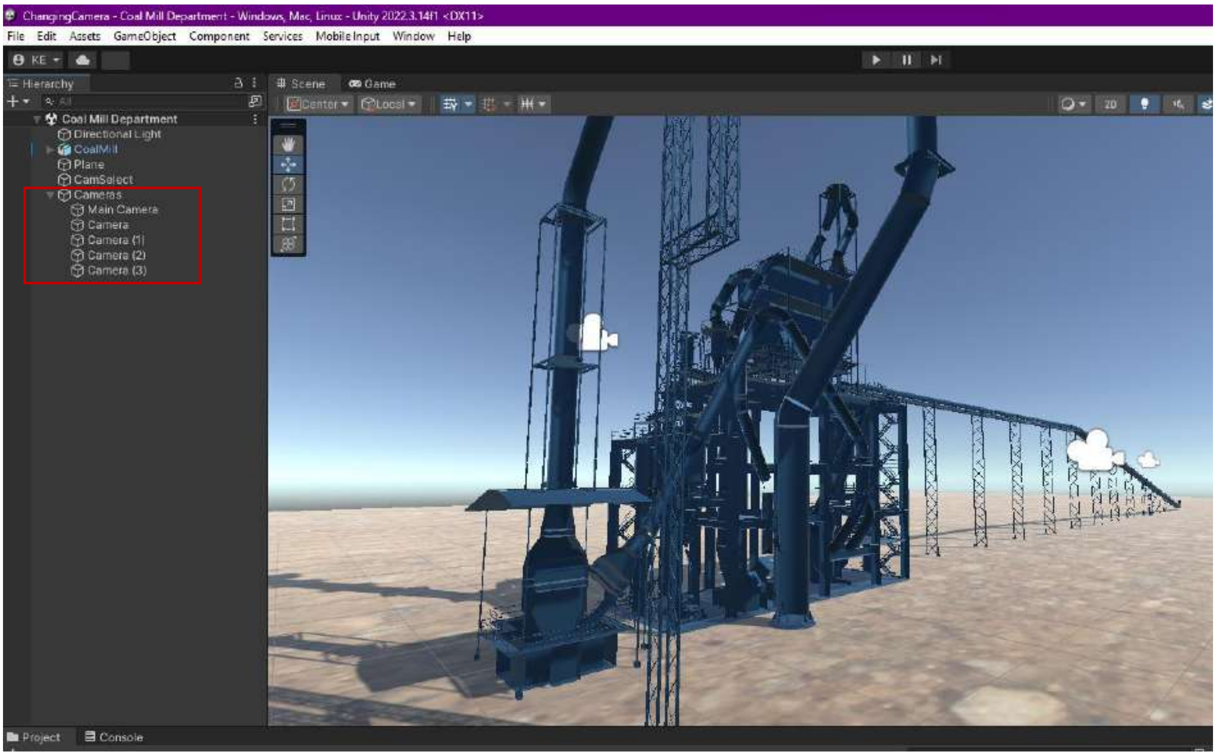


When we run it in **Play** mode, it will now be possible to switch between **4** scenes in order with the **Tab** key.

In this method, the number of cameras is <u>not</u> specified in the C# Script codes. However, the codes are based on the principle of specifying the number of cameras in the **Inspector** after connecting to the empty **GameObject** (**KameraGecis**) in the **Hierarchy** and dragging and matching the cameras to their places in the **drop-down list**.

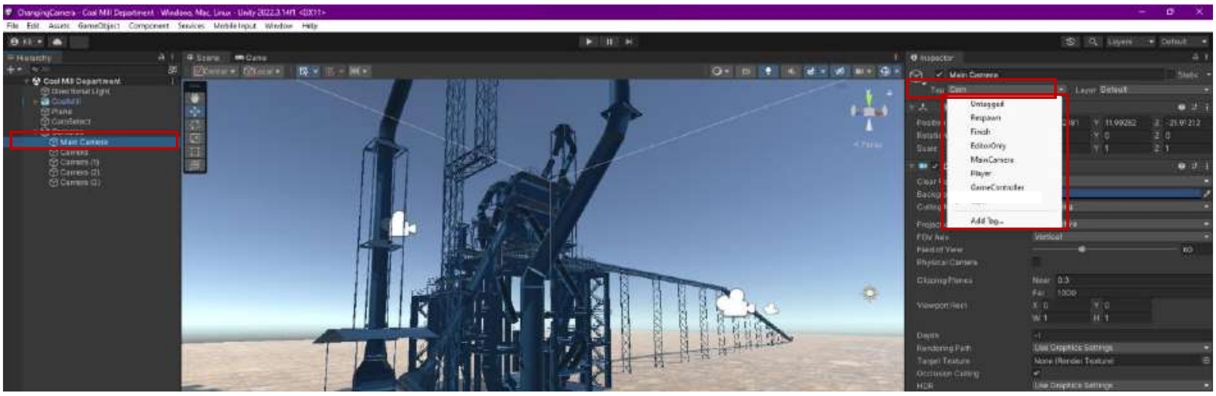
6.7.2. 2nd Method

In this training application, a mining field facility was used, and a total of **5** cameras with different perspectives were placed. The cameras were dragged under the **empty object** called **CamSelect**, which was opened with **Create Empty** in **Hierarchy**, and gathered under one roof (this is not necessary; they can remain separate).

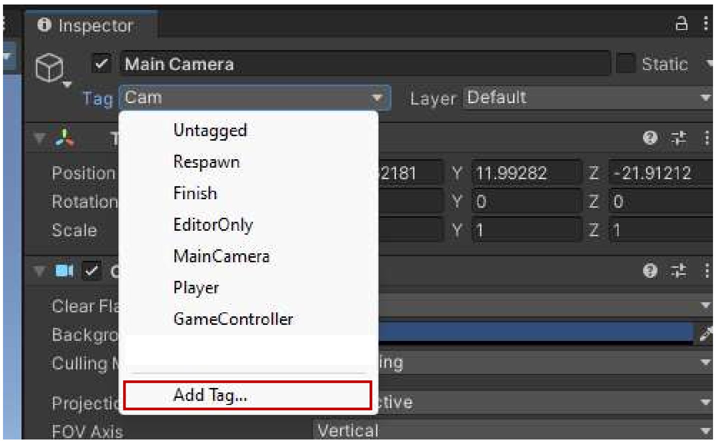


The **Cam** tag should be given to the **Tag** section of the cameras, as it is defined that way in the codes that will be shared shortly.

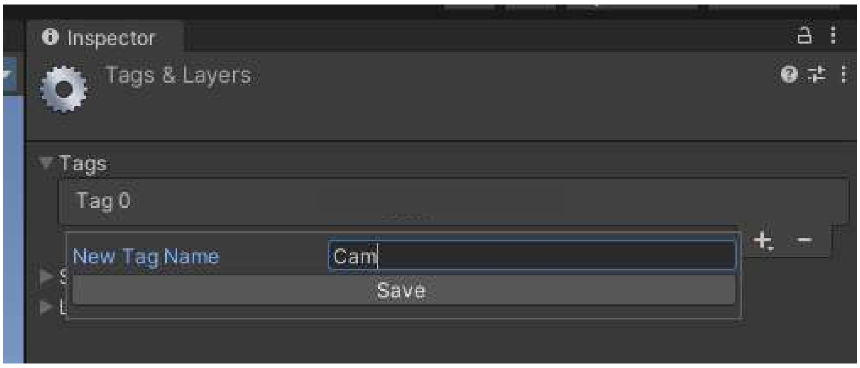
To do this, these operations must be performed in order on the cameras in the **Hierarchy**; first, let's select the camera and open its **Tag** in the **Inspector**.



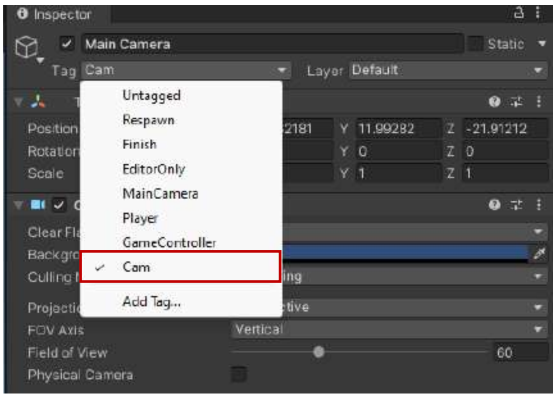
To add a **Cam** tag that does not exist here, select **Add Tag**.



Now, define our **Tag** by pressing the **+** key. Write **Cam** here and save it with **Save**.



Now all our cameras can be given the **Cam** tag that appears in the **Tag list**.



Also, in this application, an empty **GameObject** is created with **Hierarchy>Create Empty** and its name is changed to **CamSelect**.

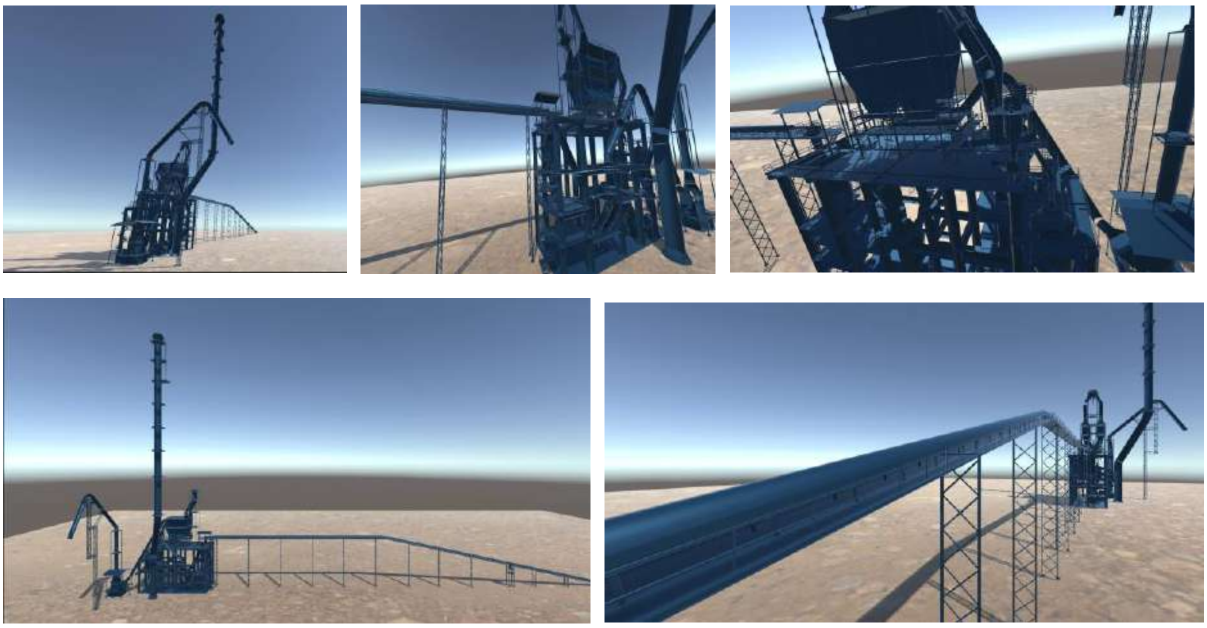
A C# Script file called **CameraChanging.cs** has been created in the Assets section.

As before, this file was opened in the editor and the following code lines were written.

```
using UnityEngine;

public class CameraChanging : MonoBehaviour
{
   public GameObject[] Cameras;              // cameras array
   public int counter = 0;                   // the counter variable

   void Start()
   {
      Cameras = GameObject.FindGameObjectsWithTag("Cam")   //find the cameras with Cam tag
   }

   void Update()
   {
      if (Input.GetKeyDown(KeyCode.Tab))   // change the cameras with Tab key
      {
         foreach (var item in Cameras)        // make the cameras passive
         {
            item.SetActive(false); //kameralrı pasifleştir
         }
         Cameras[counter].SetActive(true);   // make active the current camera
         counter++; //sayacı artır             // increase the counter by one

         if (counter == 5)   // if the counter is 5
         {
            counter = 0;   // initialize the counter
         }
      }

   }
}
```

Next, let's drag and drop the **CameraChanging.cs** code file to our **CamSelect** object in **Hierarchy**.

Our application is ready. Now, we run it in **Play** mode.

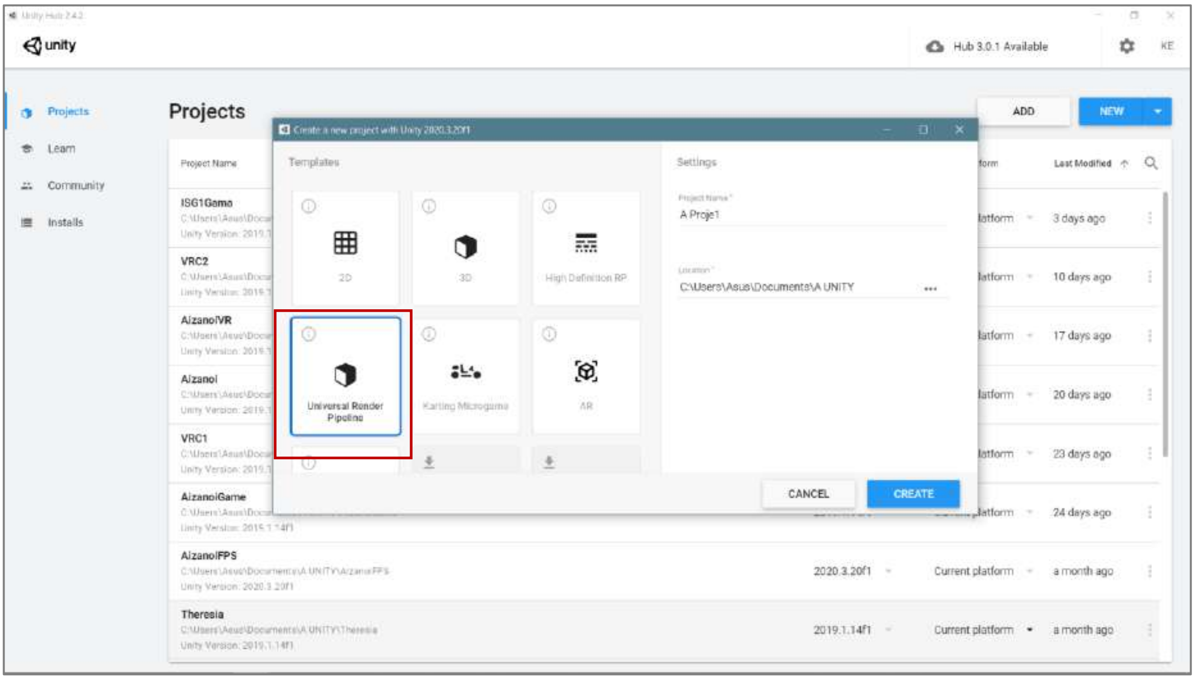


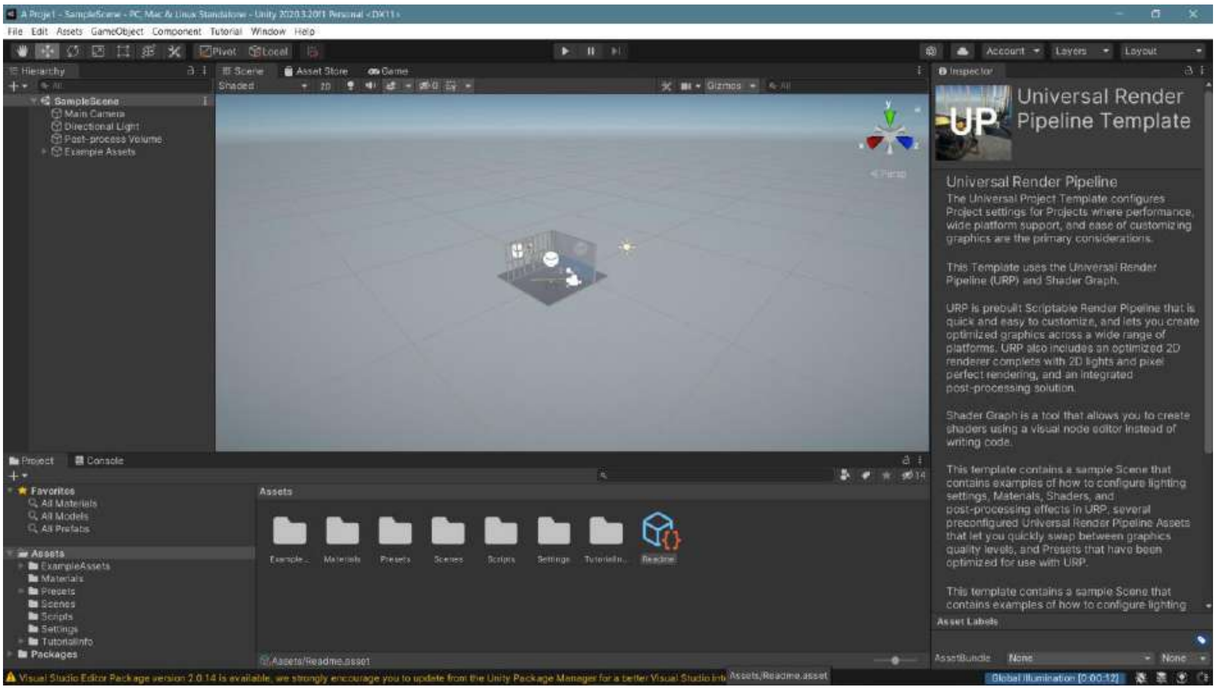
In this application, the codes are based on placing 5 cameras. For fewer or more cameras, the codes will need to be changed. In the first method, there is no limit on the number. In this method, it is not necessary to connect the cameras to the list.

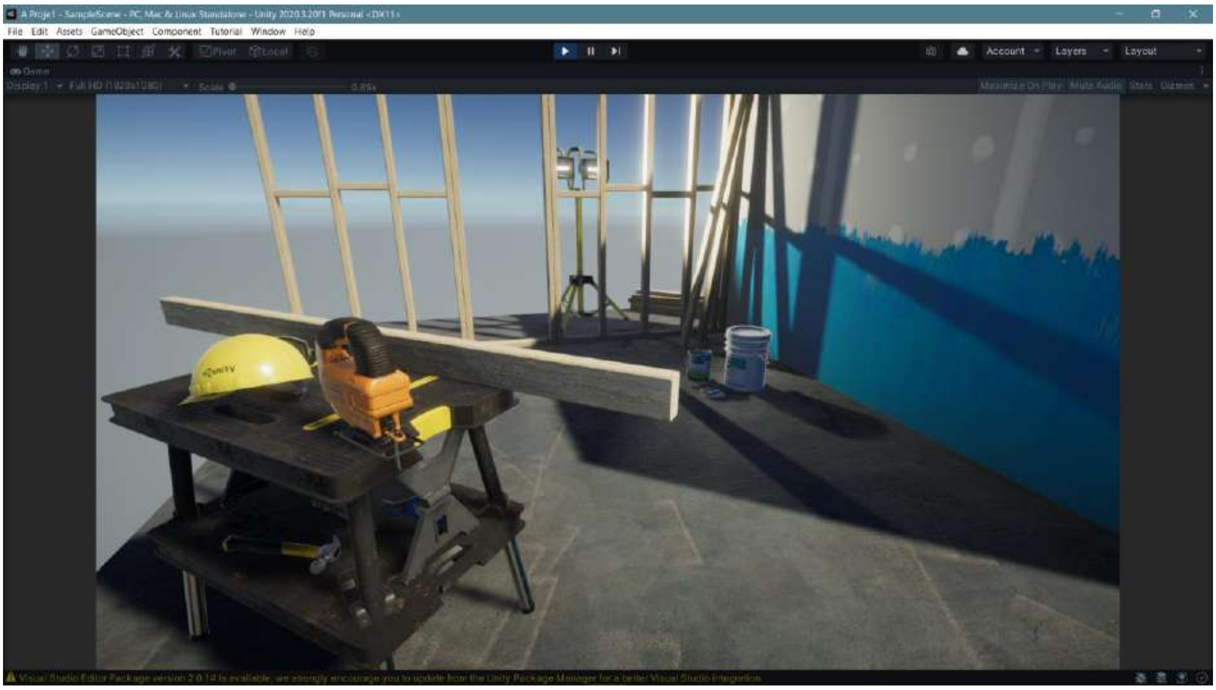
7. LIGHT AND TEXTURE WITH URP

7.1.Universal Render Pipeline (URP) – Post Processing Volume- Glow Effect

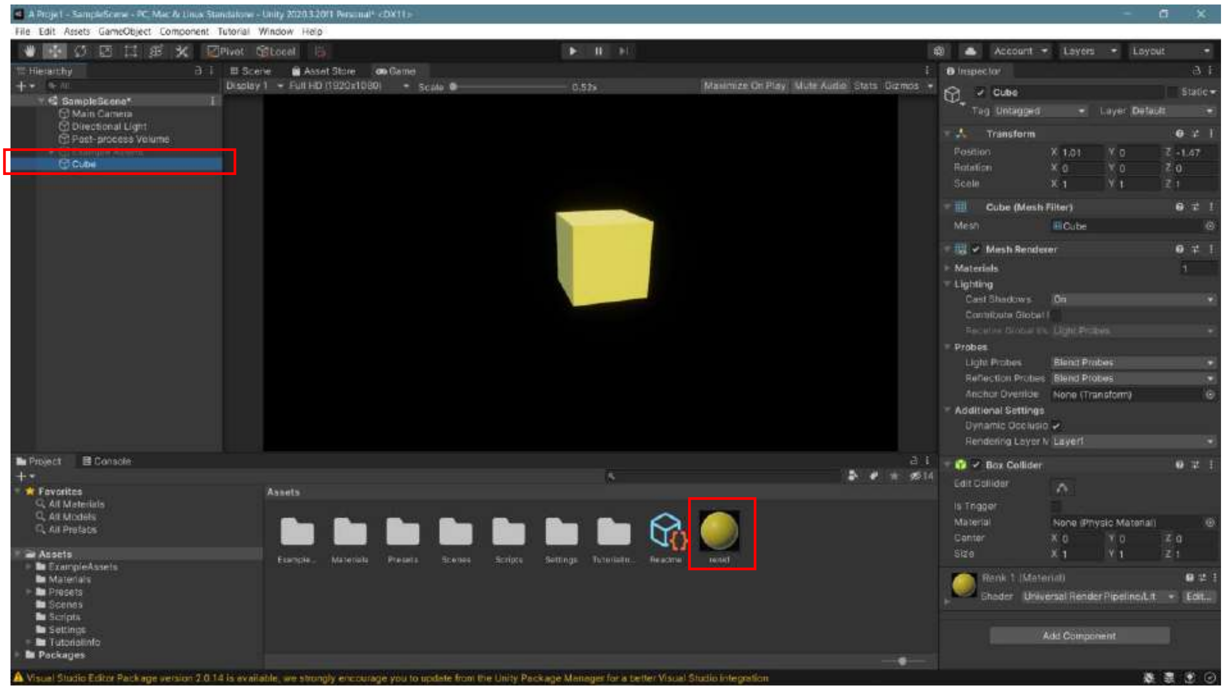
Universal Render Pipeline (**URP**) is a template in which special lighting effects are defined. If the project is planned to use URP, the process will be faster and easier if the relevant selection is made in **Unity Hub**. When the project is opened, a template scene is ready. Scene and game mode images are below.



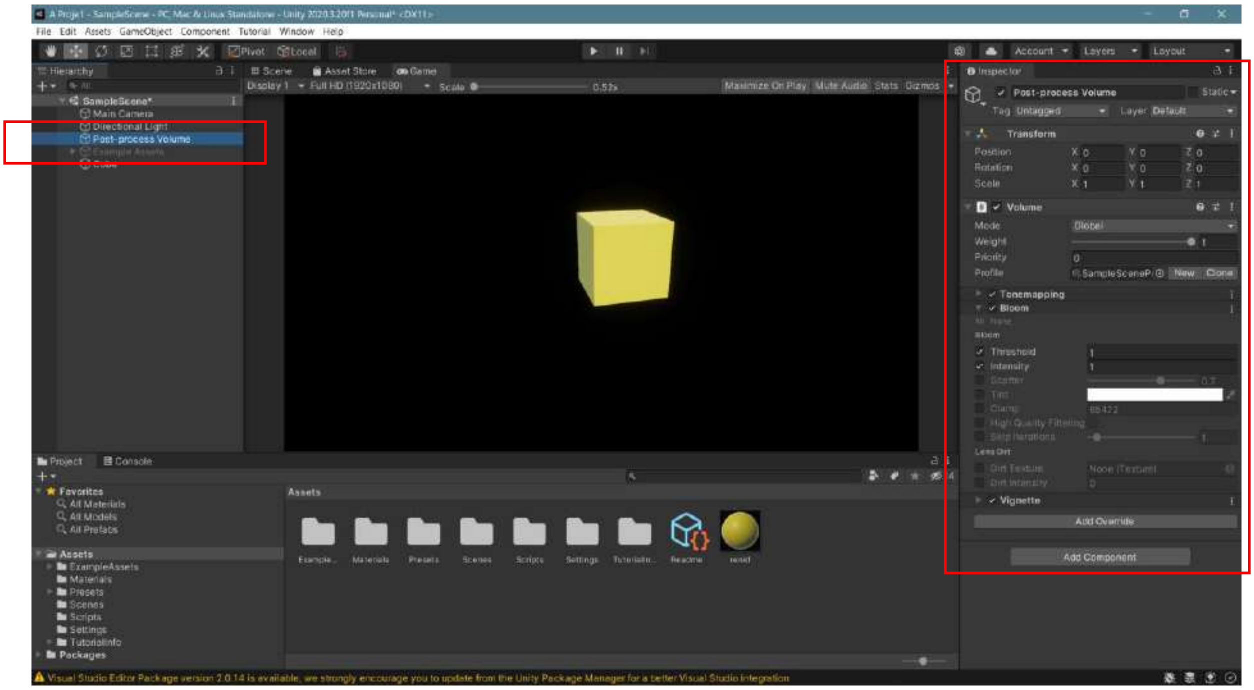




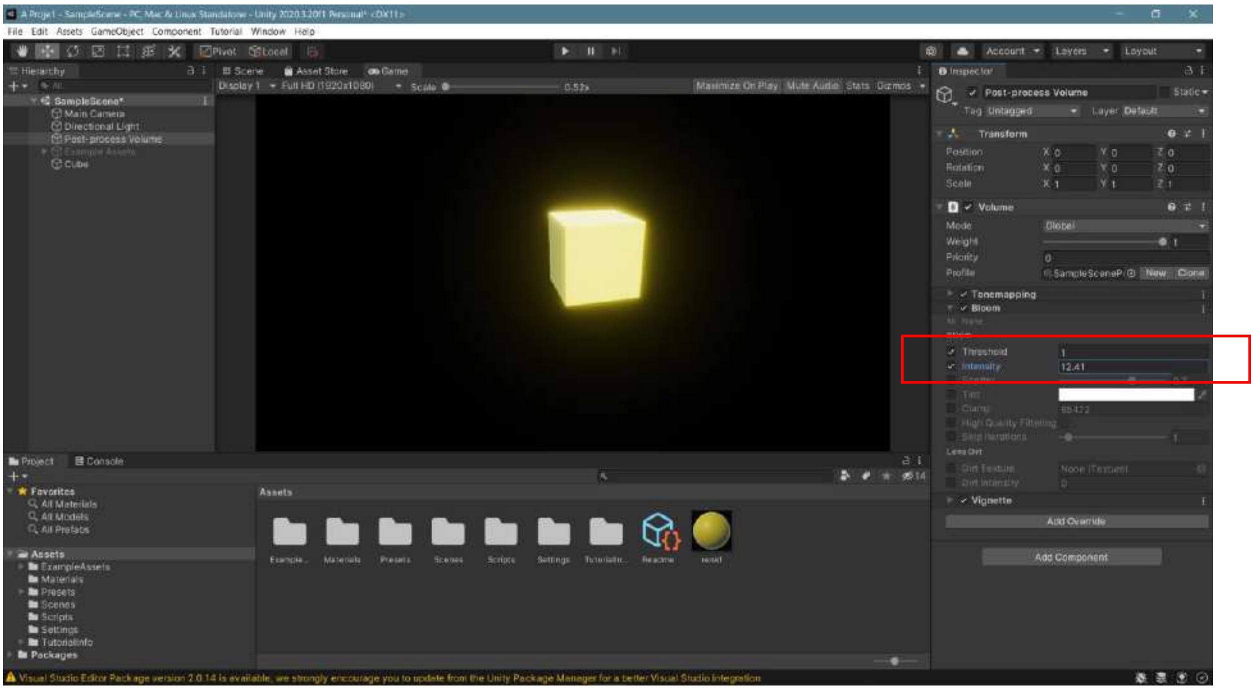
For the **glow effect** example work, let's delete or make passive the **GameObject** named **Example Assets** that creates the workshop seen in the scene. Thus, we can easily apply our design in the preset scene. After Example Assets are closed/deleted, let's add a **cube** to the empty scene. Now, create a **Material** for the cube and give it a yellow color.



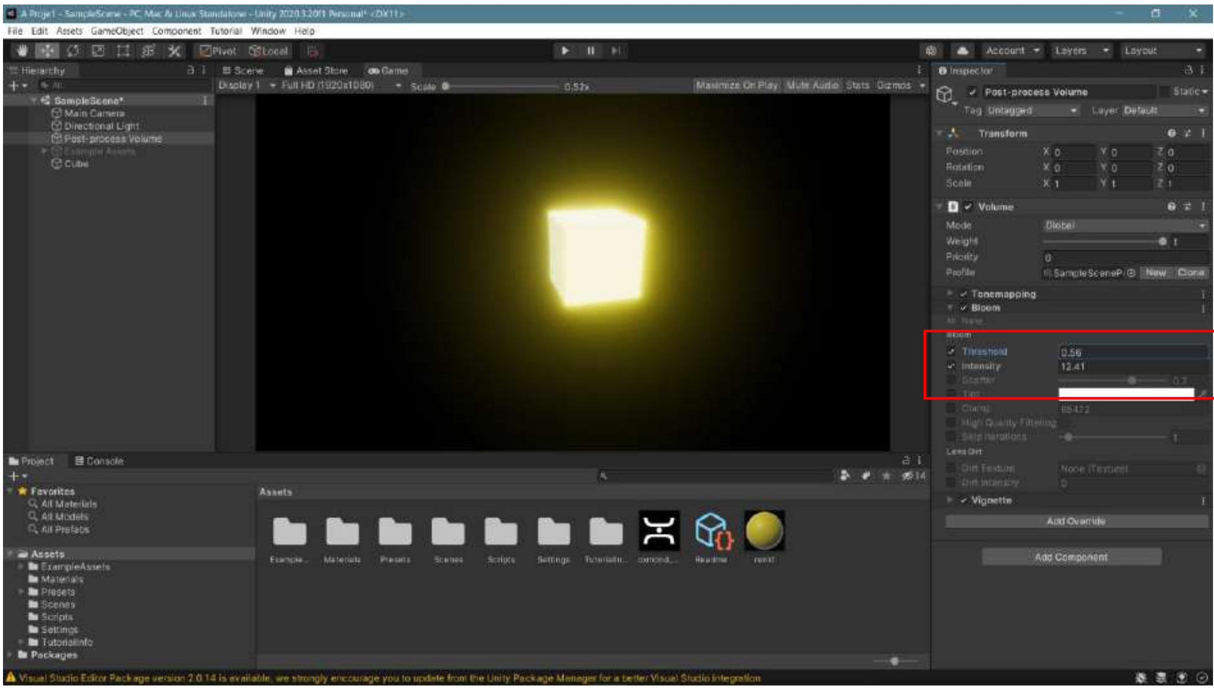
The most critical object in the scene is the **Post Process Volume**. The most critical setting on the Inspector is **Bloom**. The Intensity setting under **Bloom** is used to achieve the **glow effect**.



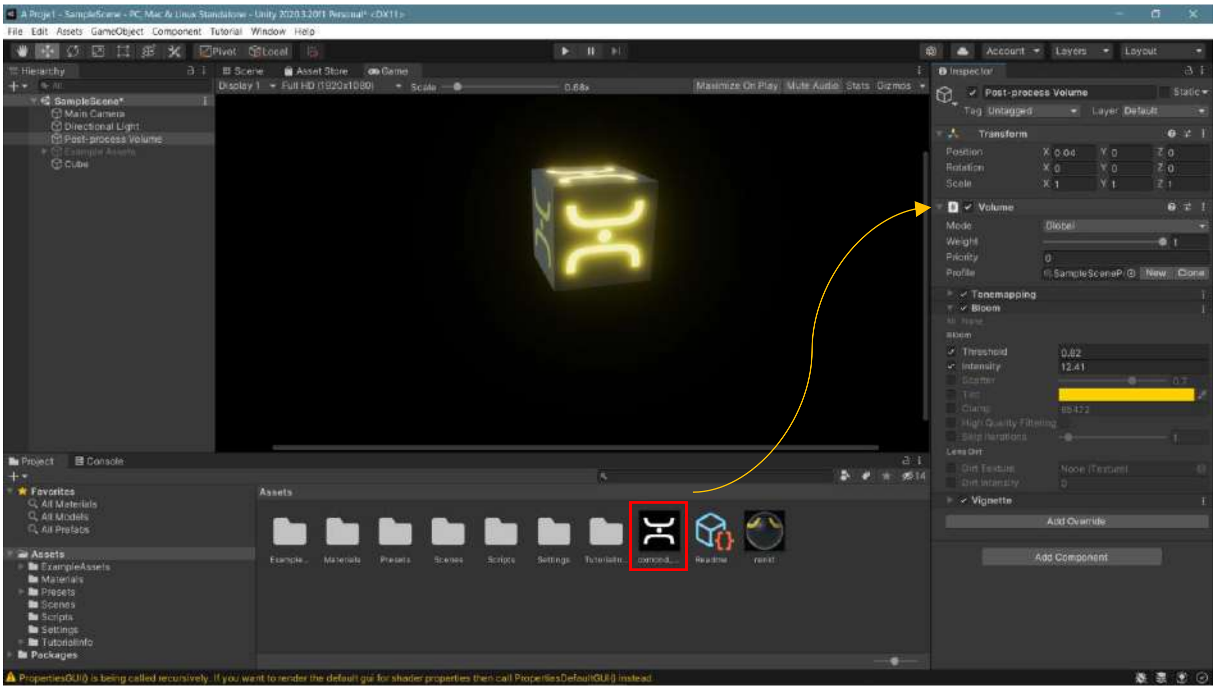
Intensity - when the intensity is increased, the result will be reflected on the screen.



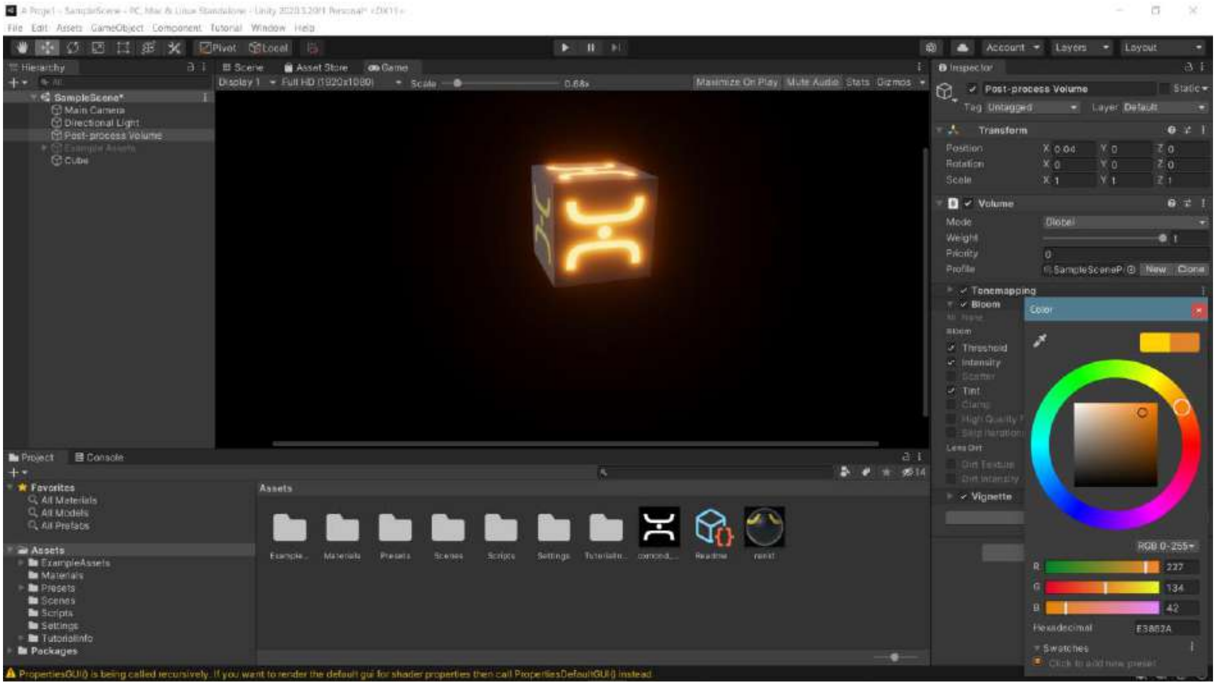
The brightness will increase as the **threshold** setting is reduced.



You can try and see which setting will give you the results you want. If you add a JPEG file with a dark background under Assets and add the yellow material to the **Base Map** section, the result will change dramatically.

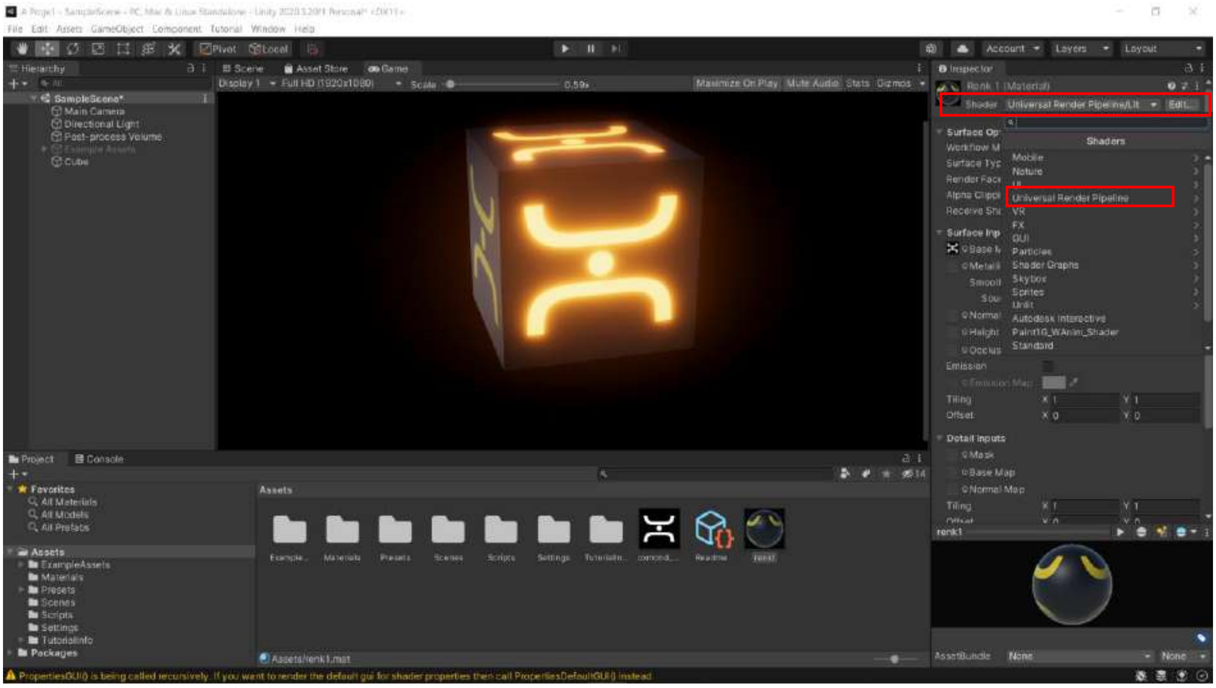


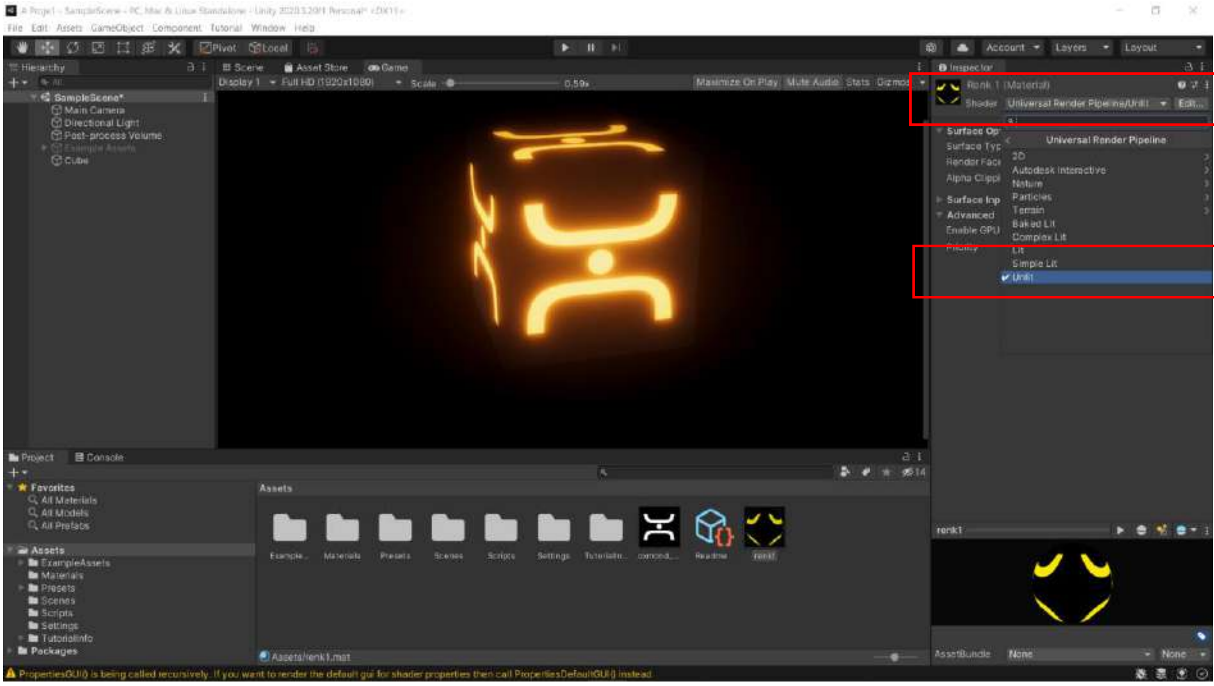
Observe the halo formed around the cube with the tint settings and colors.



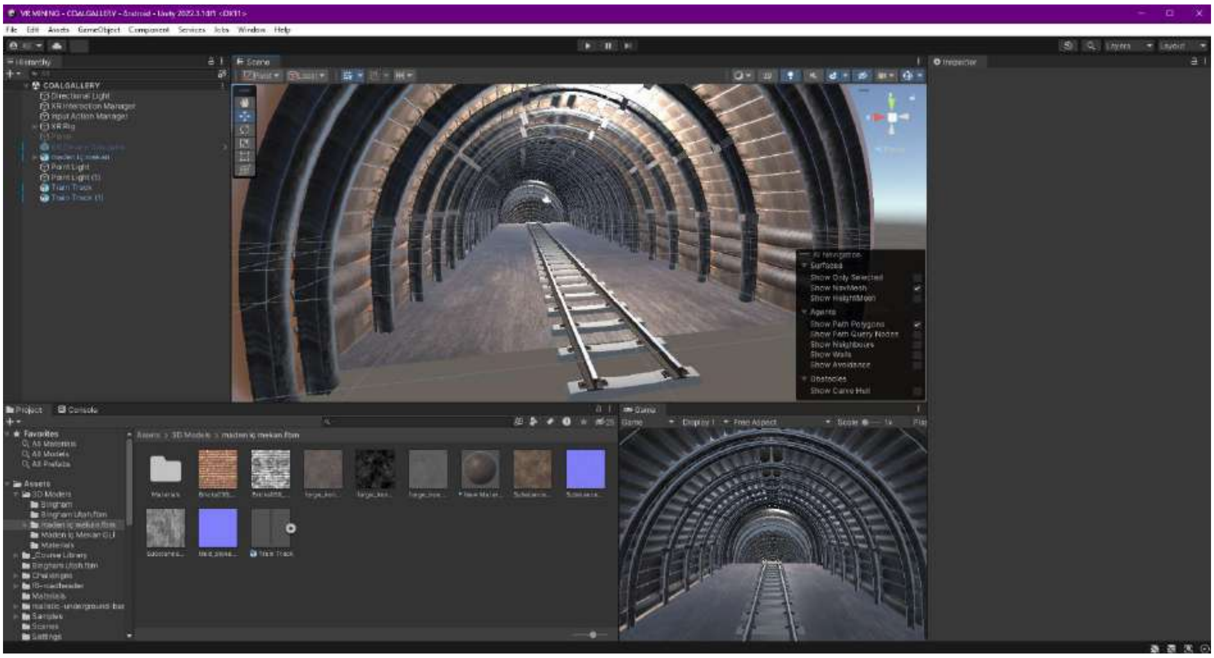
For the **non-glare** surface on the left side;

Let's select the color material. Selection is made in the order **Shader>UniversalRenderPipeline>Unlit**.





Light effects can also be used in a coal mine gallery.

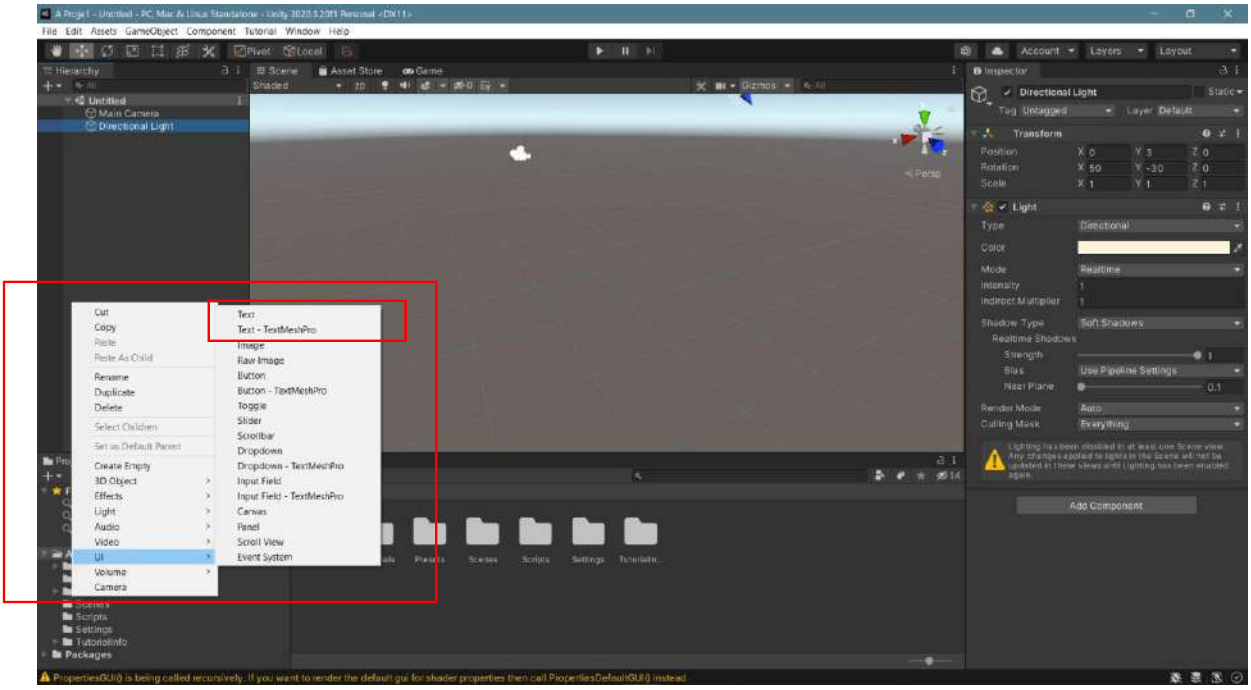


8. USER INTERFACE (UI)

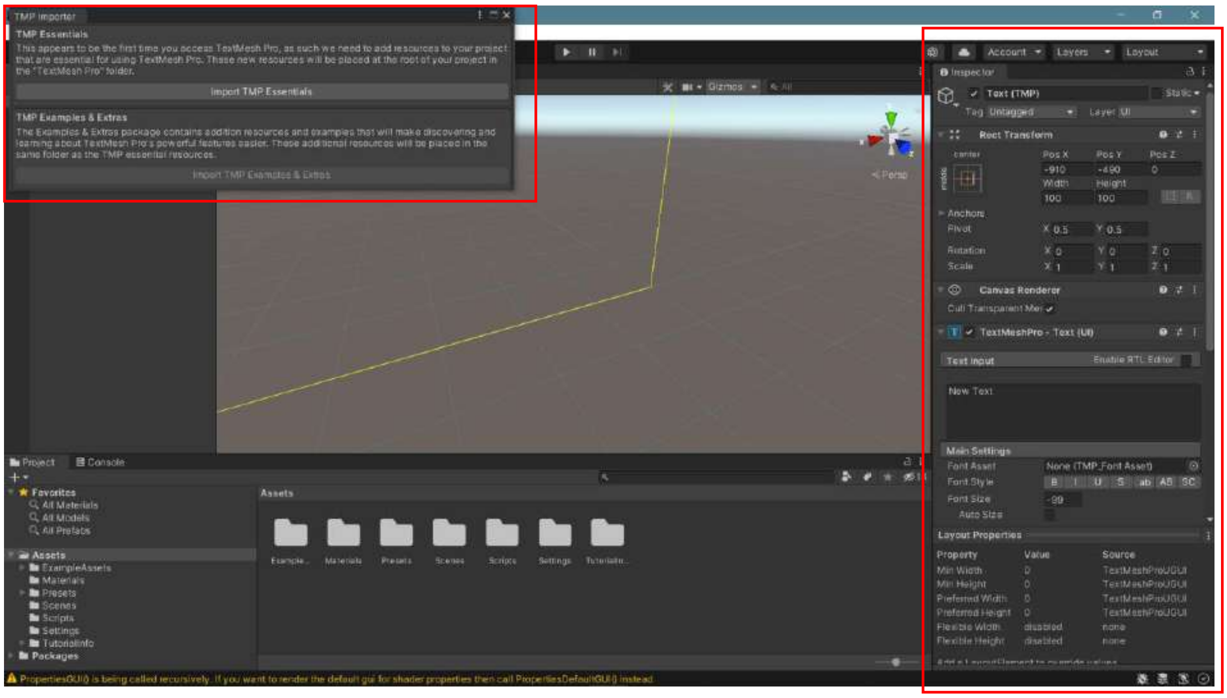
8.1.UI Text-Button-World Space- Interface Objects-3D Texts

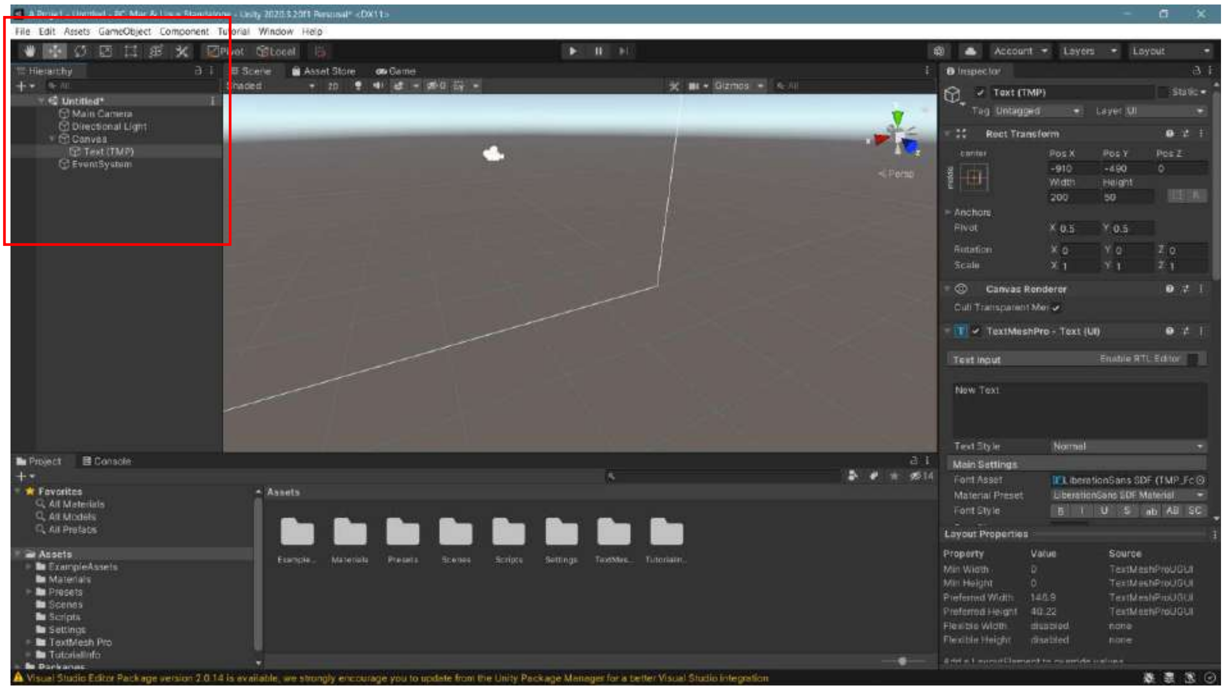
UI-User Interface-Interface is an important object that will interact with the user on the screen. Many applications are directed by a menu and its submenus. Therefore, with **UI, Text, Text-TextMeshPro, Image** etc. options will be able to do this design.

When you select **UI** with the right mouse button under **Hierarchy**, the submenu that is connected will open. The most used object is **Text**, and the more qualified text-writing object is Text-**TextMeshPro**.

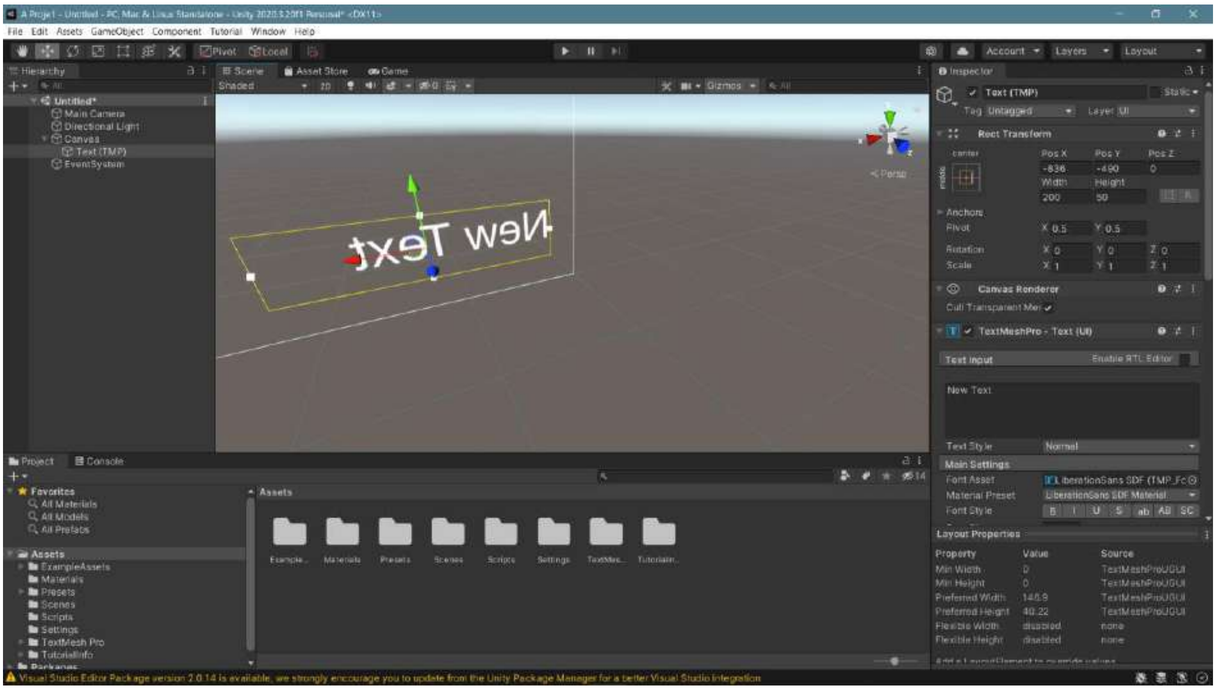


Let's select **TextMeshPro**. Then click **Import TMP Essentials** and **Import TMP Examples & Extras**. The parameters related to writing will open in the **Inspector**.

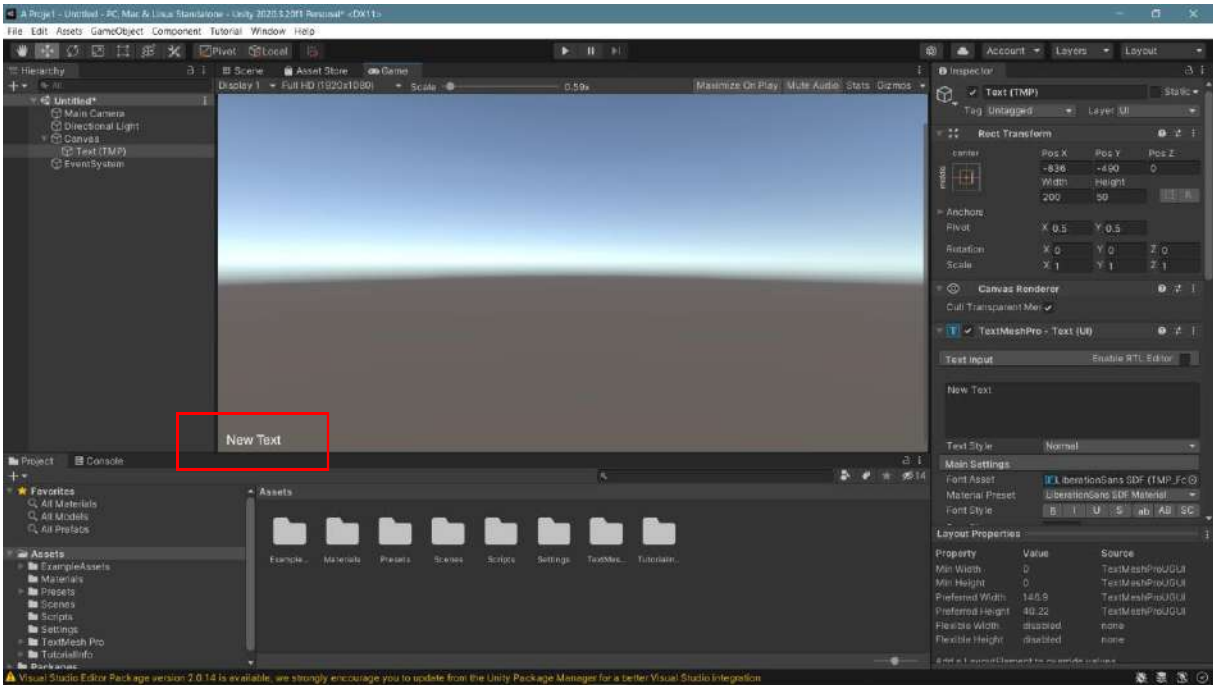


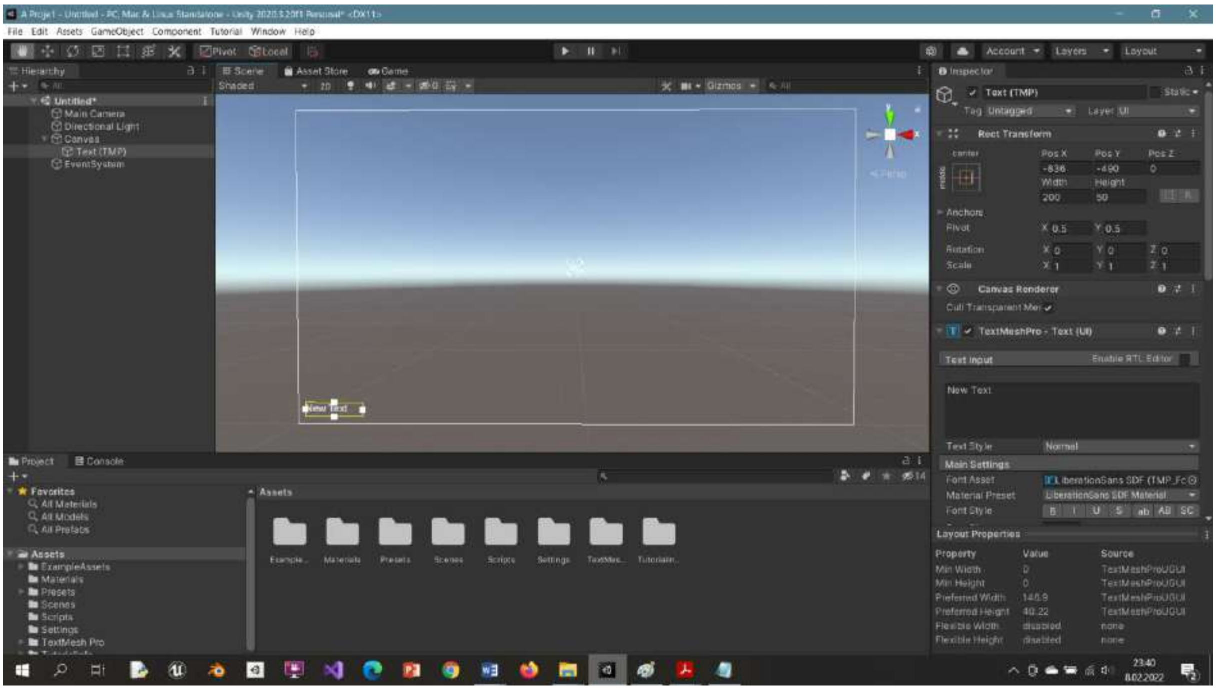


A **frame** that will cover the entire screen, which is actually **2D**, and a text called **New Text** have been created. With an object added to the **UI**, two more objects called **Canvas** and **EventSystem** are automatically added. **Canvas** is the frame of our screen. **EventSystem** is the object related to the operation to be performed. When we come back, we realize that this frame is very large. In fact, the frame is a 2D structure that can be considered independent in the 3D scene.

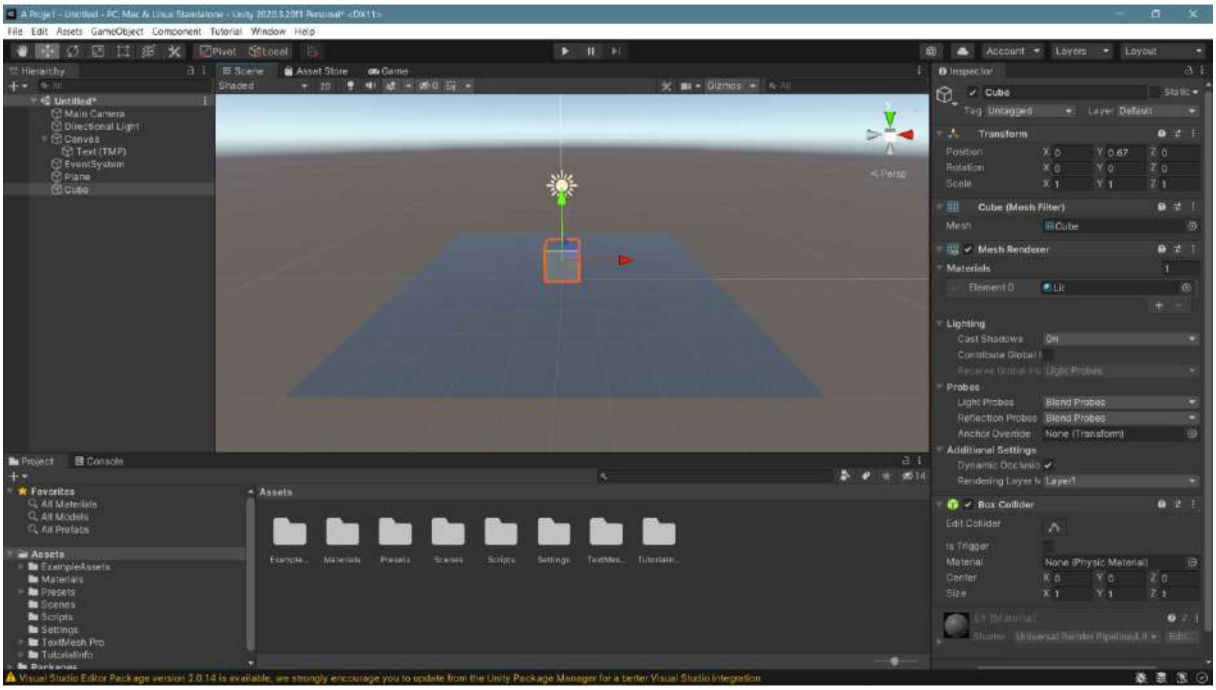


This 2D frame and text that appears to be huge is actually a small text in the lower left corner of the screen.

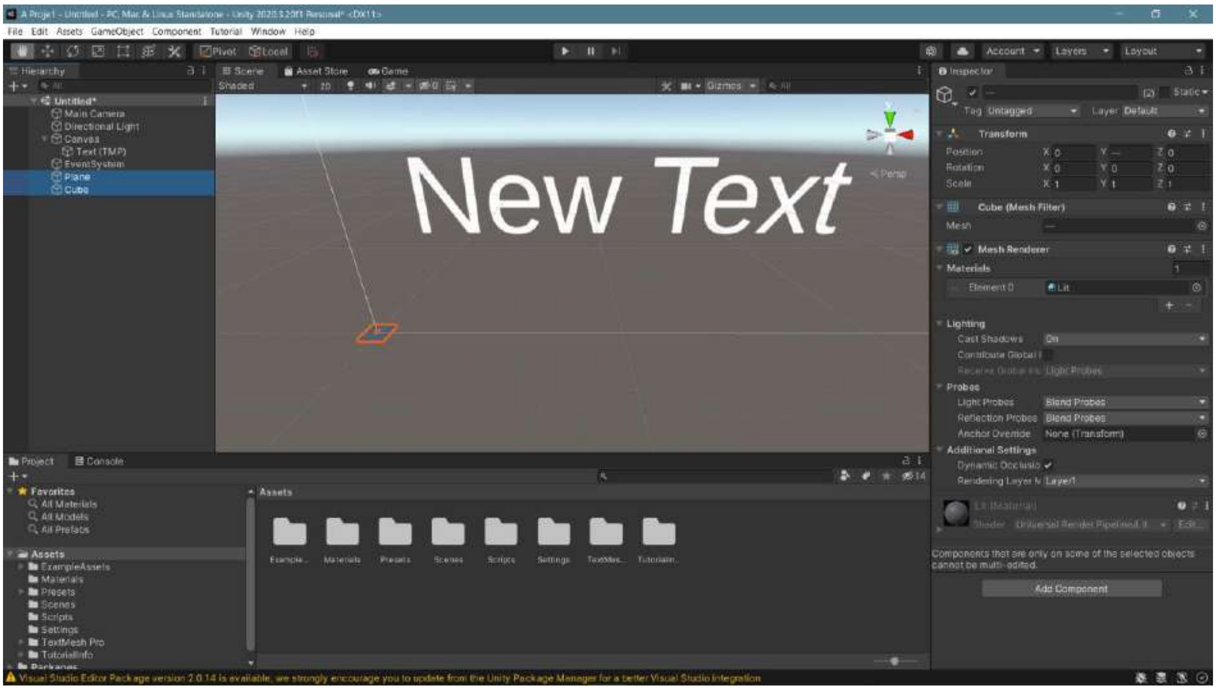




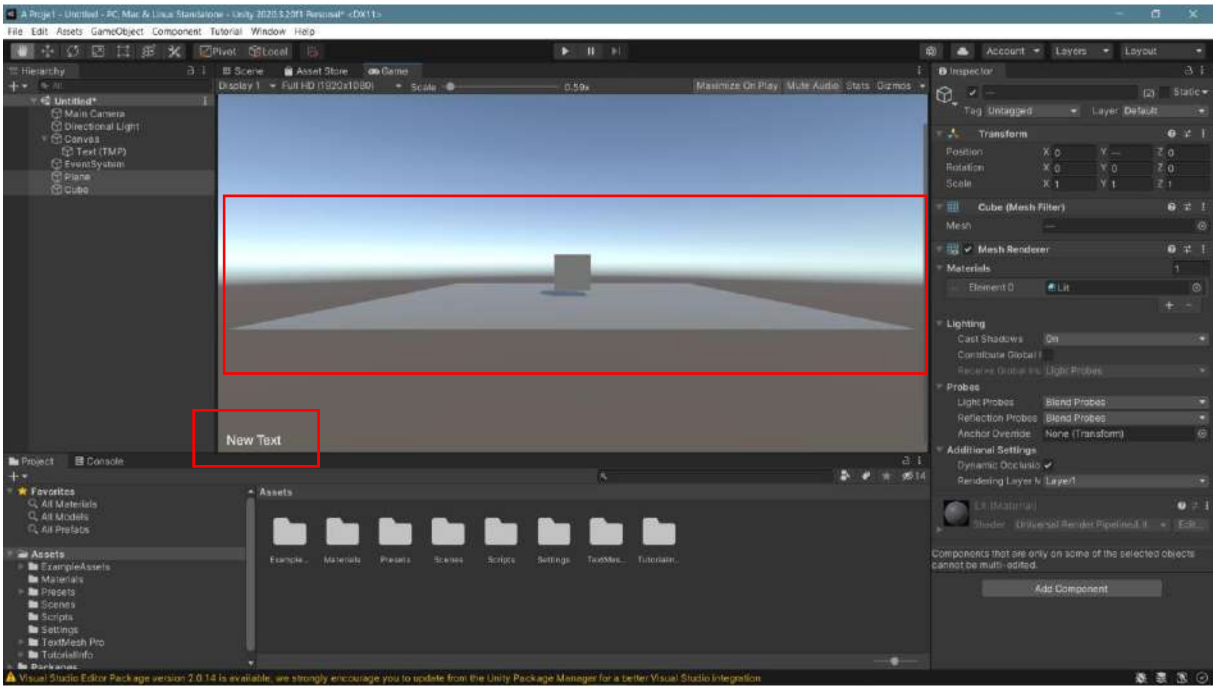
Let's add a **Plane** and a **Cube** to our scene.



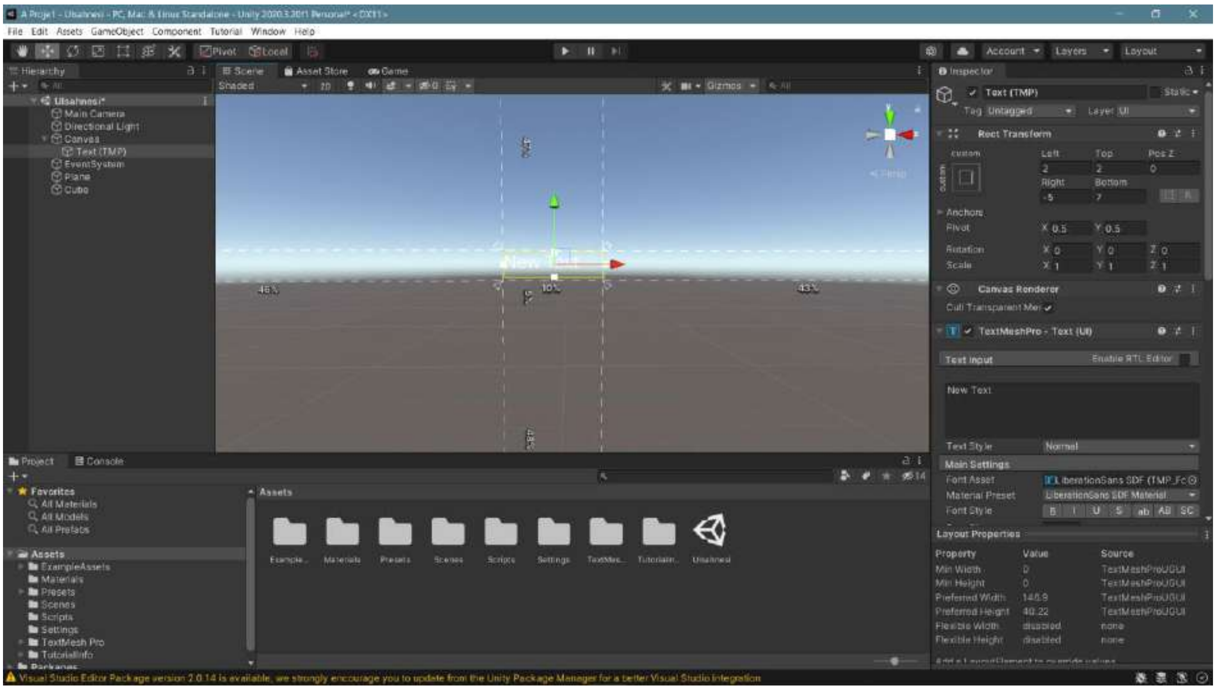
When we pull back, see where these objects are located in the **UI text** structure.



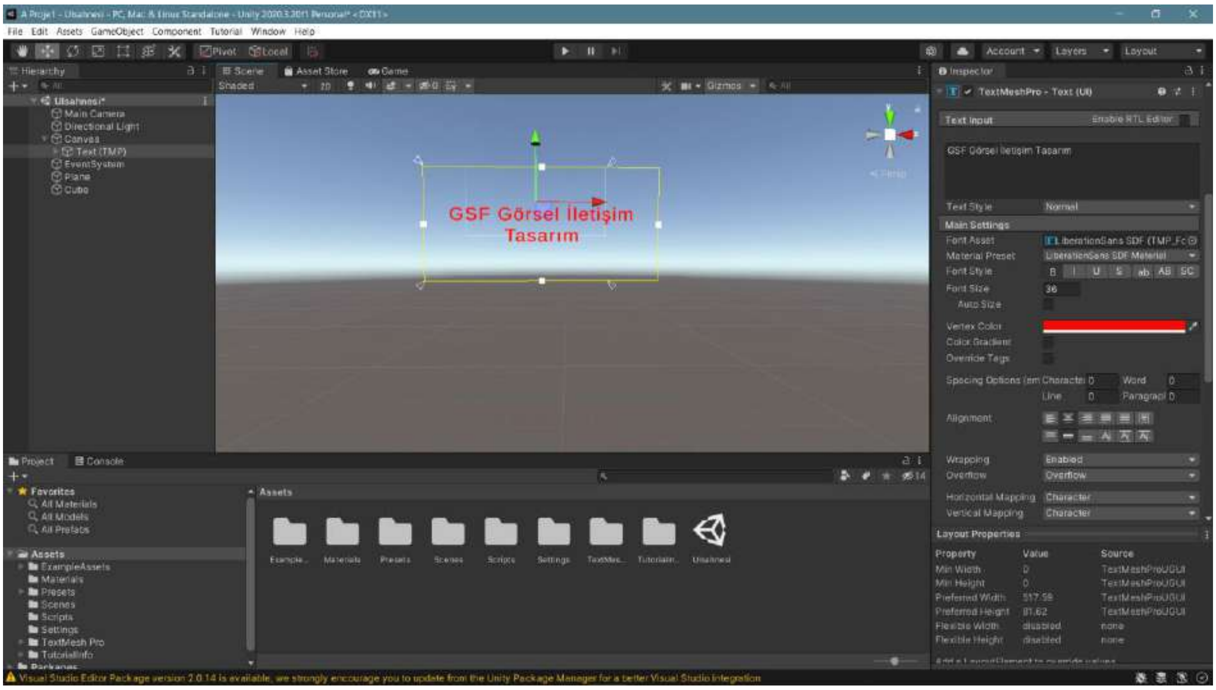
However, in **Game** mode, in the same scene, we understand that the **UI** structure is a 2D object on the screen, while other objects are in 3D structure but in a different size.



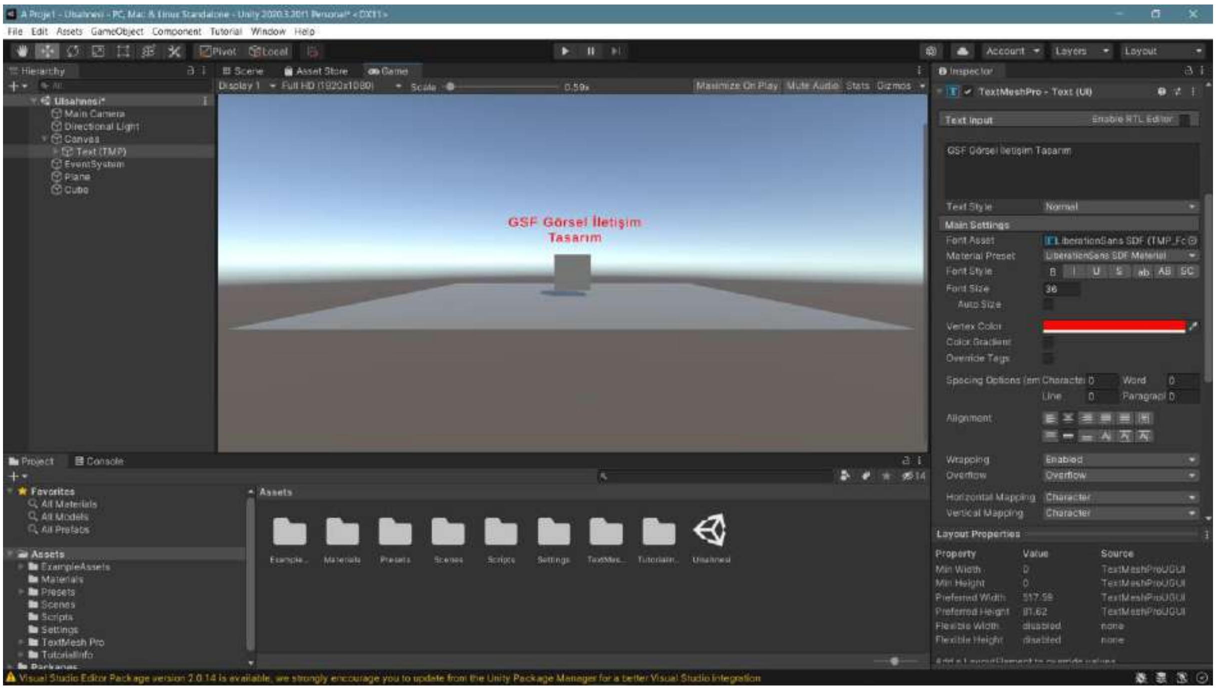
Now let's drag the **New Text** to the middle of the stage and place the anchors around it.



Now, move the text to the middle of the stage and change the text and color in the **Text** Input section.



The image in game mode will look like this.

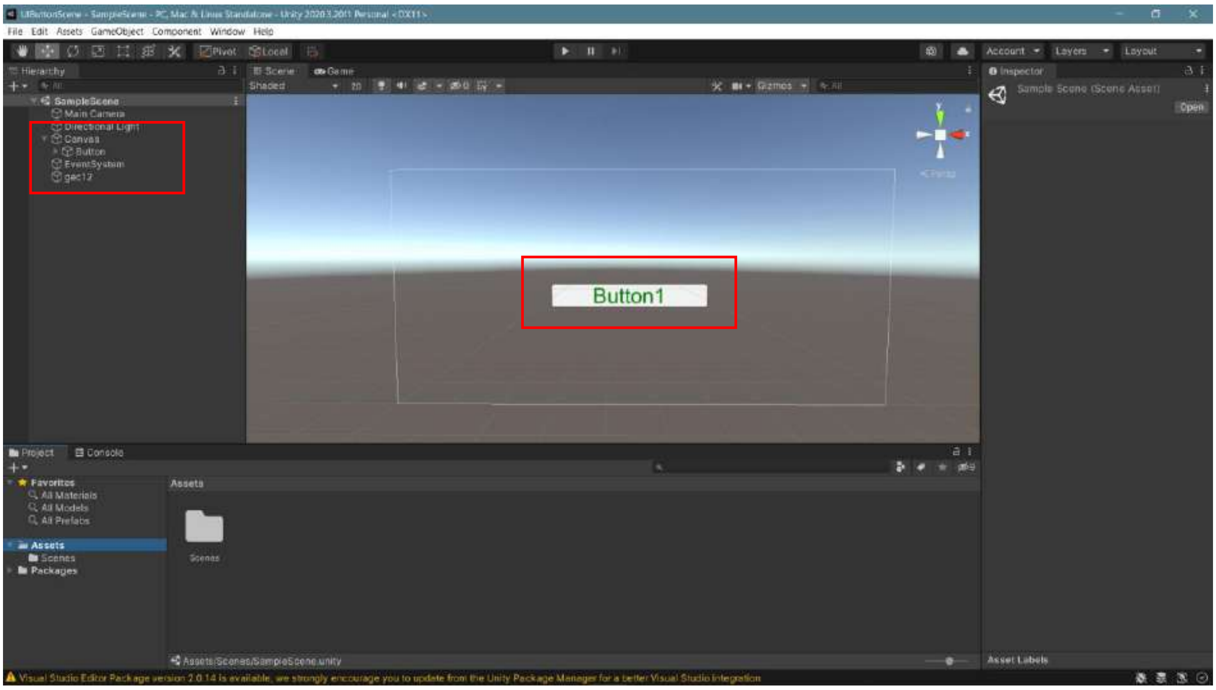


8.2.Switch between scenes-UI Button

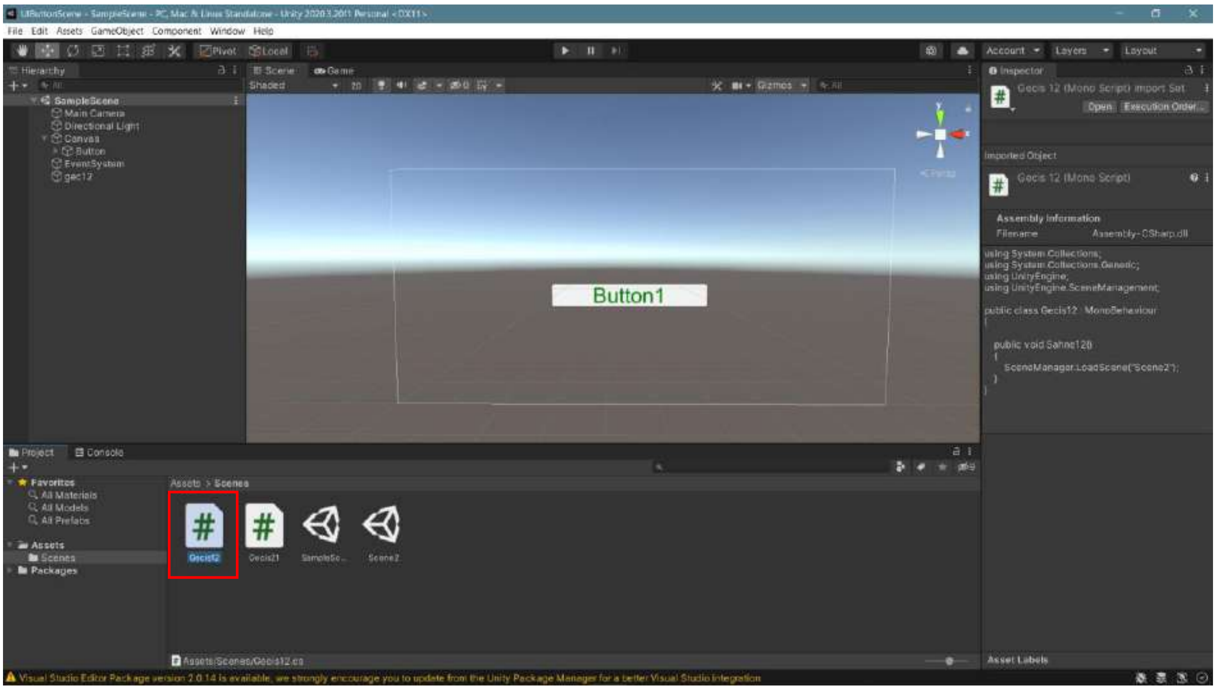
Let's assume that we added a button to our scene with the **UI Button** and that we want to add a **new Scene** to our project and switch to the new scene with this button. We can see the most practical and fastest way of doing this in an example study.

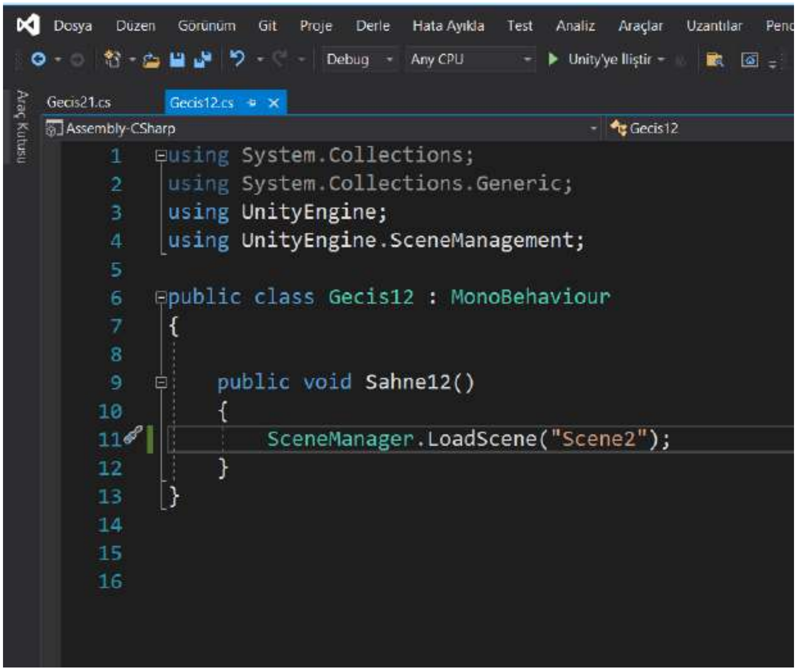
Now, continue without changing the name of our scene named **SampleScene**. Let's add a **UI>Button** here.

The **Canvas** and **Text** section under the **Button** has also been created. Change the **Text** color and size. Also, create a **GameObject** in **Hierarchy**. Here we set its name as **gec12**.



Now, create a **C# script** in the Assets section. Here we name it **Gecis12**. When we double-click this object, the **Gecis12.cs** file will open in Visual Studio 2022. The simple coding required for transitioning between scenes is done in the file.

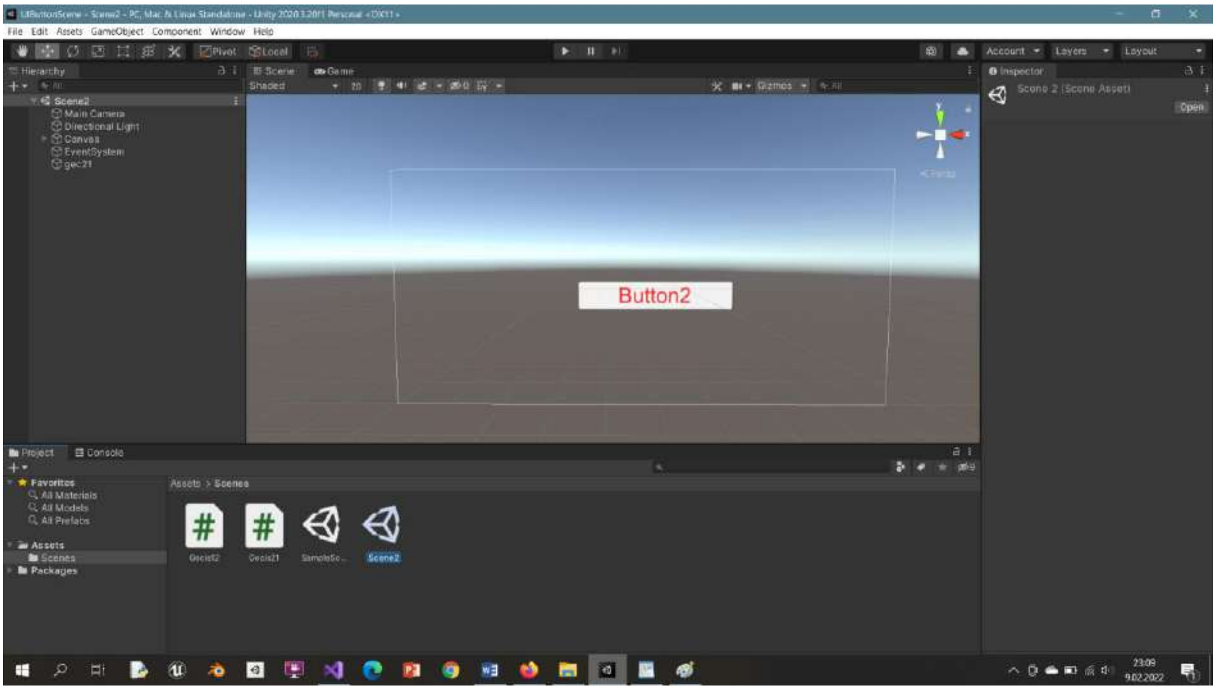




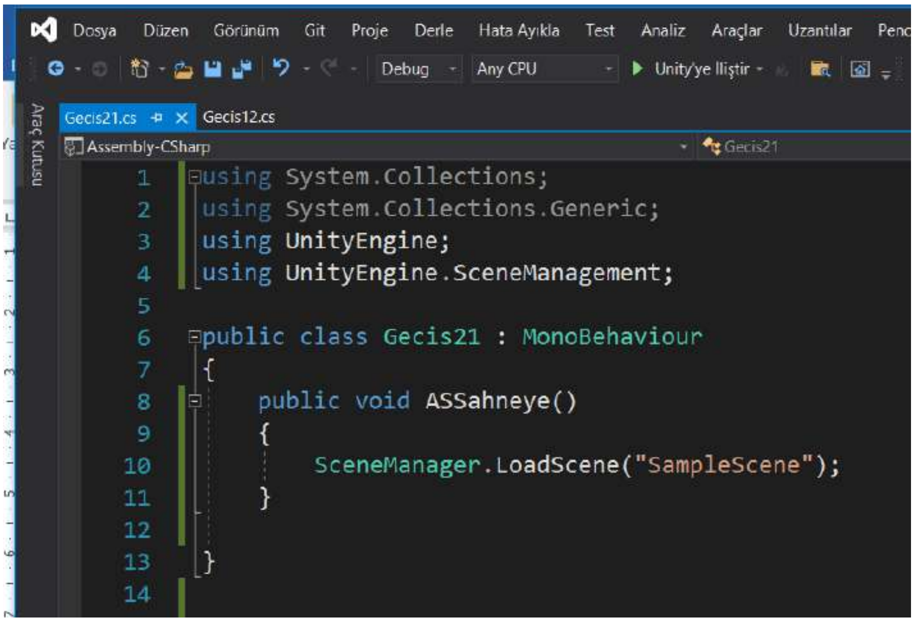
The command to switch between scenes is **SceneManager.LoadScene("Scene2");** Here we have a second scene, and its name is planned as **Scene2**.

Now let's save this scene and open a new scene under the **File** menu in the project. Name the scene as **Scene2**, as in the coding.

Create a **UI>Button** in the same way. Change its text to **Button2** and color **red**. Add a **GameObject** and name it **gec21**.

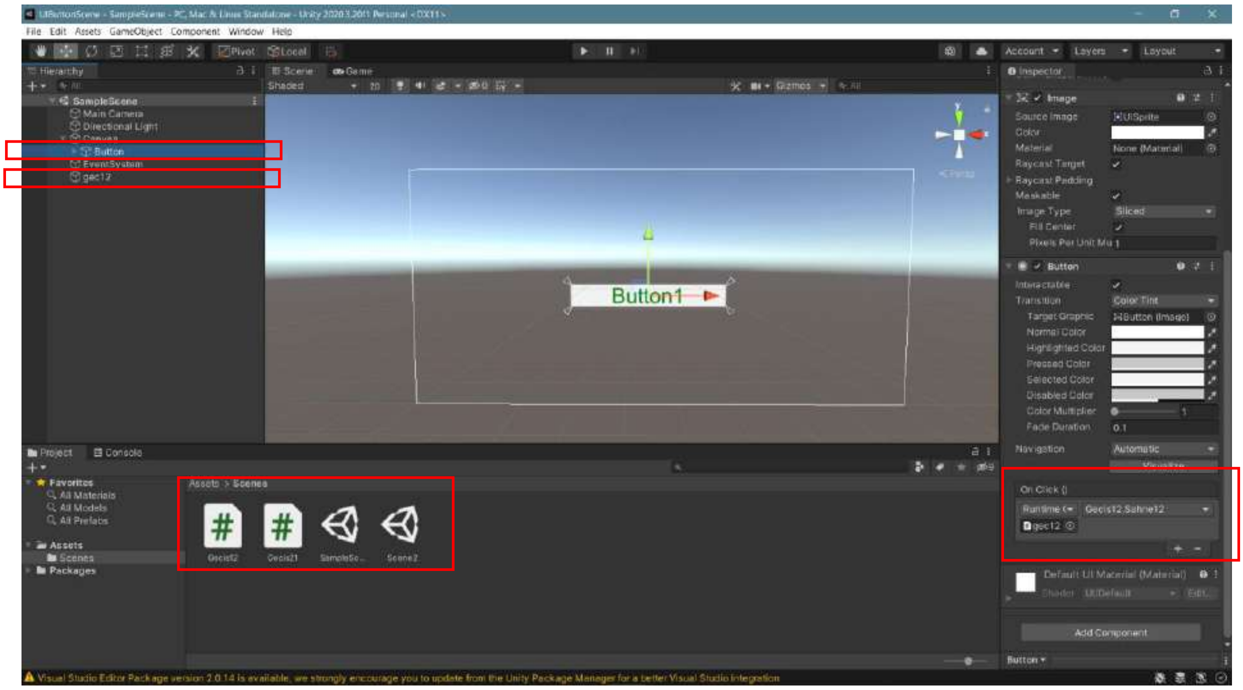


Create a C# file named **Gecis21.cs** for this scene and write similar codes in Visual Studio.

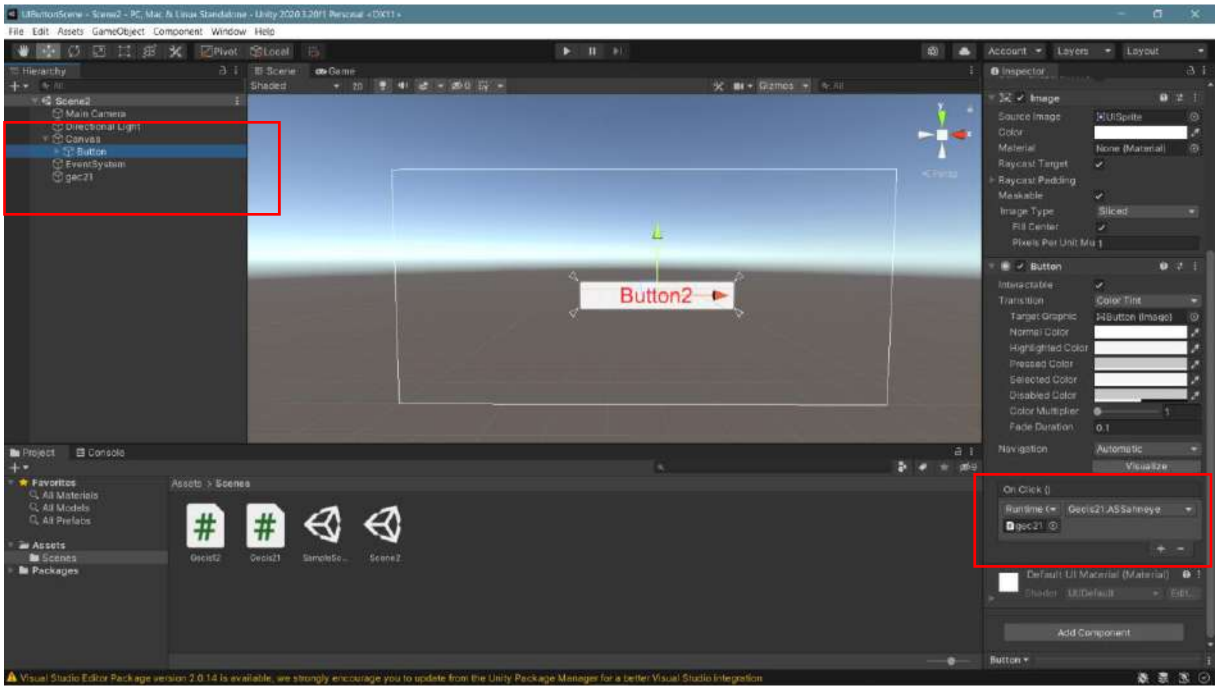


The function name is **ASSahneye(),** and our command can be written as **SceneManager.LoadScene("SampleScene");** That means switch to our first scene.

The next step is to connect the codes to the necessary objects. Now, drag and connect the code files to the **gec12** and **gec21 GameObjects**. When we select the **Buttons** in both scenes, let's press the **+** button in the **OnClick()** section in the **Inspector**. Drag the **gec12** object to the **None Object** section. Thus, we have connected our **gec12** object to the **Button** and the codes we connected to it. But another process is to specify which function we want to connect to the button in this code file. To do this, open the **No Function** section, select the name of the code file from the list, and select the **Gecis12> Scene12** function from the function list that opens under that name.

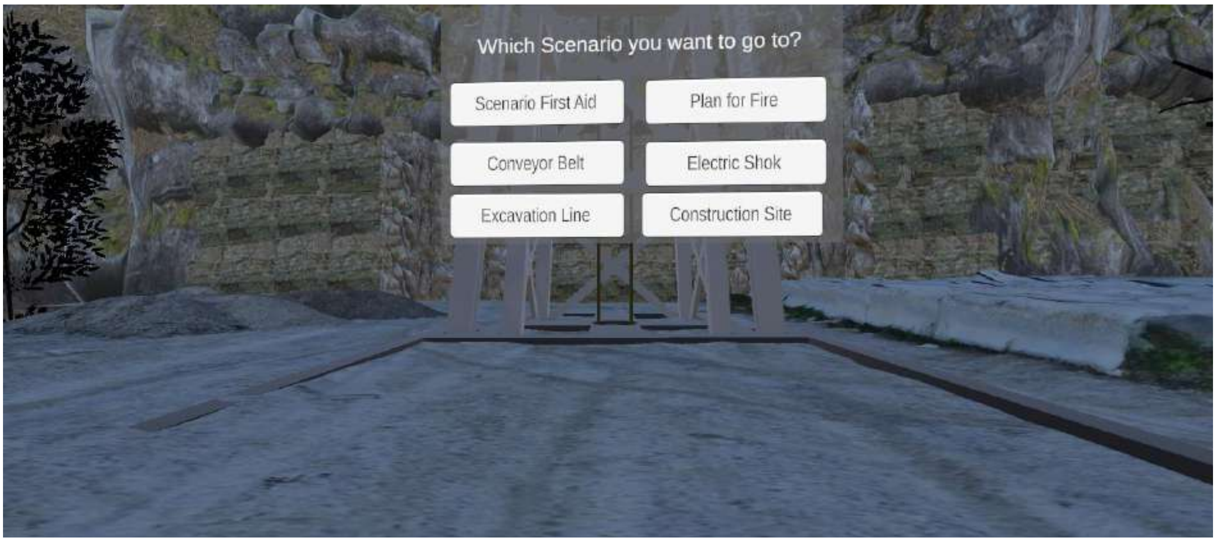


Then, do the same thing in the second scene.



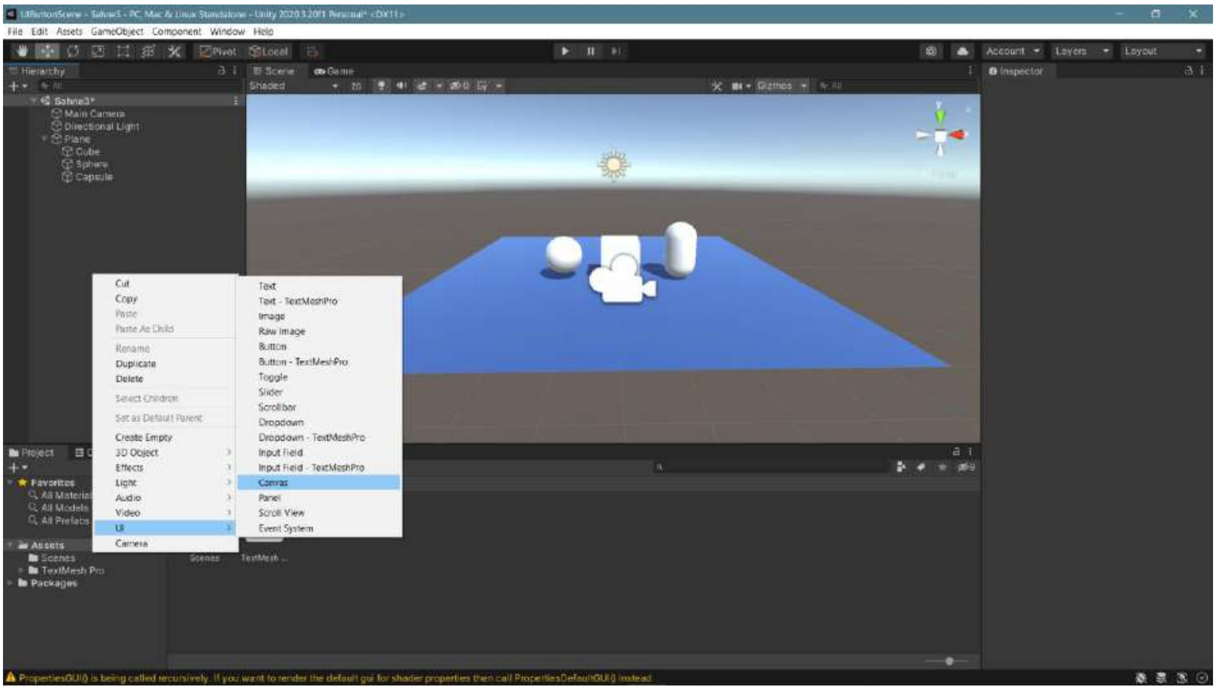
When the game starts, when **Button1** is pressed, we move to the second scene; when the second scene is pressed, and **Button2** is pressed, we move to the first scene. The scenes are very simple and only contain **UI>Buttons**. However, the purpose is to see the transition. After the scenes are designed for the purpose, it is possible to switch between many scenes. The design of buttons and interfaces has become a field of expertise. In this example, the operation of the process is discussed in its simplest form.

In a **mining project**, a more comprehensive menu driven application is given below.

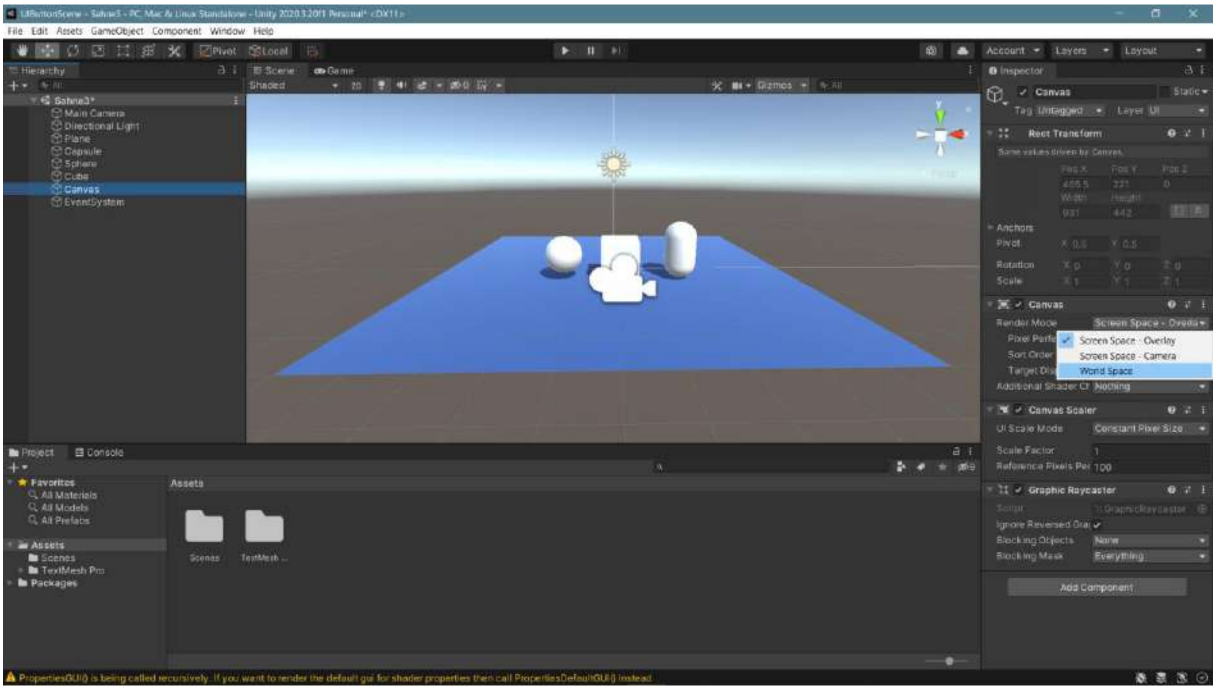


8.3.UI Text – World Space

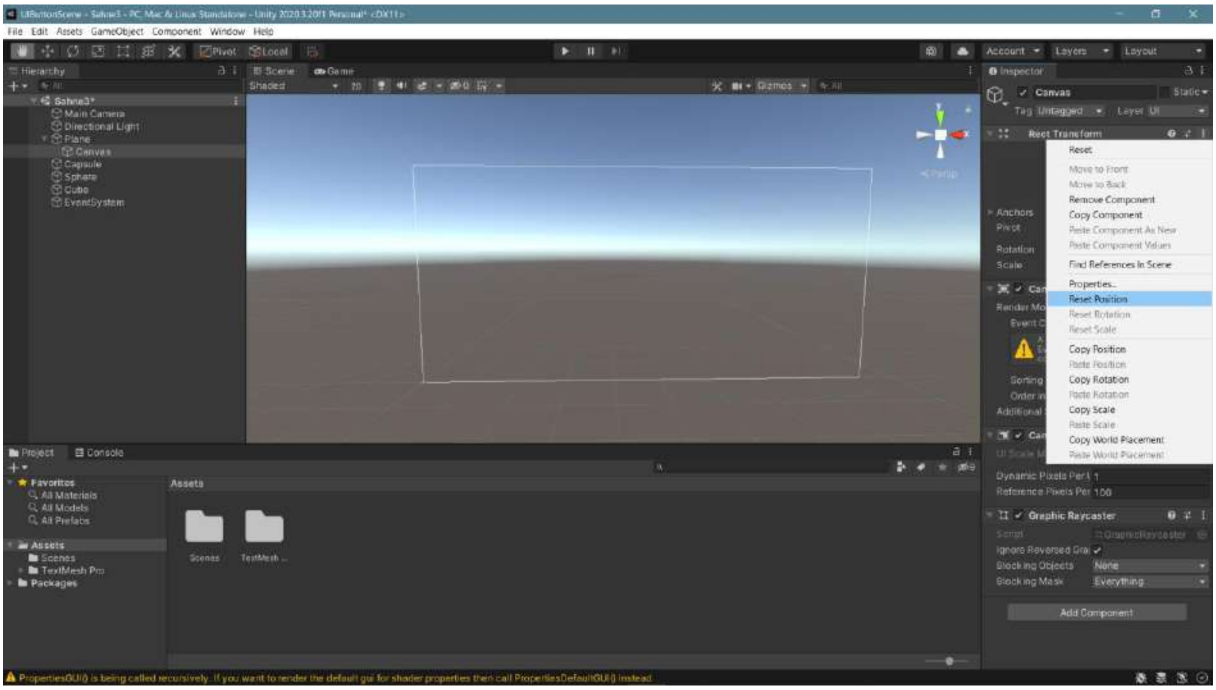
This application is for positioning **UI Texts** inside our scene. Normally, **Canvas** and **Text**, **Button**, **Image**, etc. subcomponents, which cover the whole screen due to their pixel structure and are very small next to our 3D scene, can be included in the scene with the size and positioning we want. Let's start by adding **Plane, Cube, Circle, Capsule** and **UI>Canvas** to our scene…



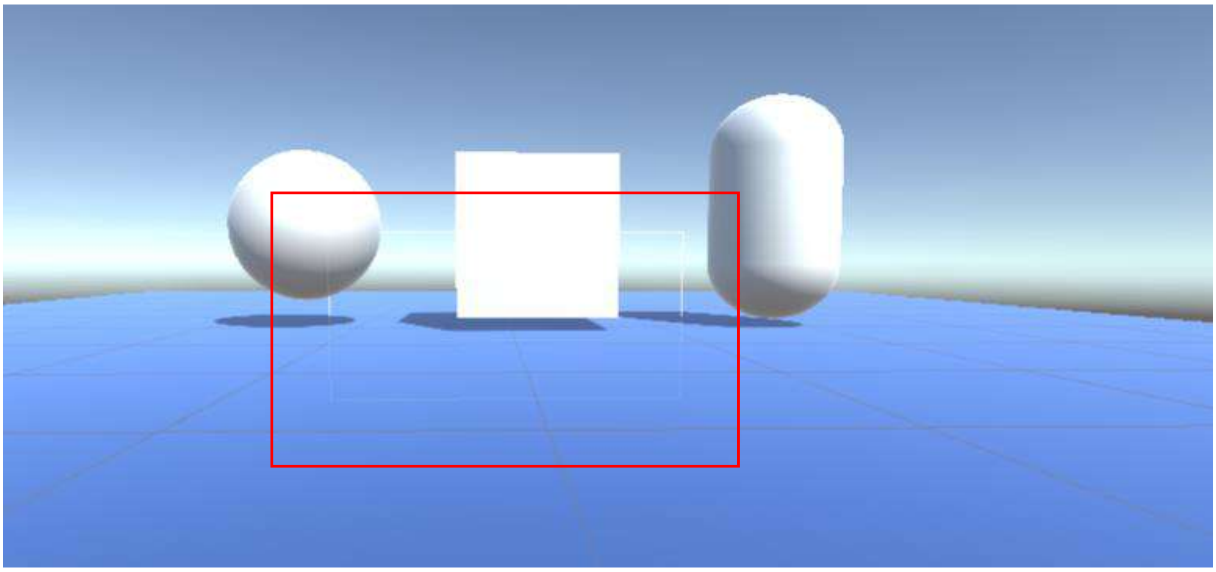
When we select **UI>Canvas**, change the setting in **Inspector>Canvas>RenderMode** to **WorldSpace**.



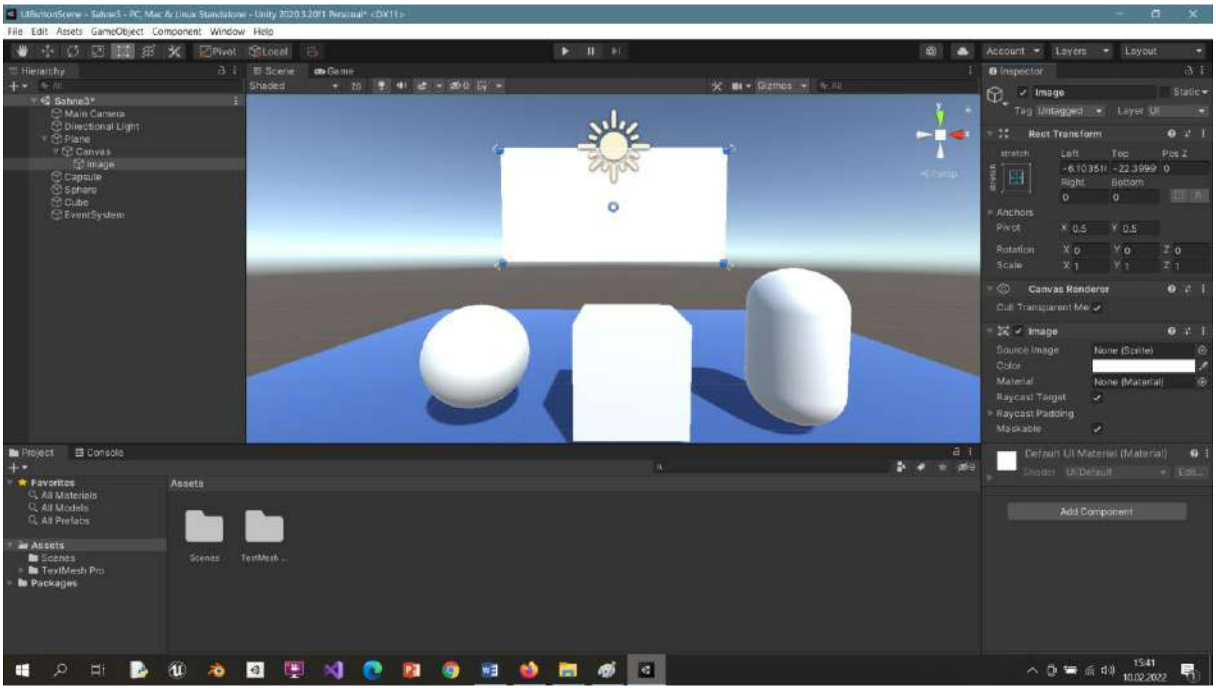
We can see how small our **Plane, Cube, Circle** and **Capsule** objects are.



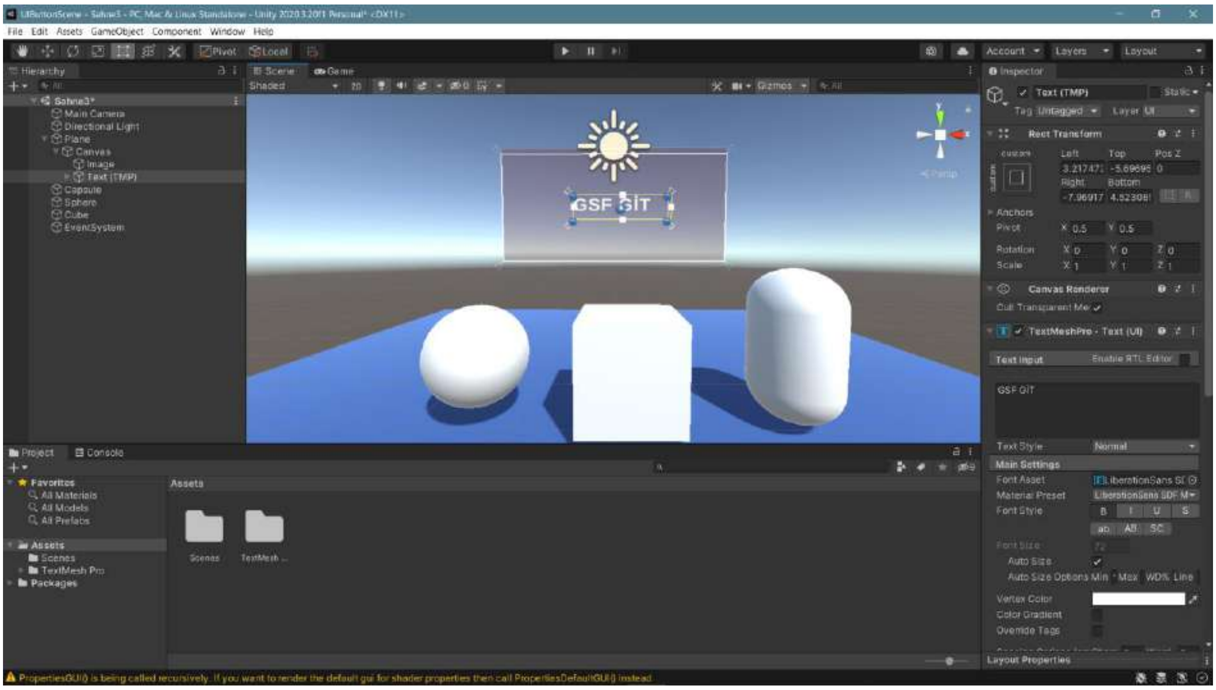
To move our **Canvas** from being an object covering the screen to an in-stage plane, bring the **Canvas** under the **Plane** to the **Parent-Child** position and provide coordinate unity with the **RectTransform>ResetPosition** operation. Then, reduce the **X, Y, Z** values in **RectTransform>Scale** to fit our stage; for example, **0.03**. When we focus on the **Canvas**, we can see its location and size.



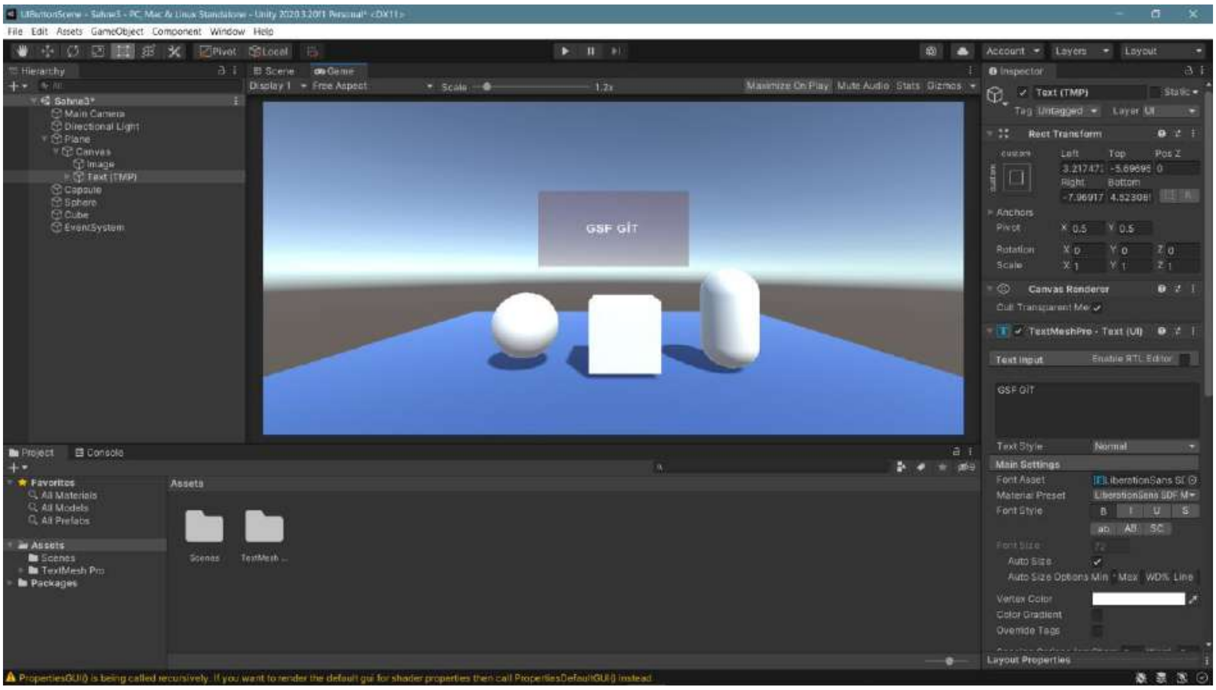
Now we can move the **Canvas** to the point we want and add **Text**, **Image**, **Button**, etc. to it. Let's continue the example by adding **UI>Image** under **Canvas** after positioning. If we want the **Image** to cover the **Canvas**, we can bring it to the size we want with the **Stretch** or direct scale button under the **Anchor**. The initial opening color of the **Image** is white.



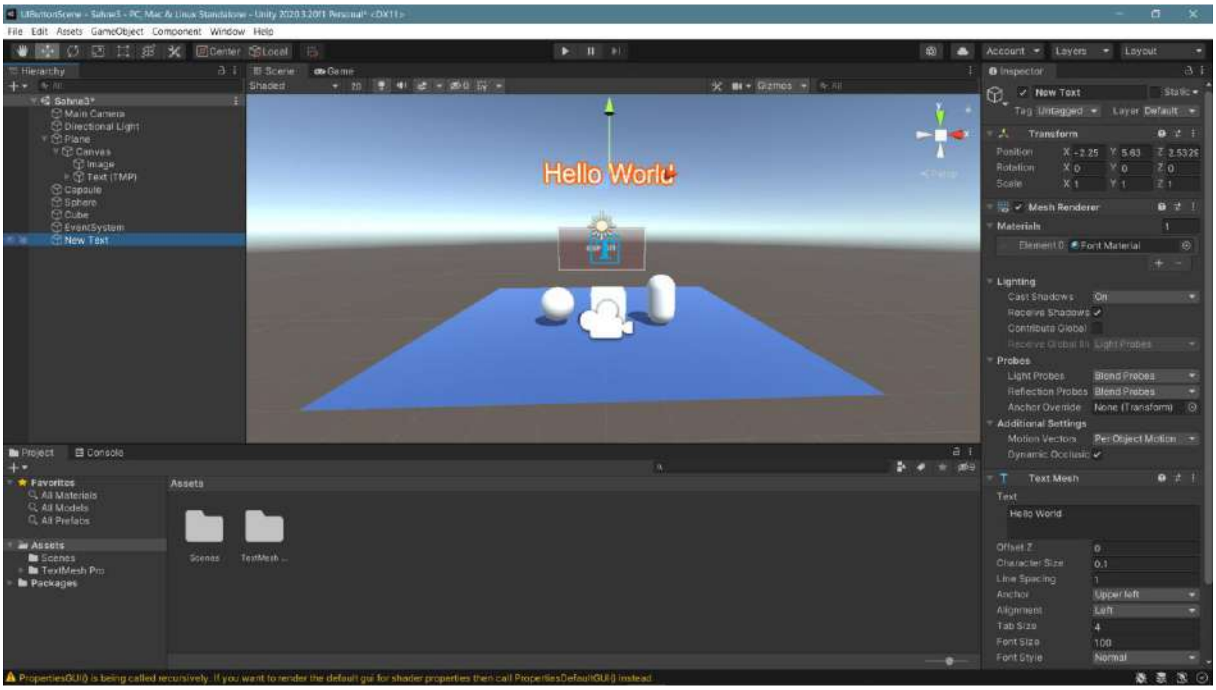
Change the color and transparency via **Color**. It's time to add text. Let's add **UI>TextMeshPro** to Canvas. We wrote **GSF GIT** instead of *New Text* in the **Text** section and made size and position settings.



Now, check the image in **game** mode.



To add text to the scene, select **3DObject>3DText** in **Hierarchy** and place the text in the desired location on the scene. The resolution of the text can be increased by decreasing the **Character Size** and increasing the Font Size.

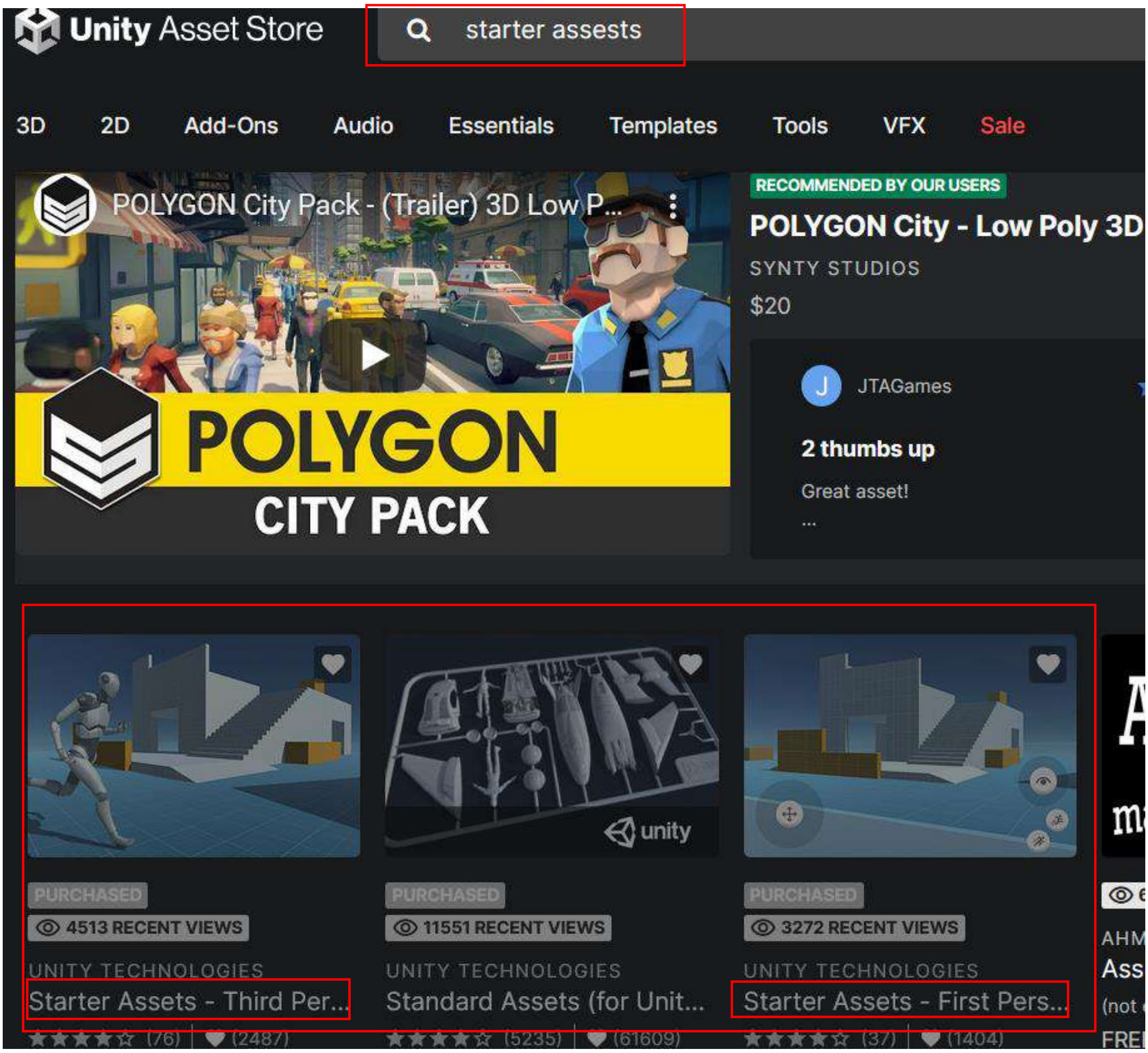


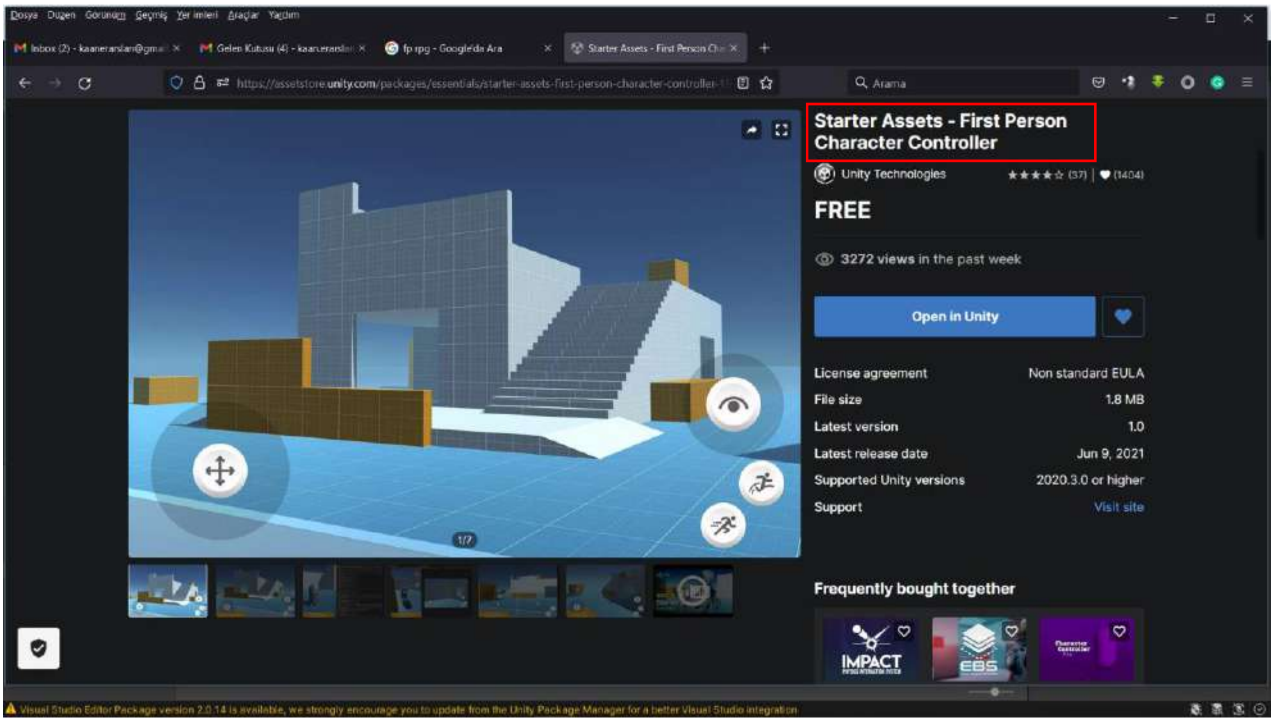
Font types, colors, sizes, and locations are up to the designer. Check it out in **Game** mode, too.

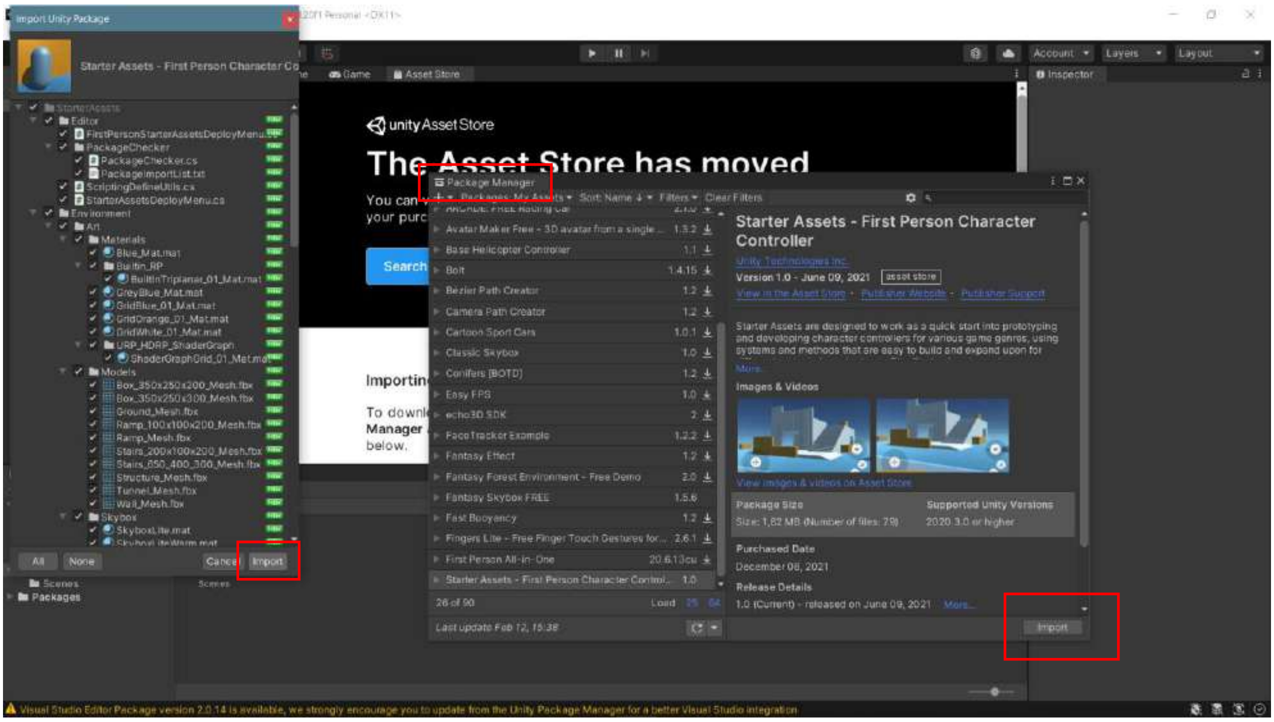
9. FPS VE TPS APPLICATIONS

9.1.FPS-First Person Shooter and RPG-Role Playing Game

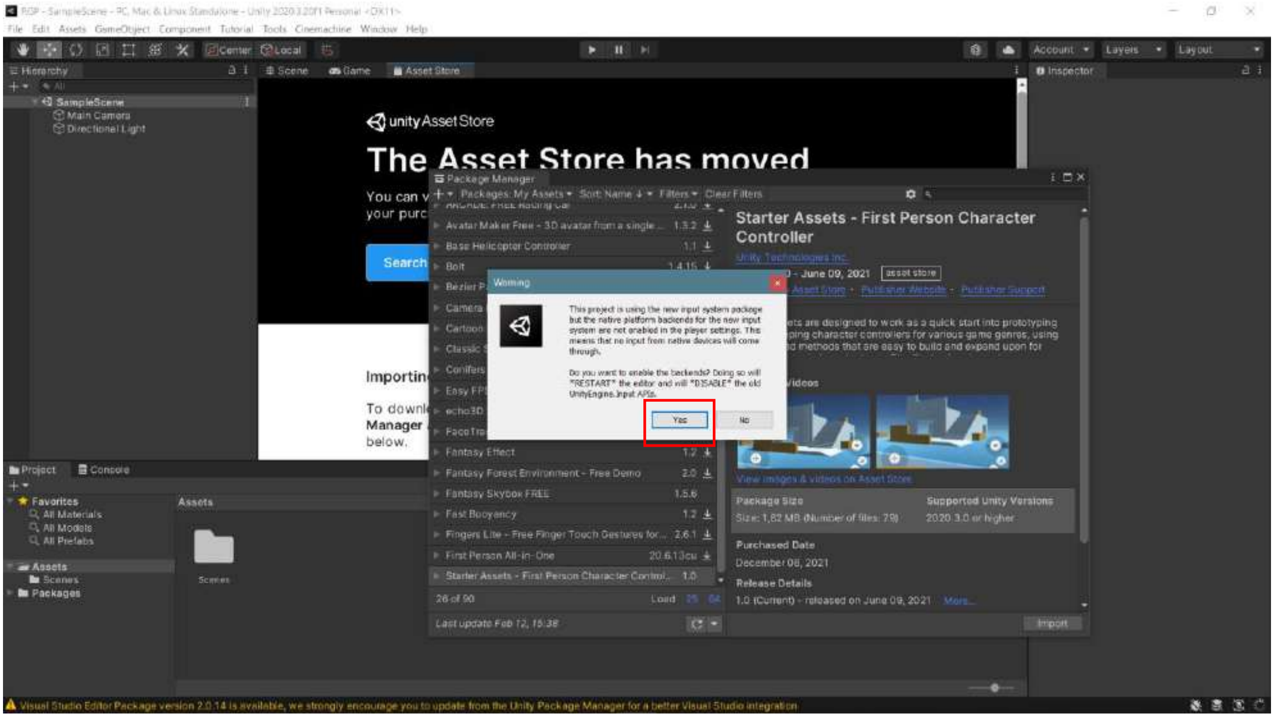
If the game has a plot where you play through the eyes of the game hero and in his shoes, this type of game is called **FPS-First Person Shooter** or **FP-RPG-First Person Role Playing Game**. For the application, we can search for **Starter Assets**, which is offered free of charge by **Unity Technologies** in **the Unity Asset Store**. In this way, *Standard Assets* was previously downloaded. Now, there are two more packages on the list; **Starter Assets First Person** and **Starter Package Third Person**. Now, add the **First Person** one to our assets and import it from the **Package Manager**.

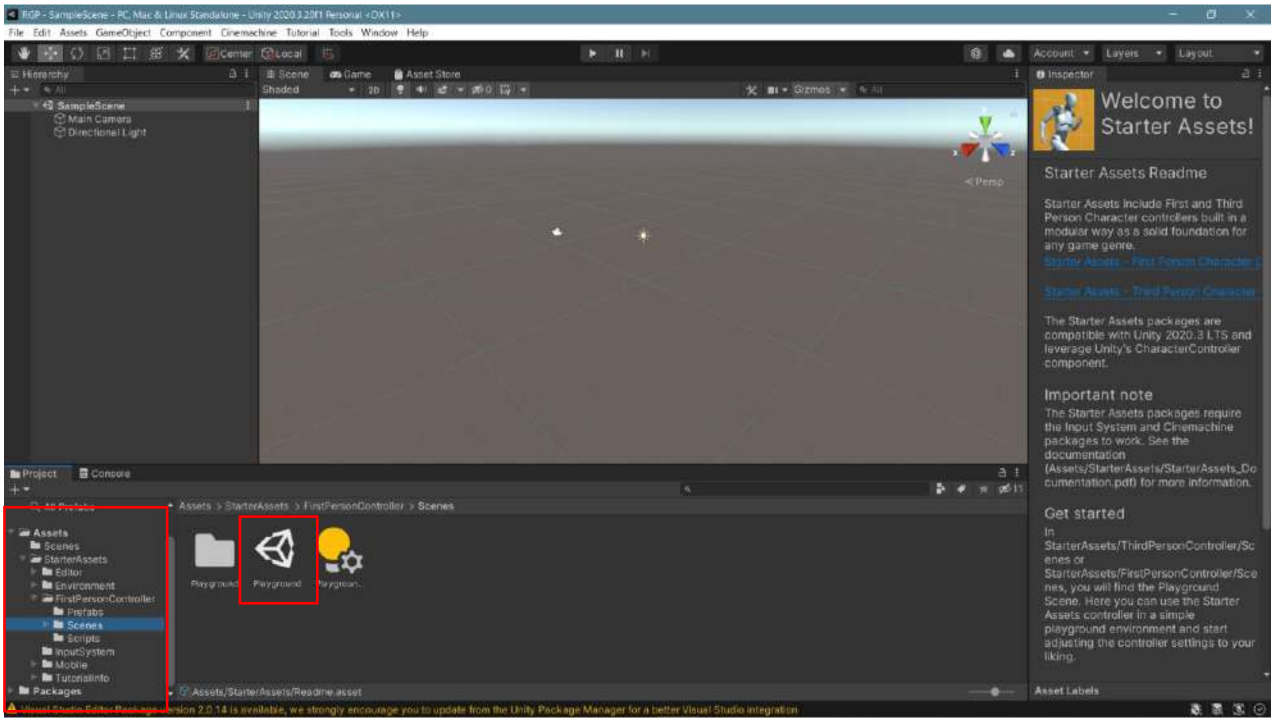




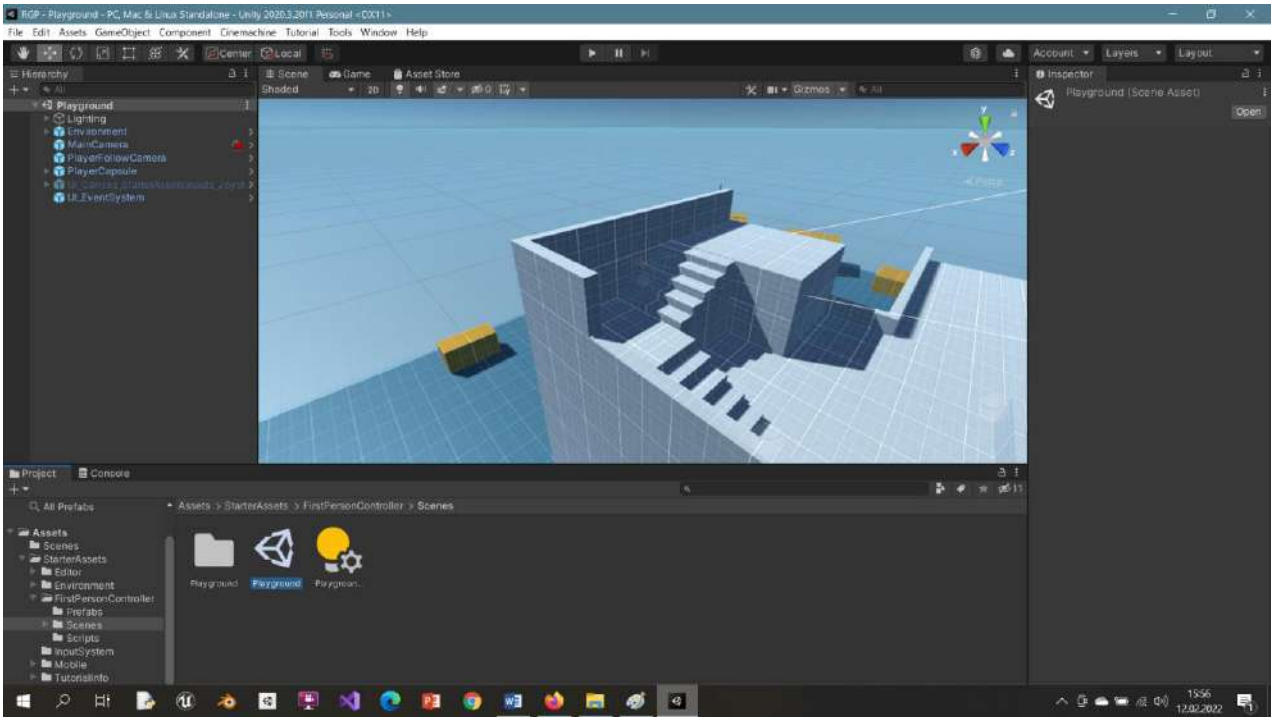


After this process, a confirmation window will open to **re-adjust** Unity settings. After saying **Yes**, Unity will be closed and automatically **reopened** with its new settings. Standard Assets are visible in our Assets window. Now, let's go to **Starter Assets>FirstPersonController>Scenes** and see the **PlayGround** scene below it.



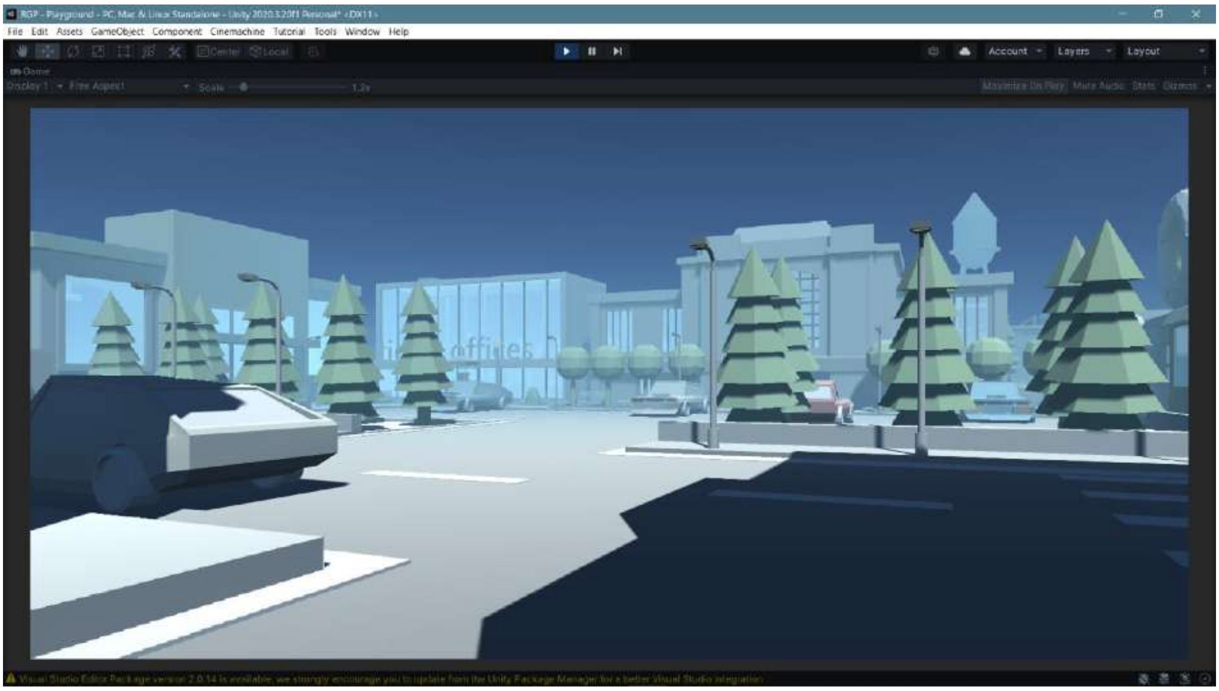


When we double click on the **PlayGround** scene file, our new scene with all its edits will appear on the screen.



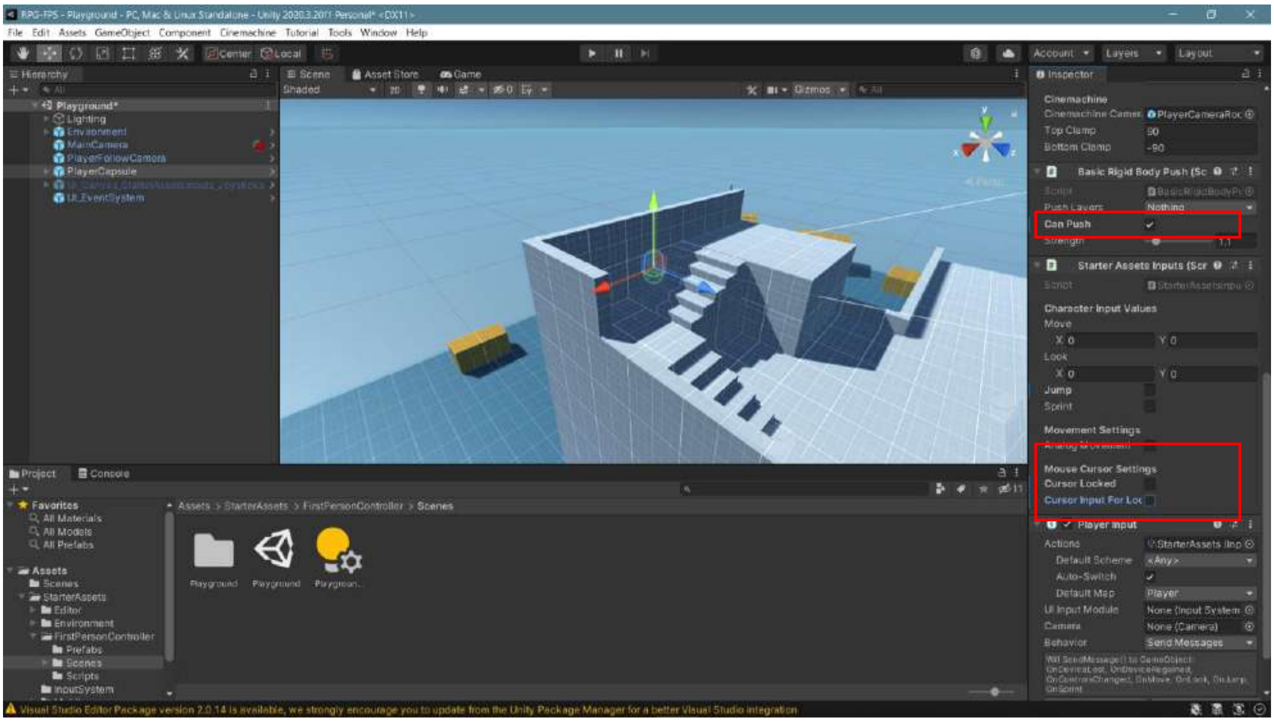
When the game is launched, we will see that we can move around the stage with the arrow and shift keys and the mouse, look and move in all directions, and jump with the spacebar.

It is possible to create or add our own scene instead of the **Environment** that creates the scene. In this way, we start to move within our own design. By taking a Low Poly city model from the Sketchfab site to our scene and making the **Environment** object passive, it became possible to walk around the city.



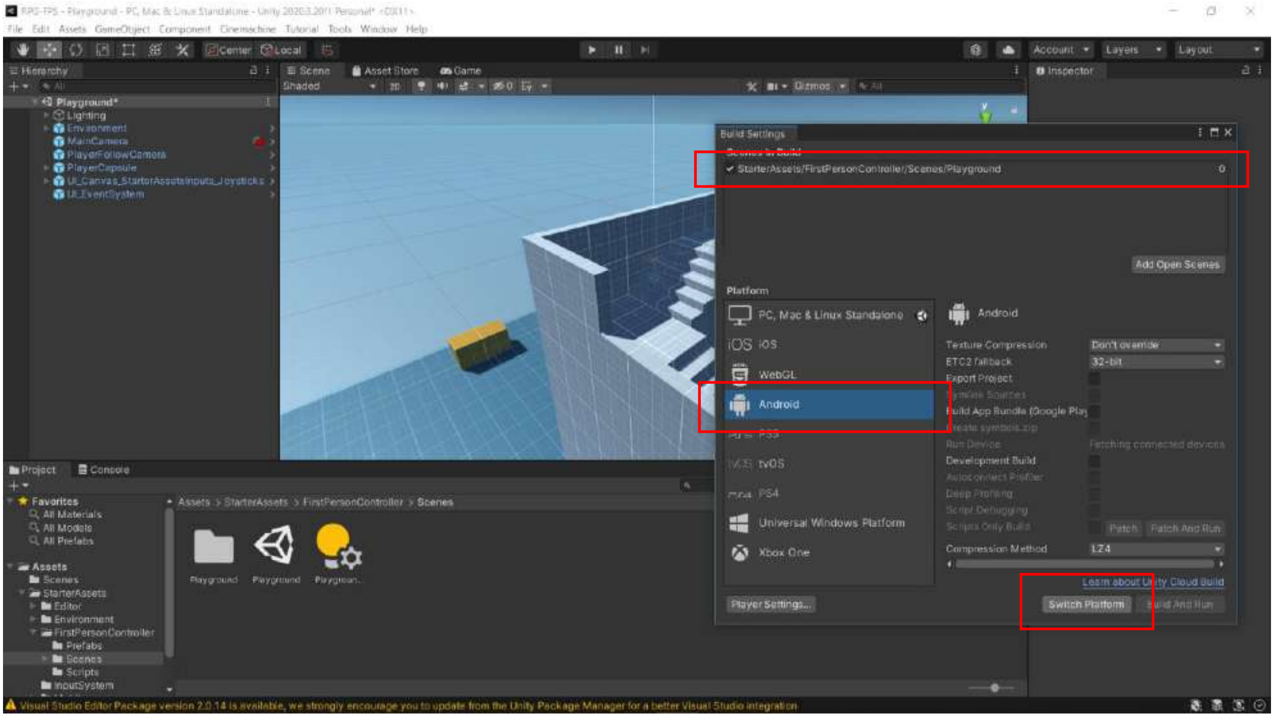
It is also possible to use this application on mobile devices. However, since there are no keys like on the keyboard on mobile devices, these controls will be made with keys called virtual **TouchPad** or **joystick**.

First, let's go to the **Movement Curser Settings** setting in the **PlayerArmature Inspector** section and **uncheck** the boxes. Check the **Can Push** box in the **Basic Rigid Body Push** section so that it can push objects.



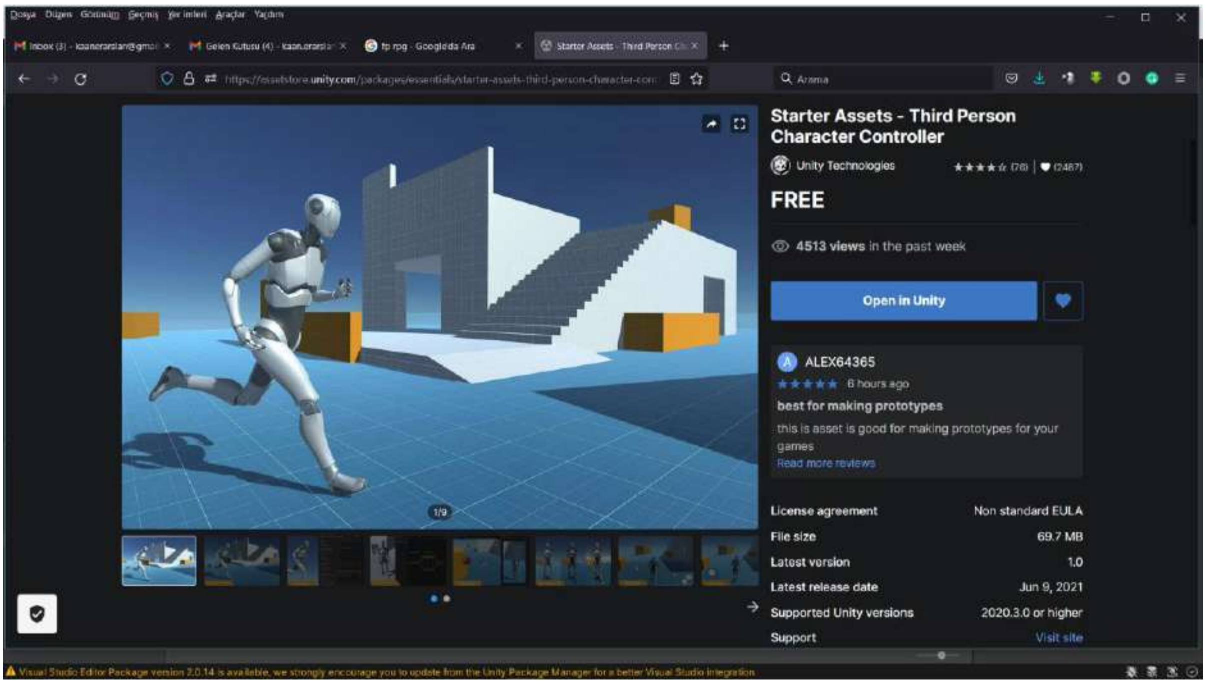
Now, activate the **UICanvasSaterterAssetsIputs_Joysticks** object which is previously passive in the **Hierarchy** section.

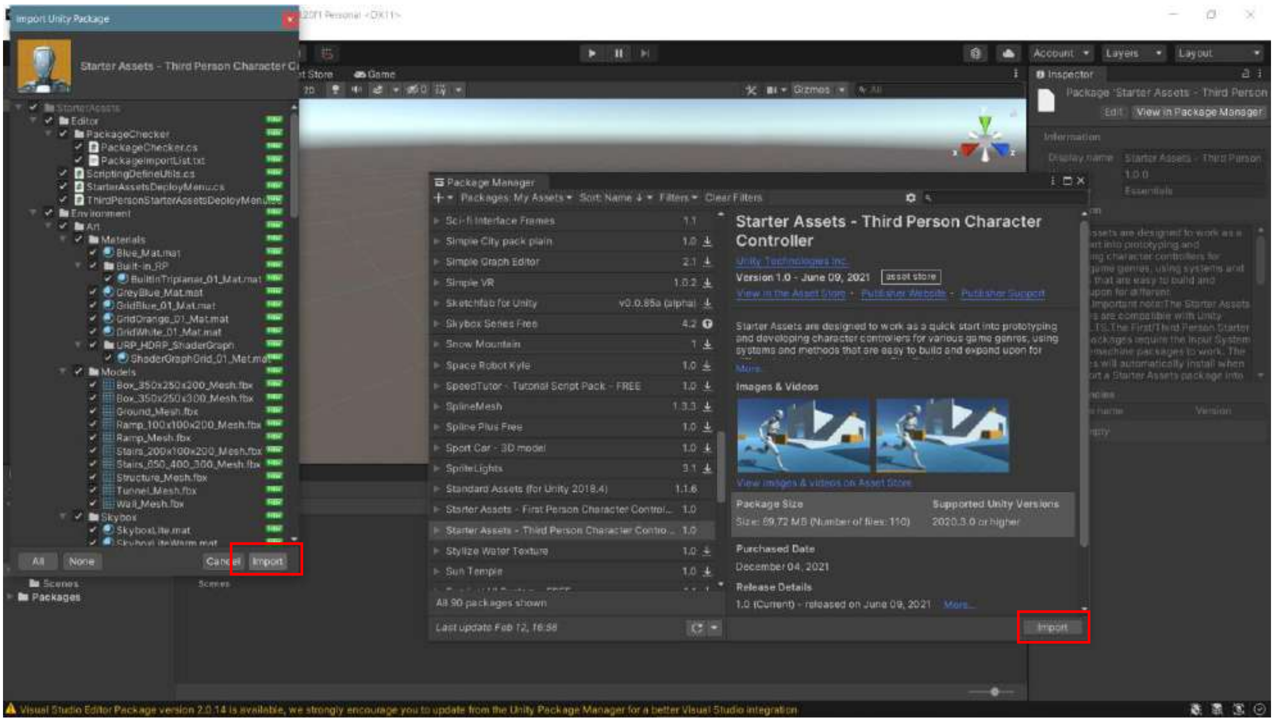
Control is now enabled with the added **joystick**, not the **keyboard**. In this way, by going to **Build and Settings**, it will be possible to convert it to **Android** or IOS application according to the **phone-tablet** type and play it with the **joystick** on mobile devices.



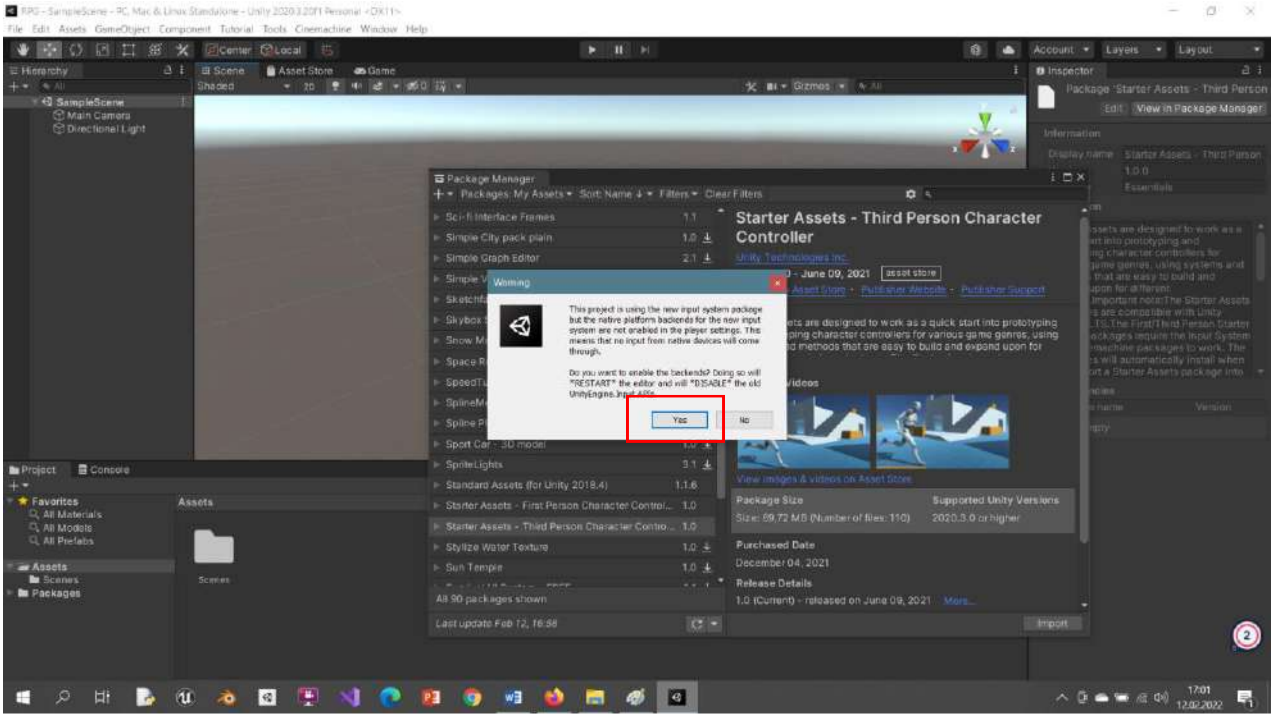
9.2.TPS Third Person Shooter-Starter

A **Third Person Shooter** or **Third Person Role Playing Game** type application is a type of game setup where we follow the player and see their movements in the form of animations. Let's follow a similar path for this. This time, **Starter Assets - Third Person** will be downloaded and imported from the Asset Store.

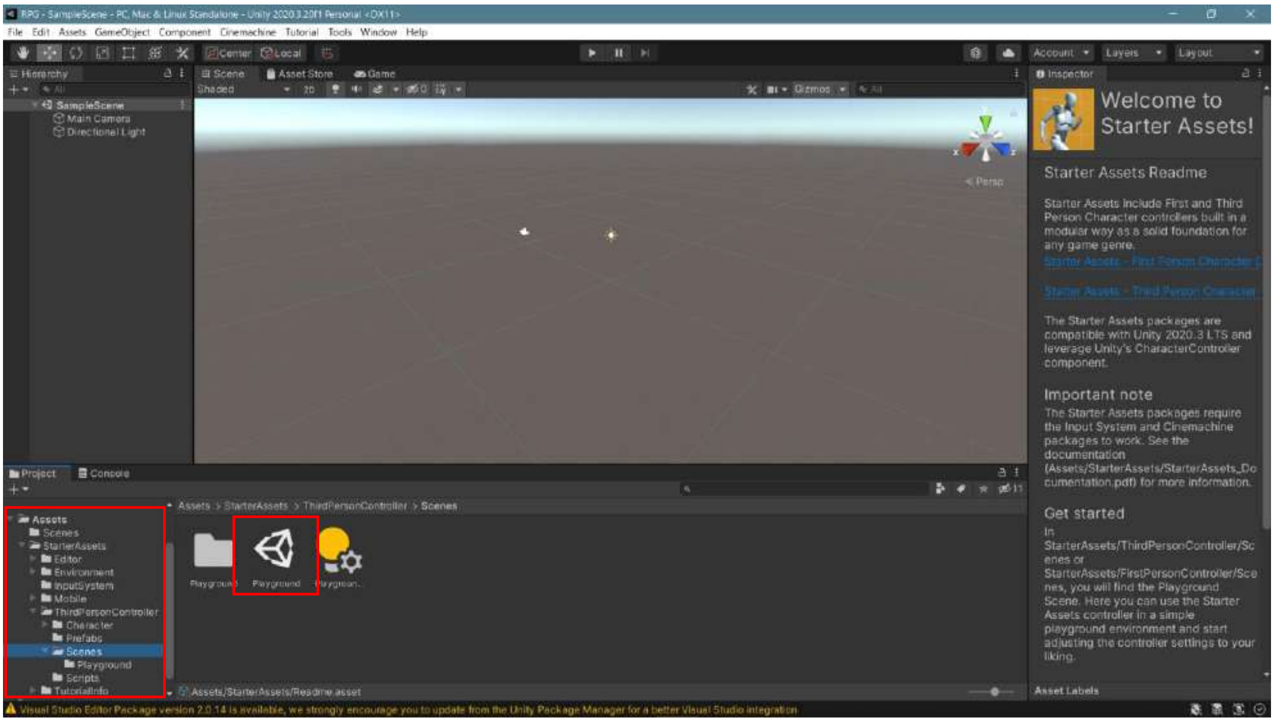


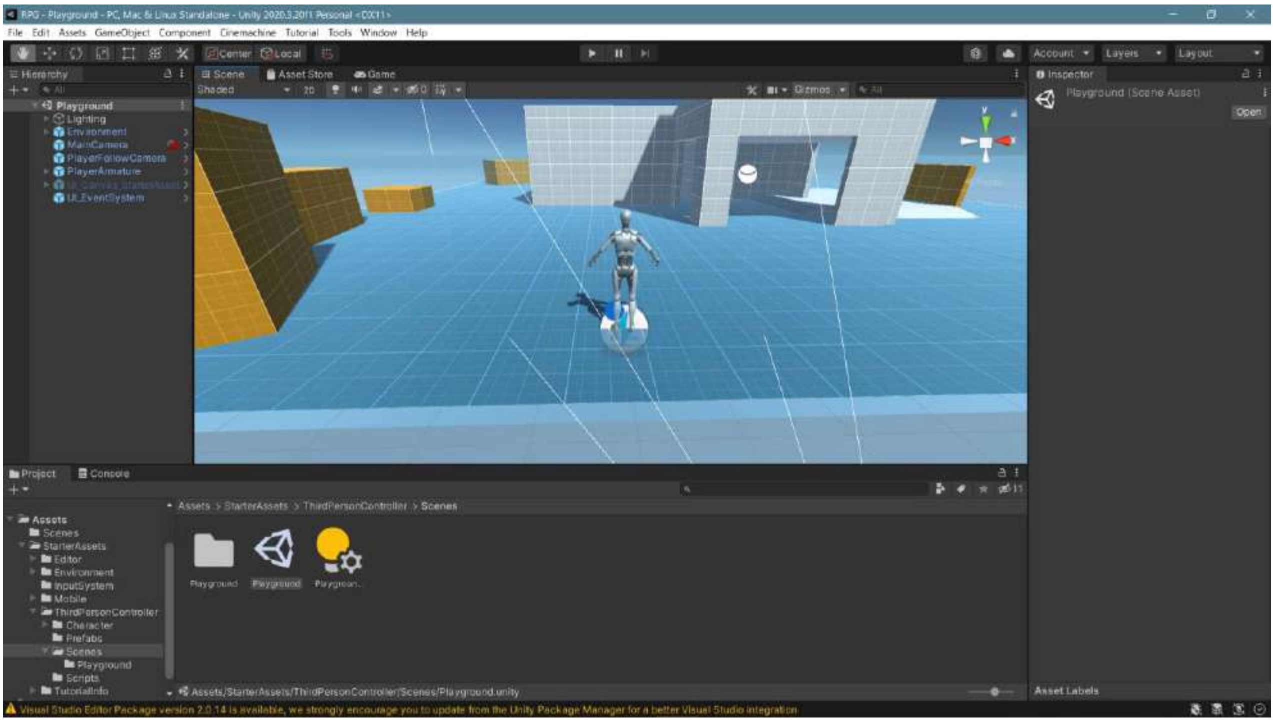


Similarly, you will be asked for **restart** confirmation for the settings to take effect, click **Yes** and the project will be closed and **restarted**.

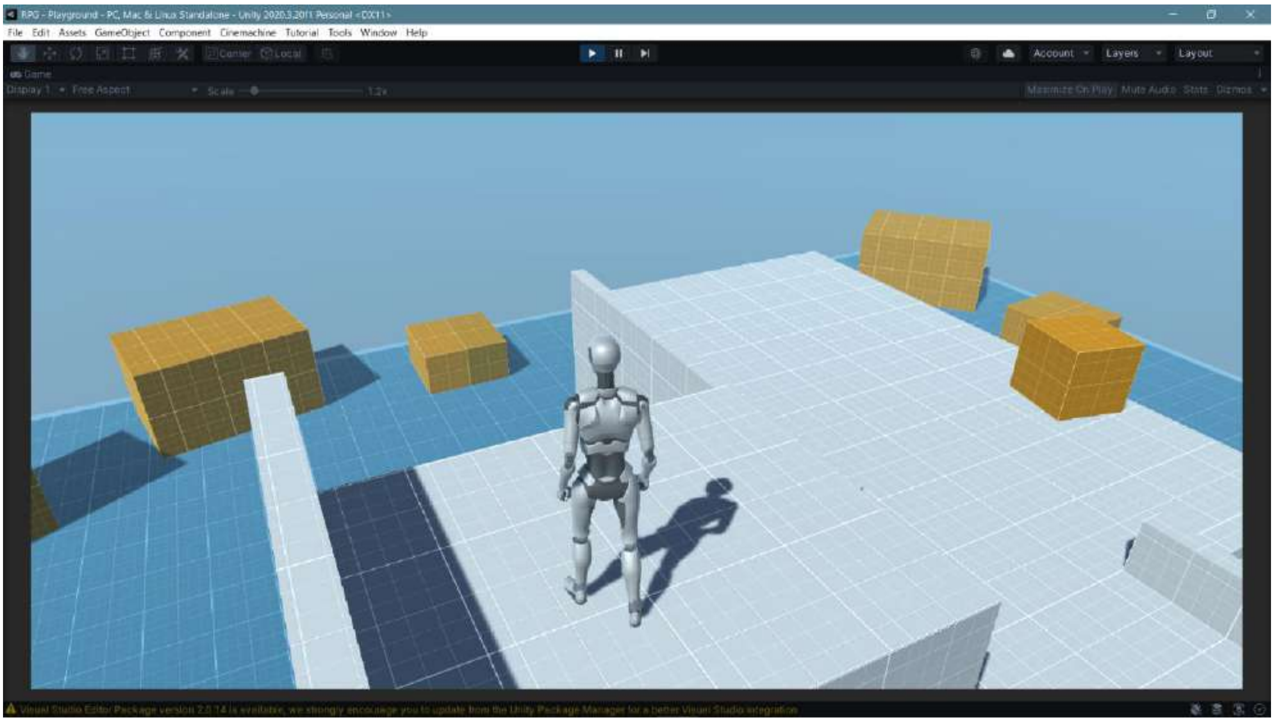


In the package that comes to the **Assets** section, we will just need to **double-click** the **Playground** scene in **Assets>Starter Assets>Third Person Controller>Scenes**.



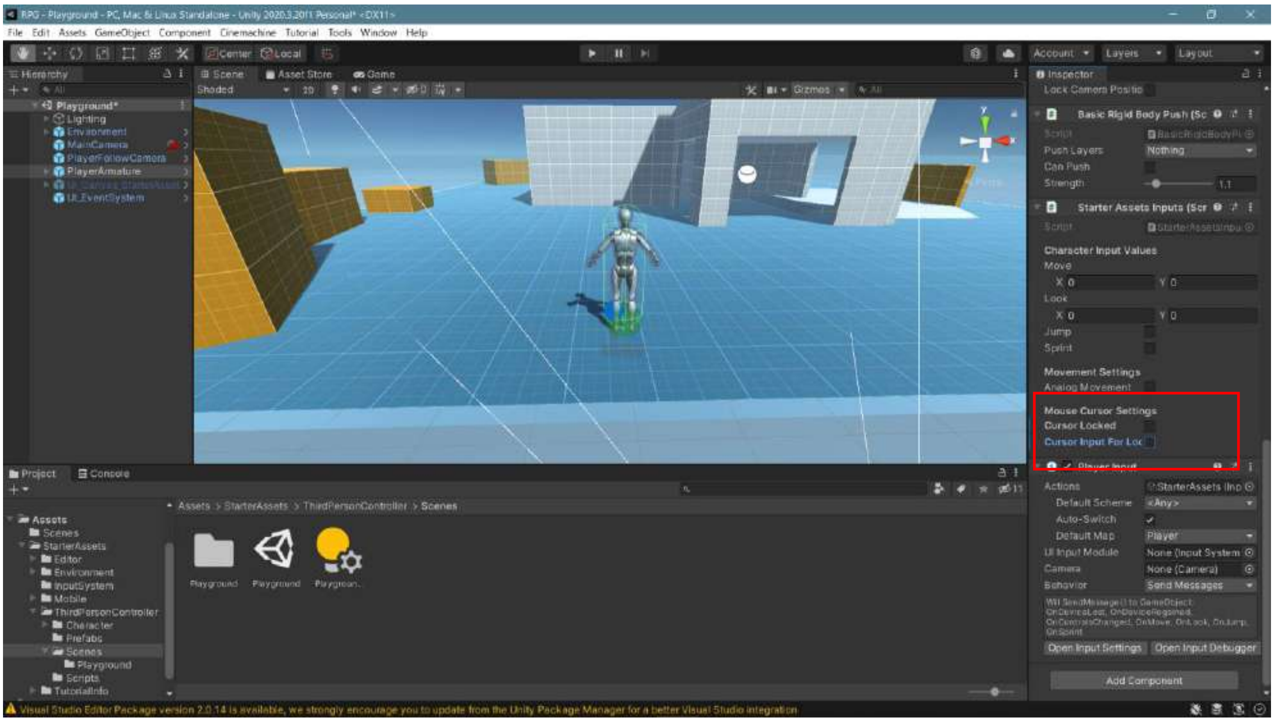


The player being followed on stage is a robot. When the game starts, it will be seen that we can control the movements with the arrow keys, shift, space bar and mouse.



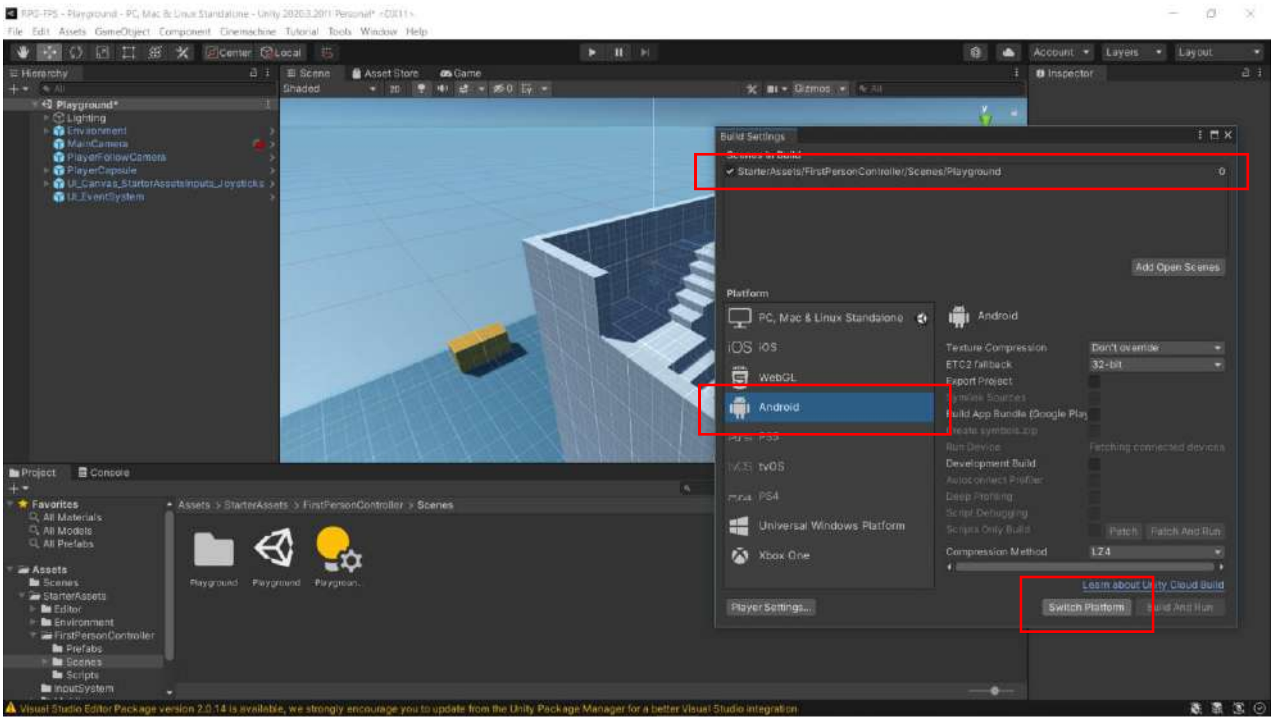
This game can also be adapted to mobile devices. However, since there are no keys like on a keyboard on mobile devices, these controls will be made with a virtual **TouchPad** or **keys**.

First, let's go to the **Movement Curser Settings** setting in the **PlayerArmature Inspector** section and **uncheck** the boxes. If the **Can Push** box is checked in the **Basic Rigid Body Push** section, it will also be possible to push objects.

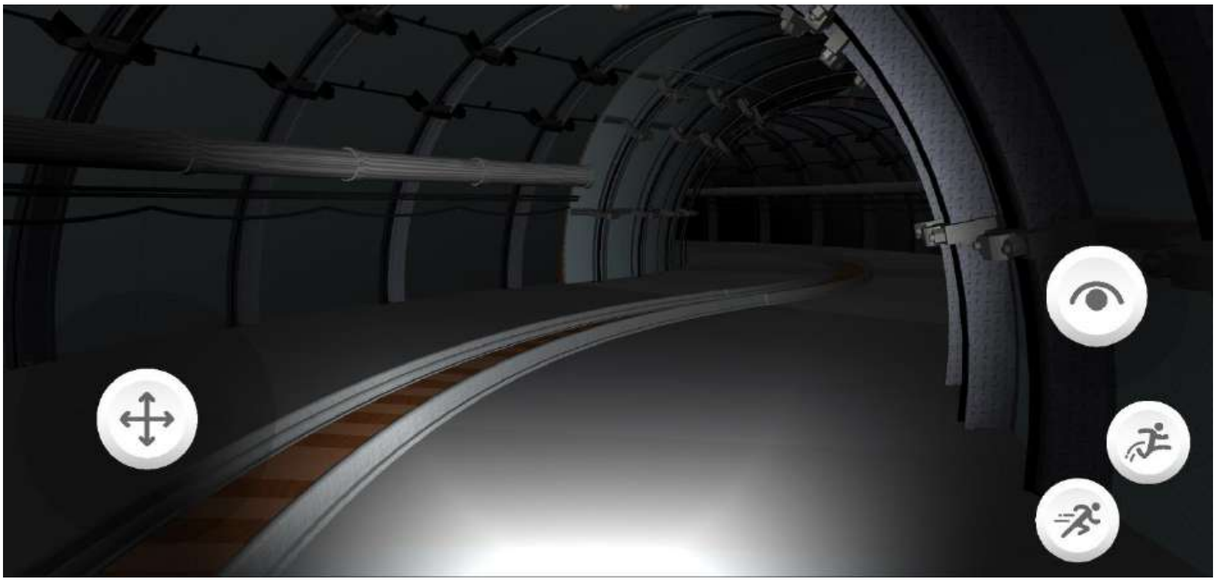


Then, activate the **UICanvasSaterterAssetsIputs_Joysticks** object which is previously passive in the **Hierarchy** section.

Control is now enabled with the added **joystick**, not the **keyboard**. In this way, by going to **Build Settings**, it will be possible to convert it to **Android** or IOS application according to the **phone-tablet** type and play it with the **joystick** on mobile devices.

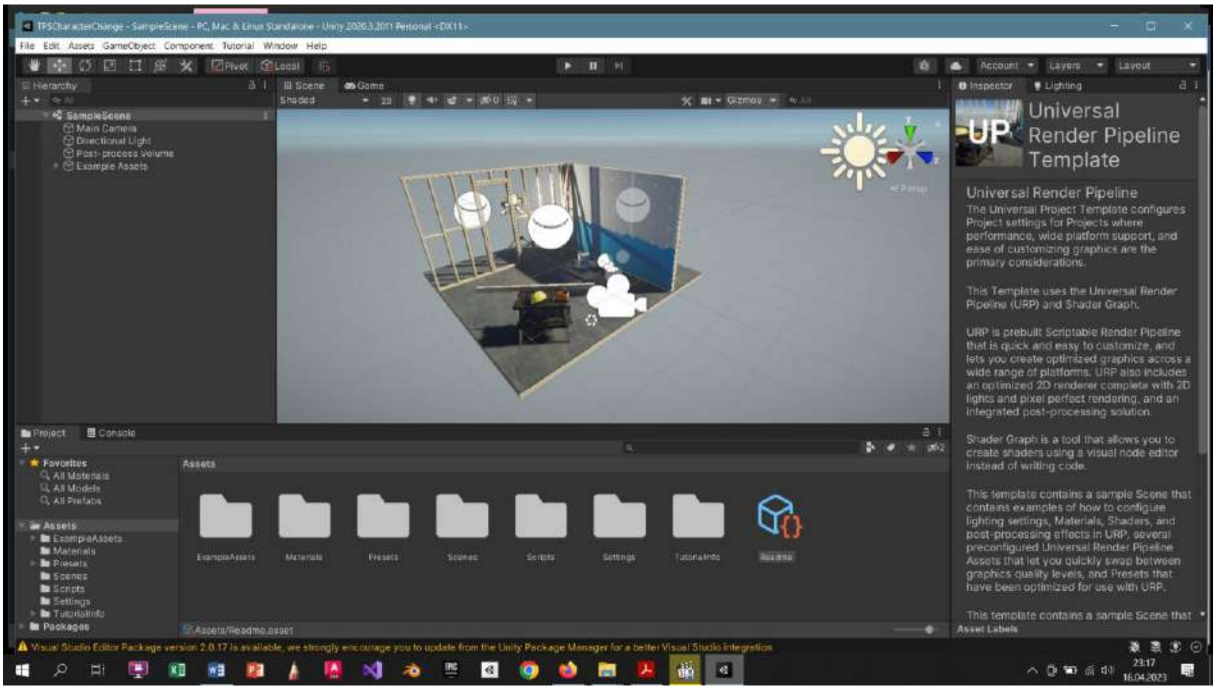


By using a mine gallery model, virtual joystick application can be performed. Here, FPS object also carries the light source (Spotlight) and makes illumination of the gallery with it.

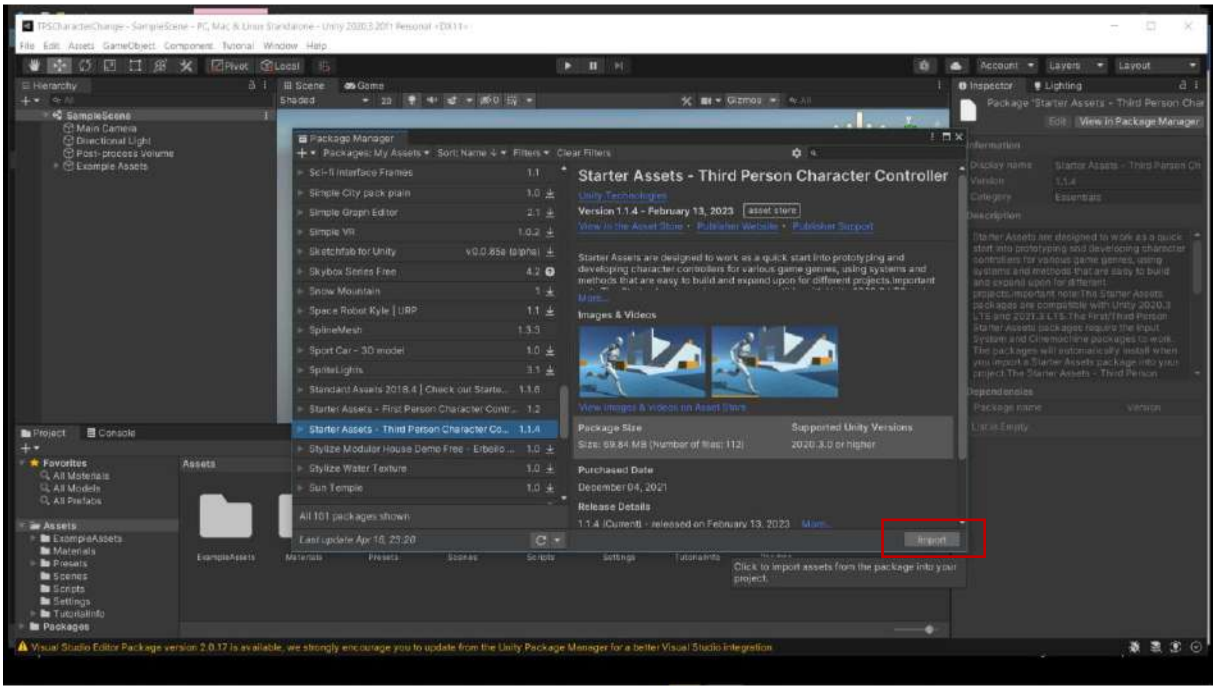


9.3.Third Person Character Controller – Armature Change

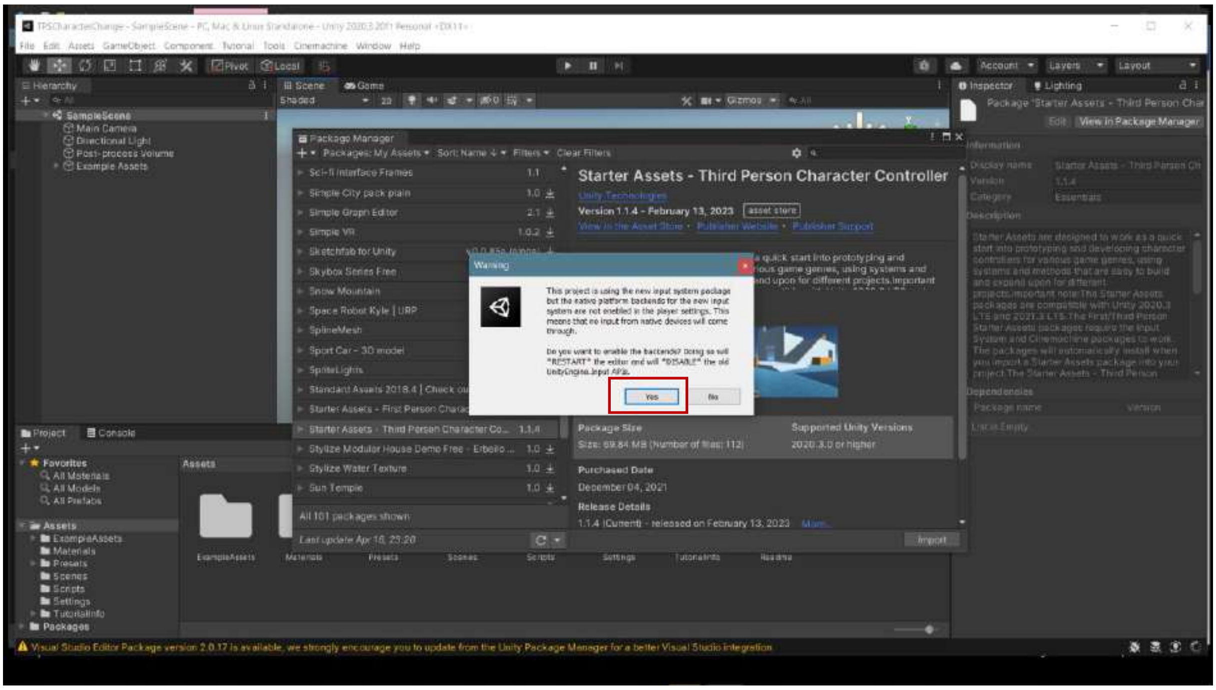
The app can be experienced on Unity 2020.3.10f and above. The **Universal Render Pipeline** (**URP**) template will be used.



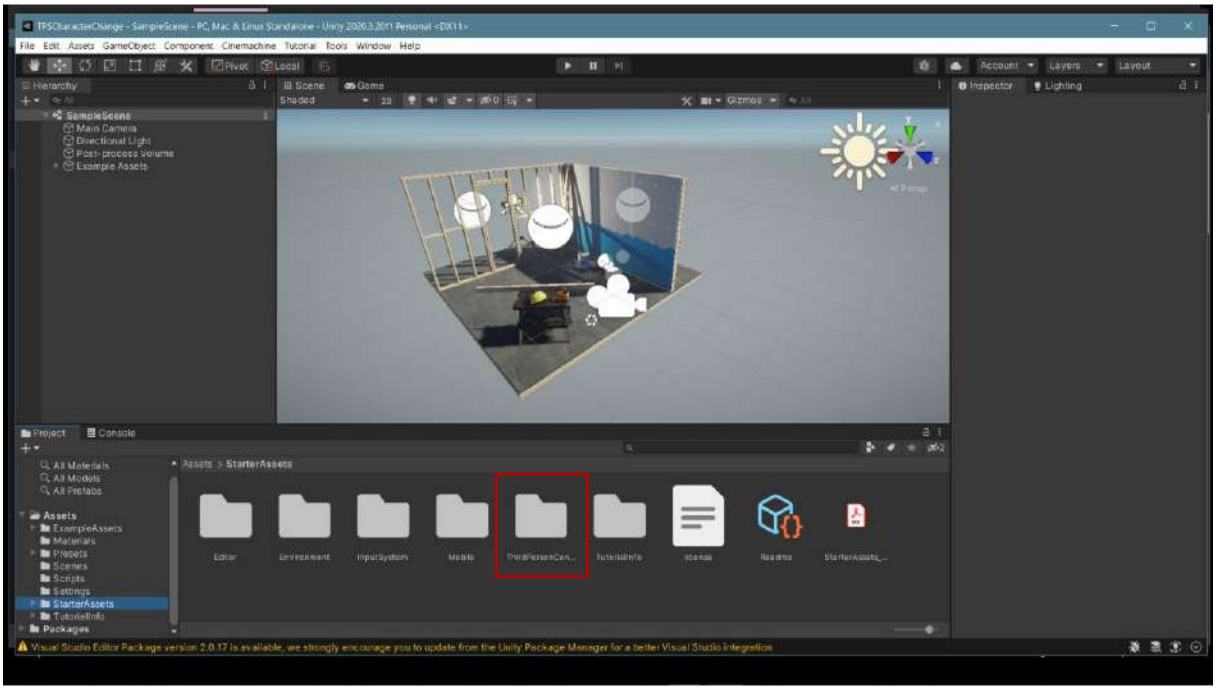
From **Window -> Asset Store**, **Starter Assets** will be downloaded to our **Third Person Assets** project.



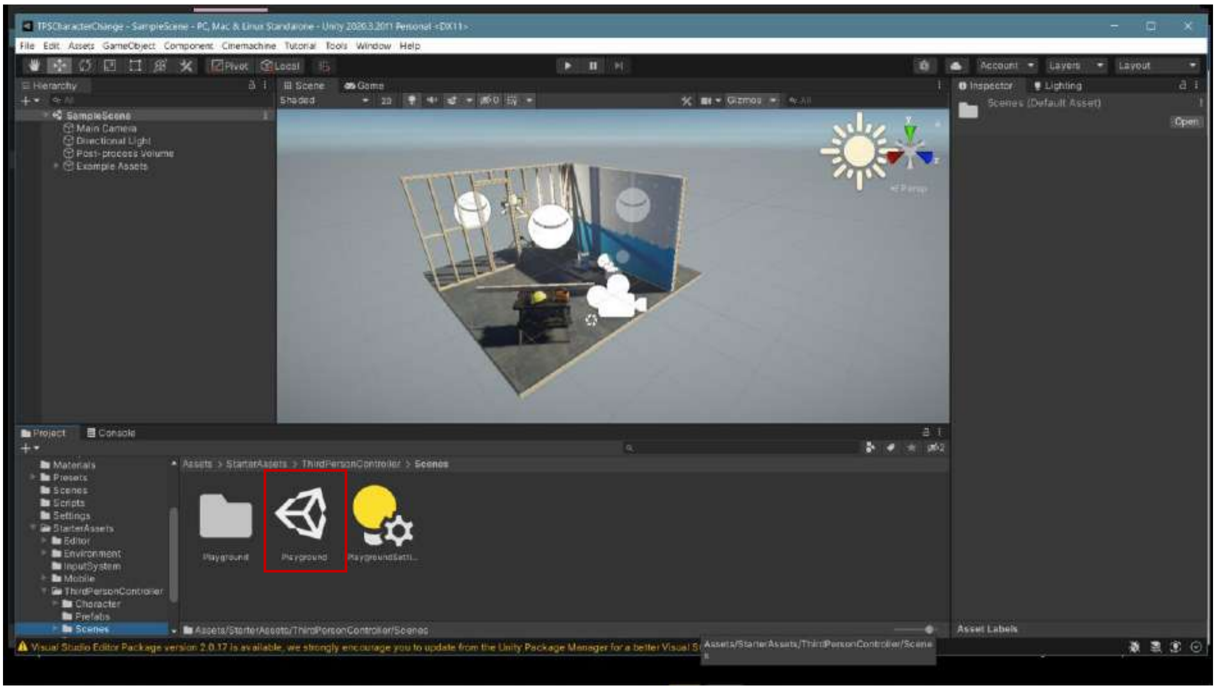
After import, accept the **restart** request.



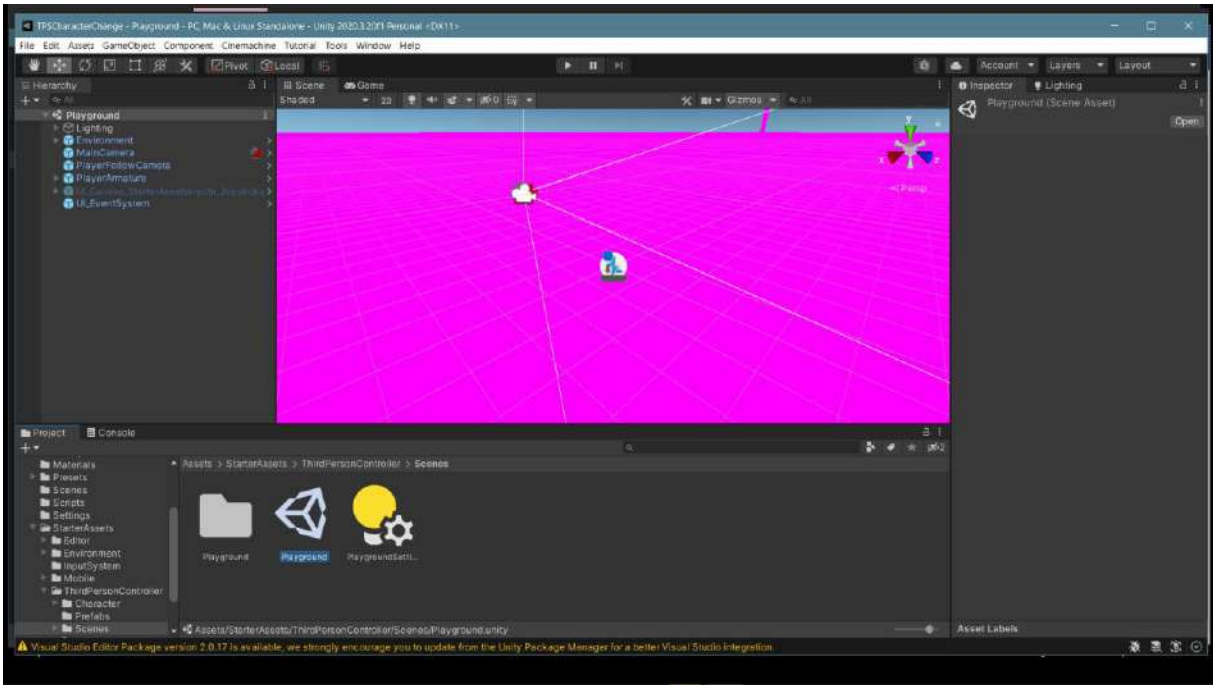
Since the **URP** template is used, the sample scene will appear on the screen.



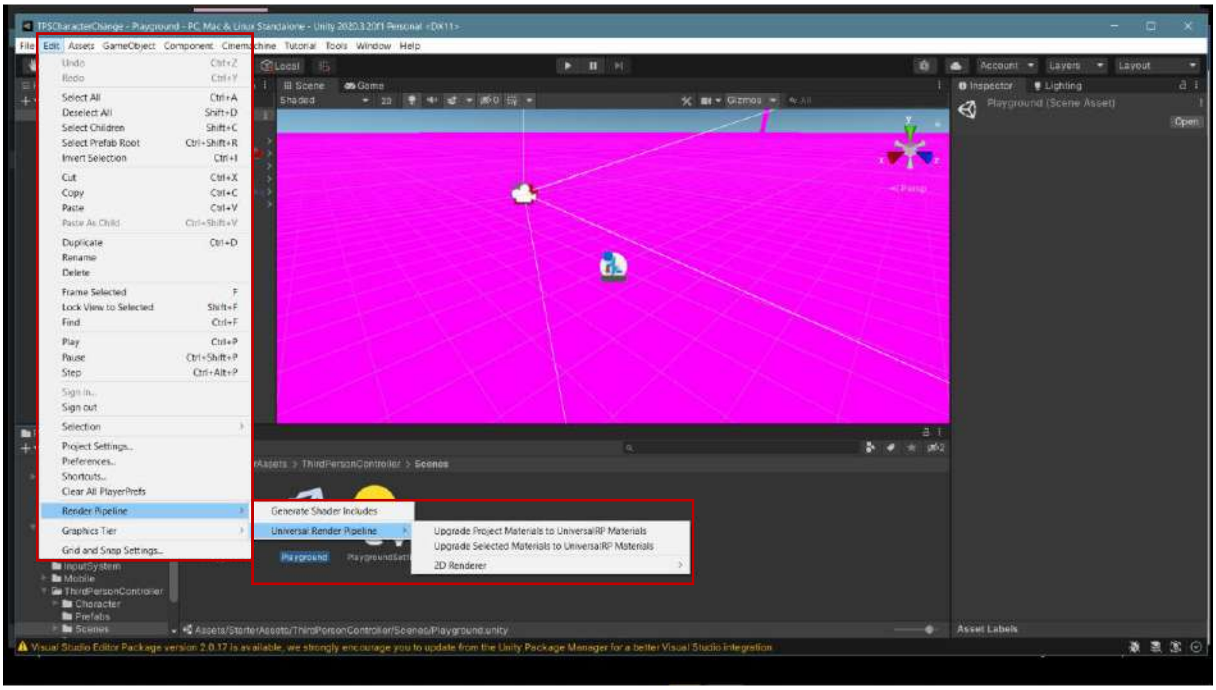
Under **Starter Assets**, double click on **ThirdPersonController->Scenes->Playground** to load the scene.



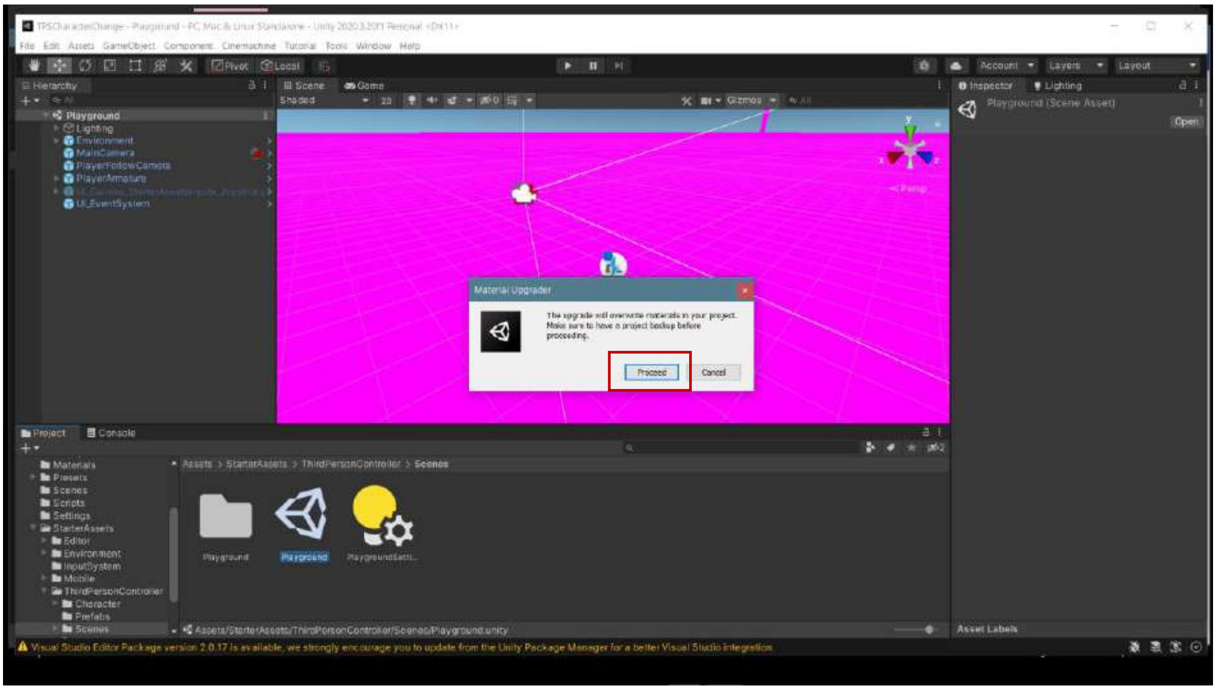
A pink screen will appear because there is an incompatibility between **URP** and **ThirdPersonController (TPS)**.



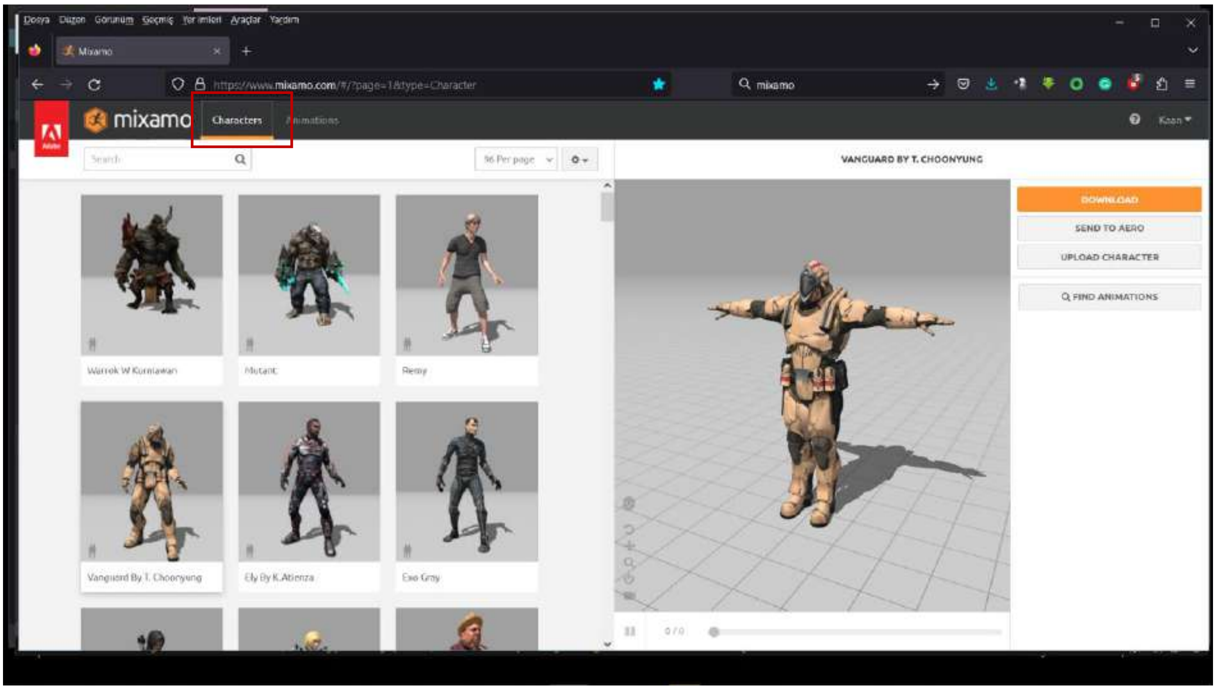
To solve the problem, select E**dit->Render Pipeline->Universal Render Pipeline->Upgrade Project Materials** to **UniversalRP Materials**.



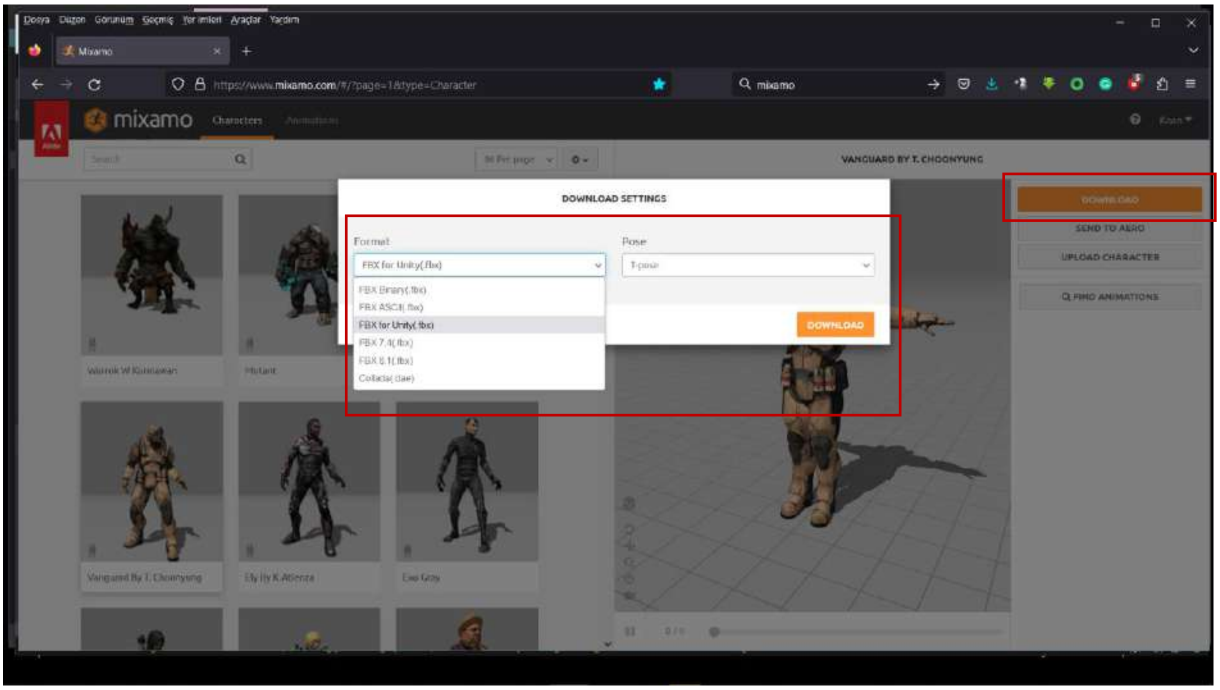
Proceed key will be proceeding here.



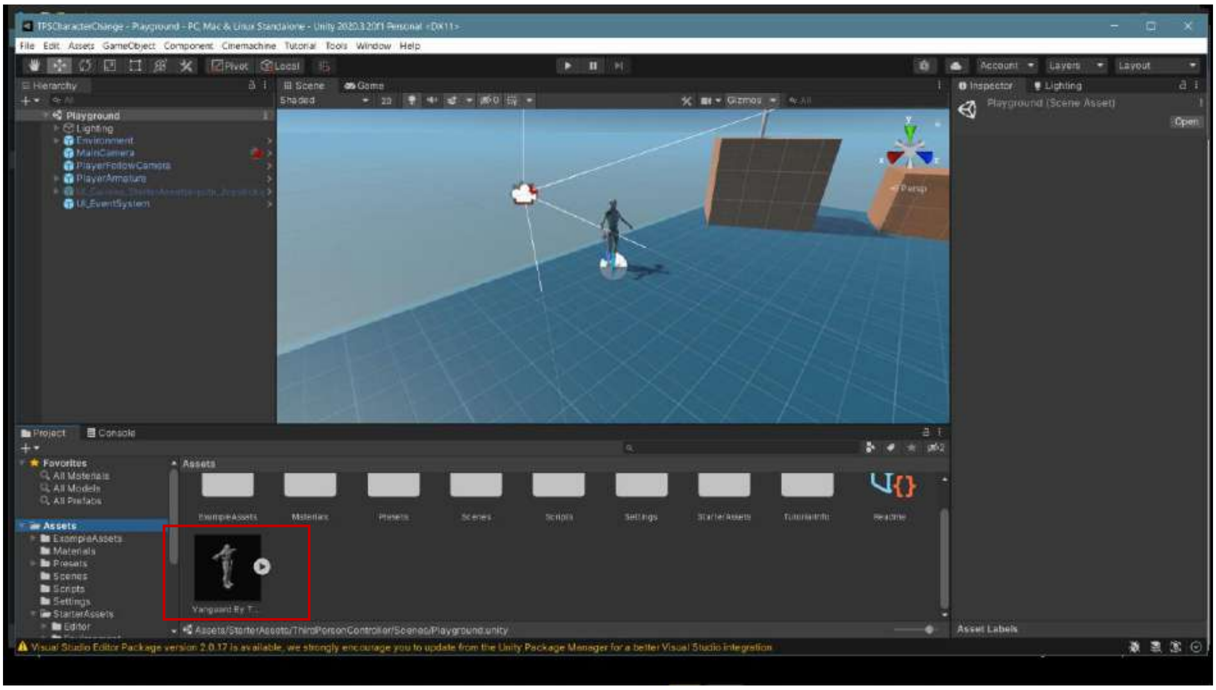
The problem will be solved. Now let's determine the character that we will replace with the robot. For this, let's choose a character from the **Adobe Mixamo** site.



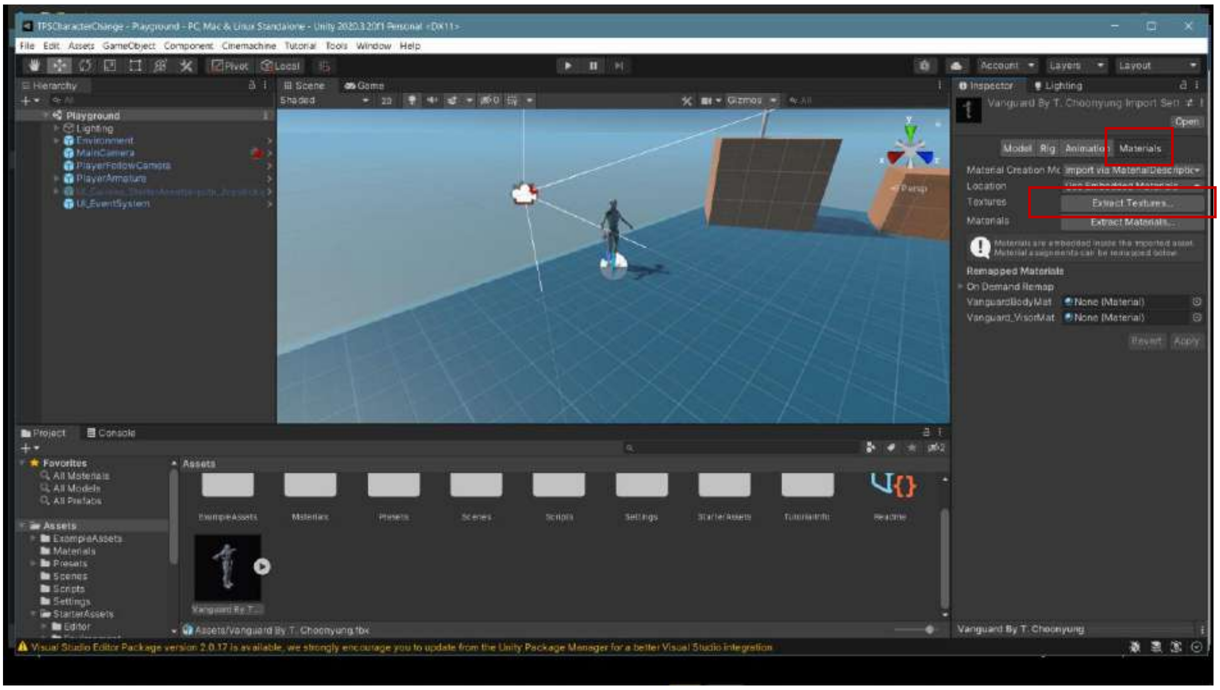
From the window that opens with the **Download** button, select **FBX for Unity (fbx)**.



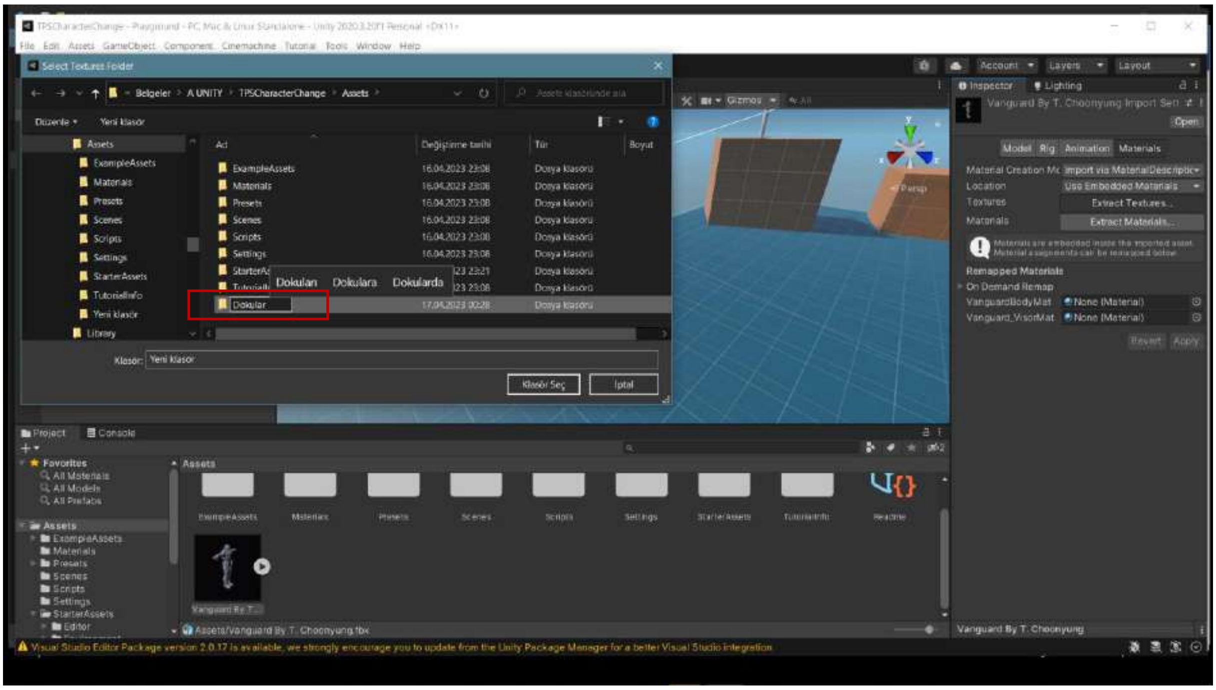
Now, drag the **fbx** file we downloaded into our Unity project (maybe under Assets).



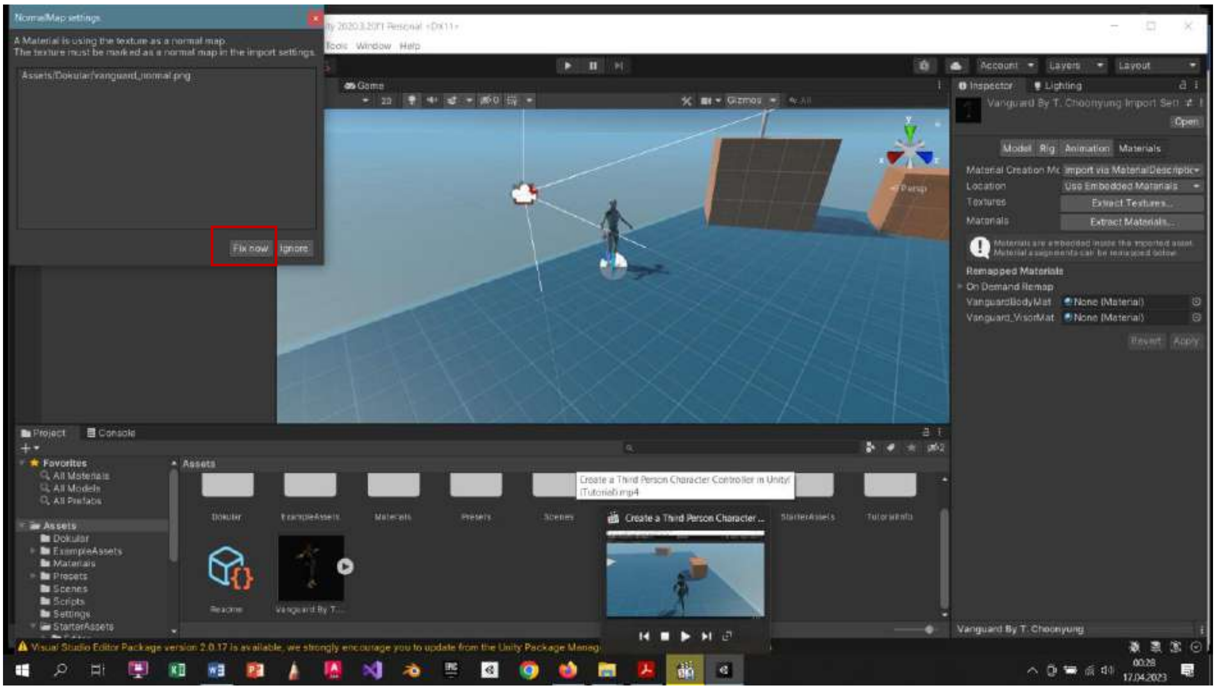
The character is completely white, and the textures are not visible. To solve this, select the file and click **Inspector->Materials->Textures->Extract Textures**.



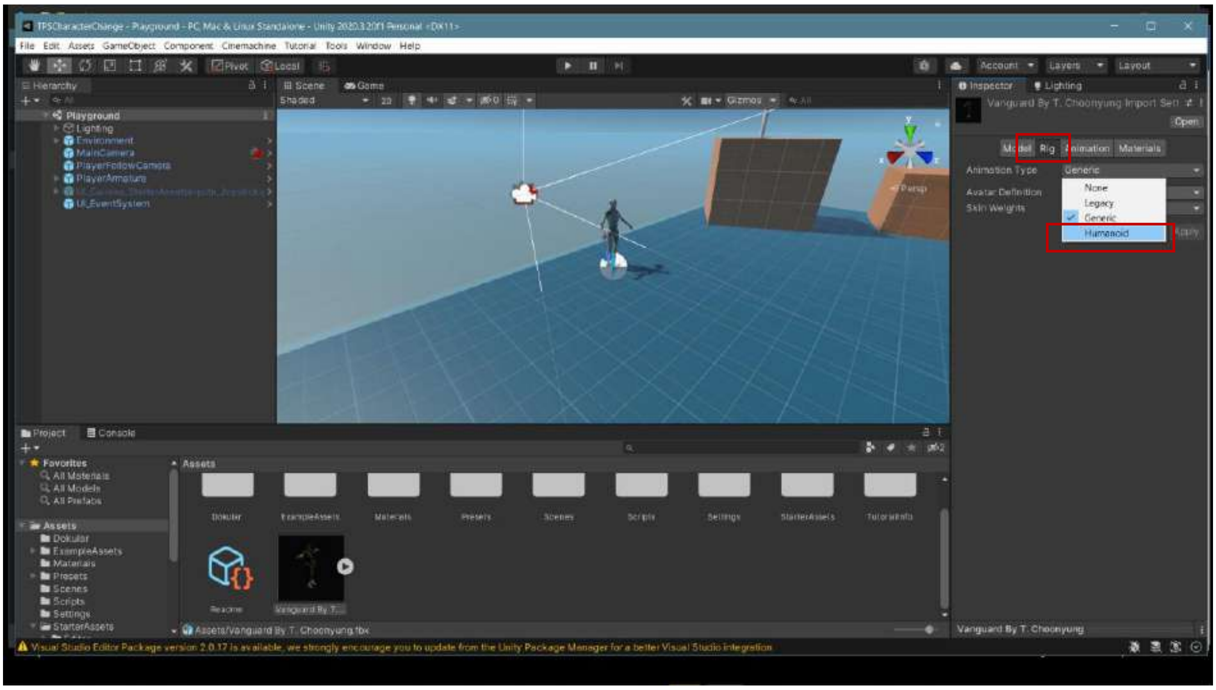
It asks us for a folder name. Open a new folder with the **right mouse button** and select it.



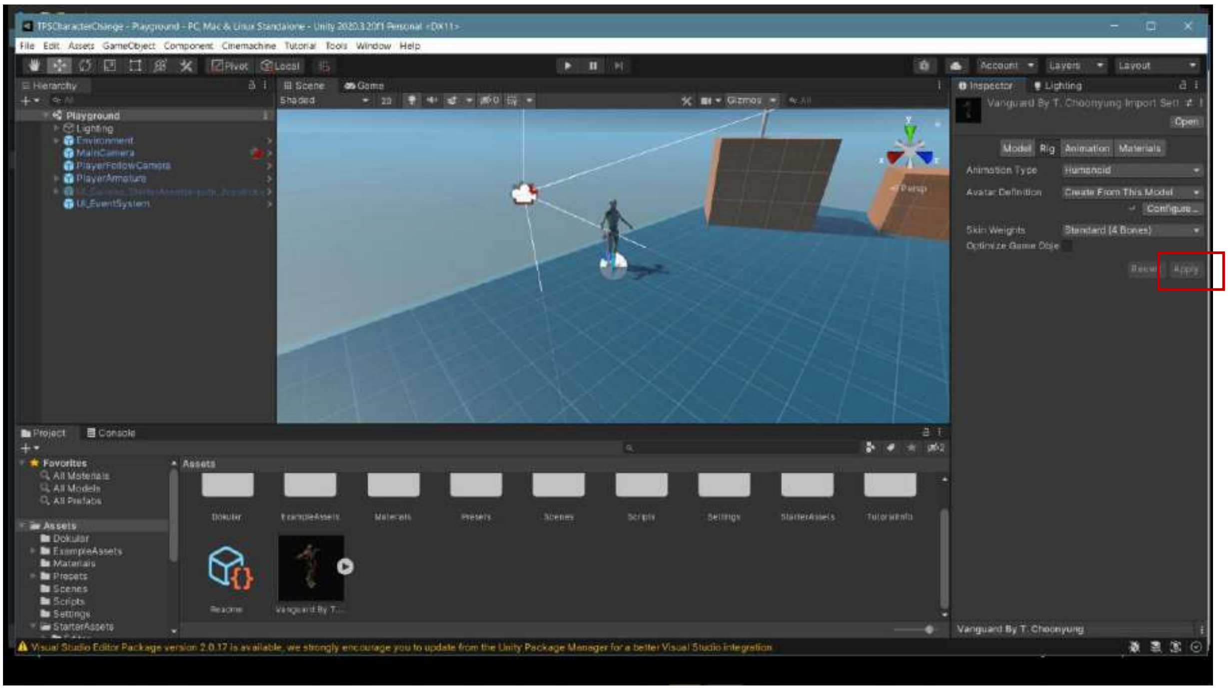
Then, press **Fix** now to solve the material problem.



Also, press **Inspector->Rig->Animation Type->Humanoid**.

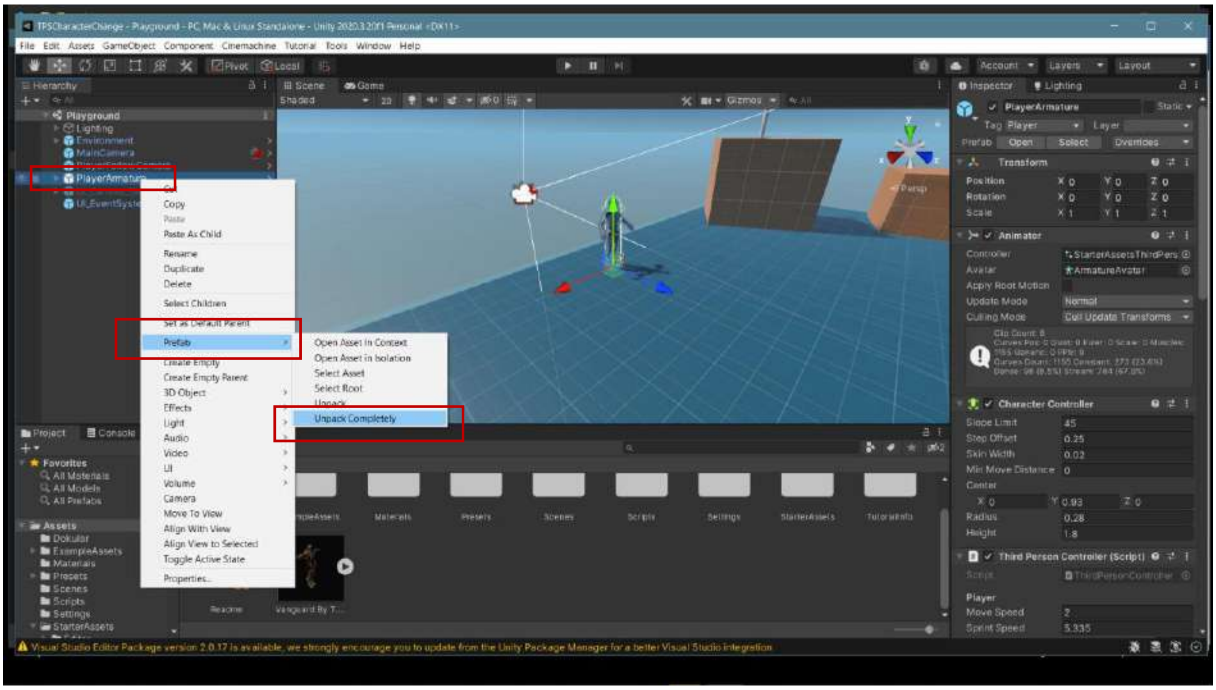


Confirm the changes with the **Apply** button.

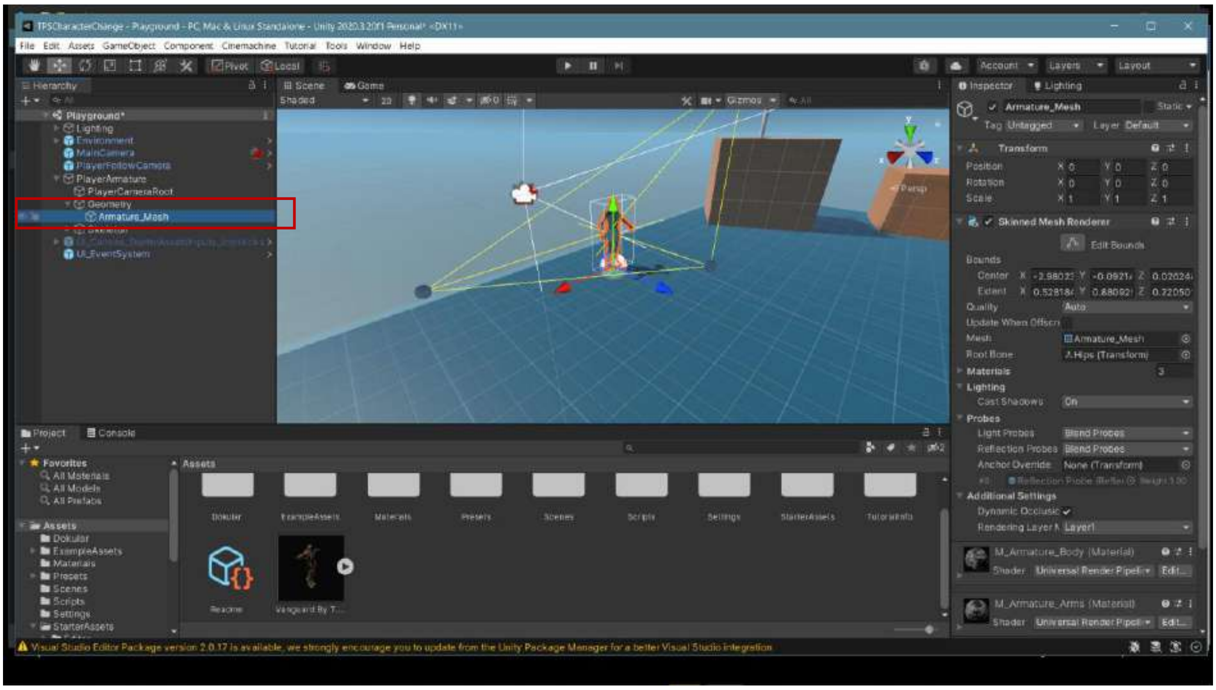


Color and textures are matched.

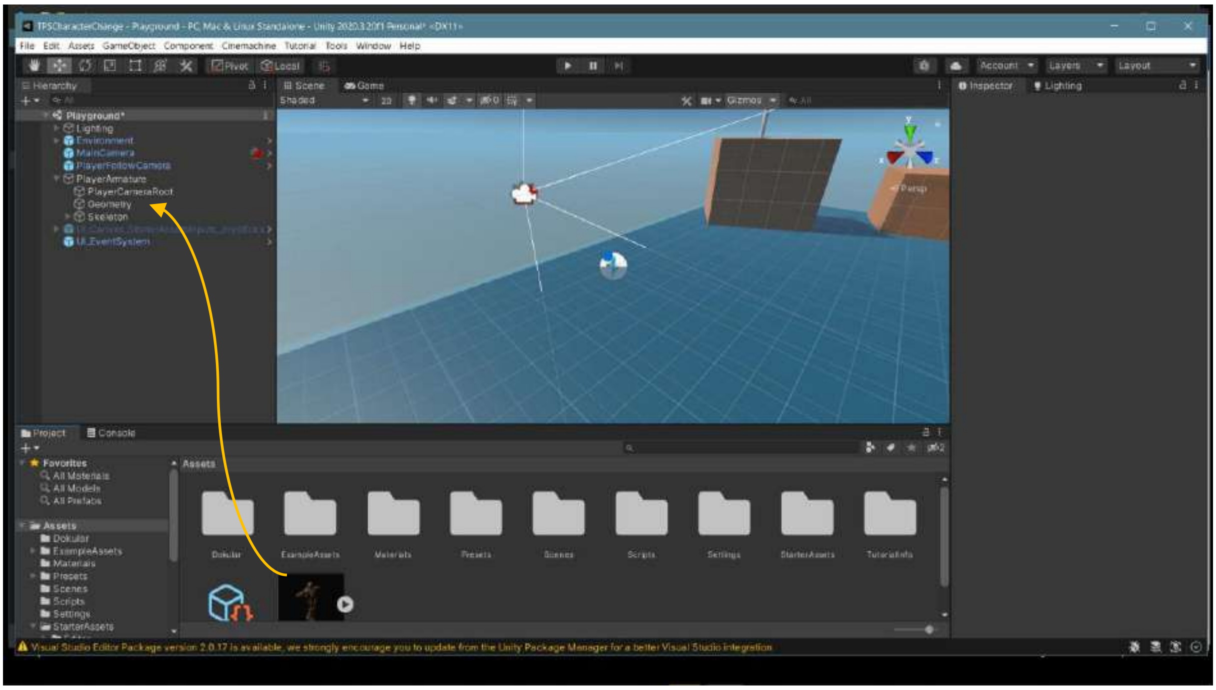
Now, in the **Hierarchy** section, unpack all the sub-parts with **PlayArmature->Prefab->Unpack Completely**.



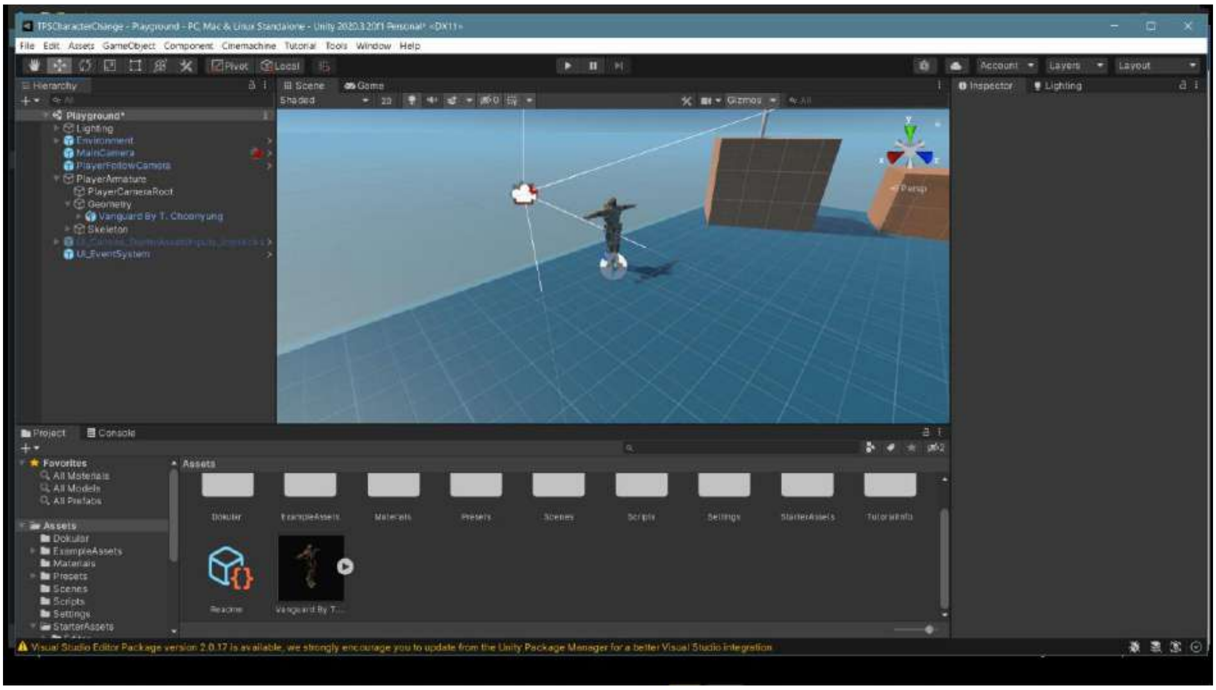
In the next step, to **delete** the robot: **PlayArmature->Geometry->Armature_Mesh** will be deleted. The robot is now removed from the scene.



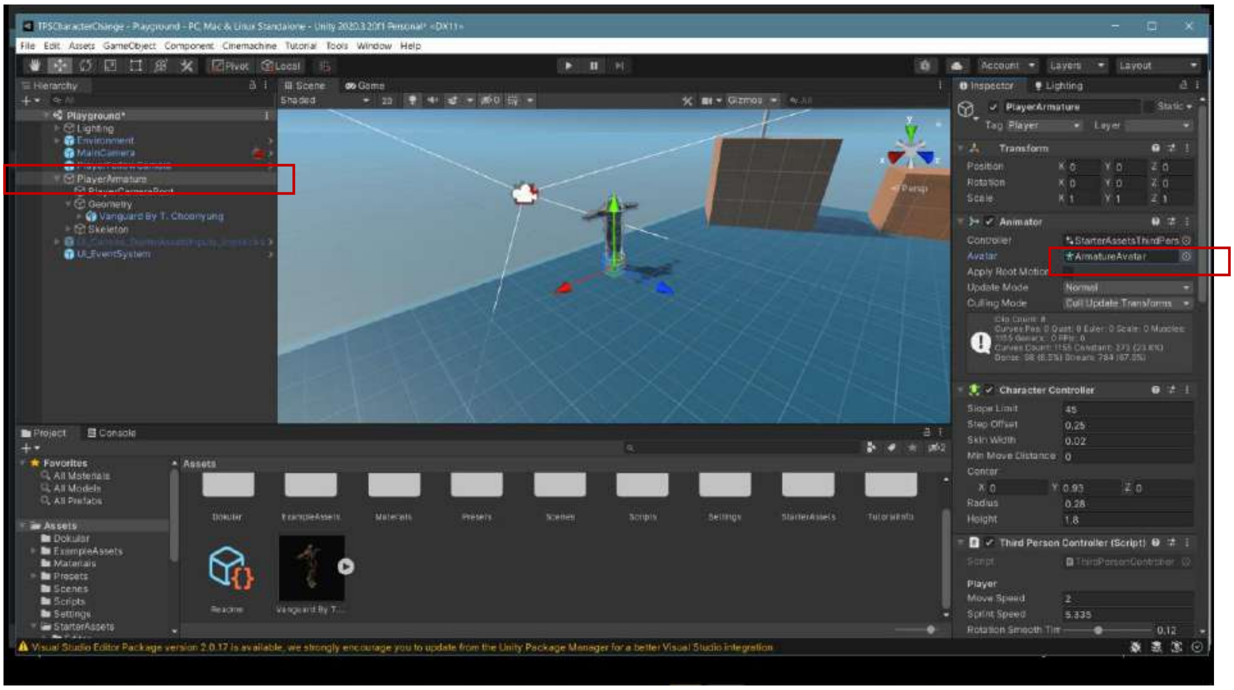
Drag the **fbx** character we downloaded from **Mixamo** and prepared for use under **Geometry**.



Our **fbx** (film box) character came instead of robot.

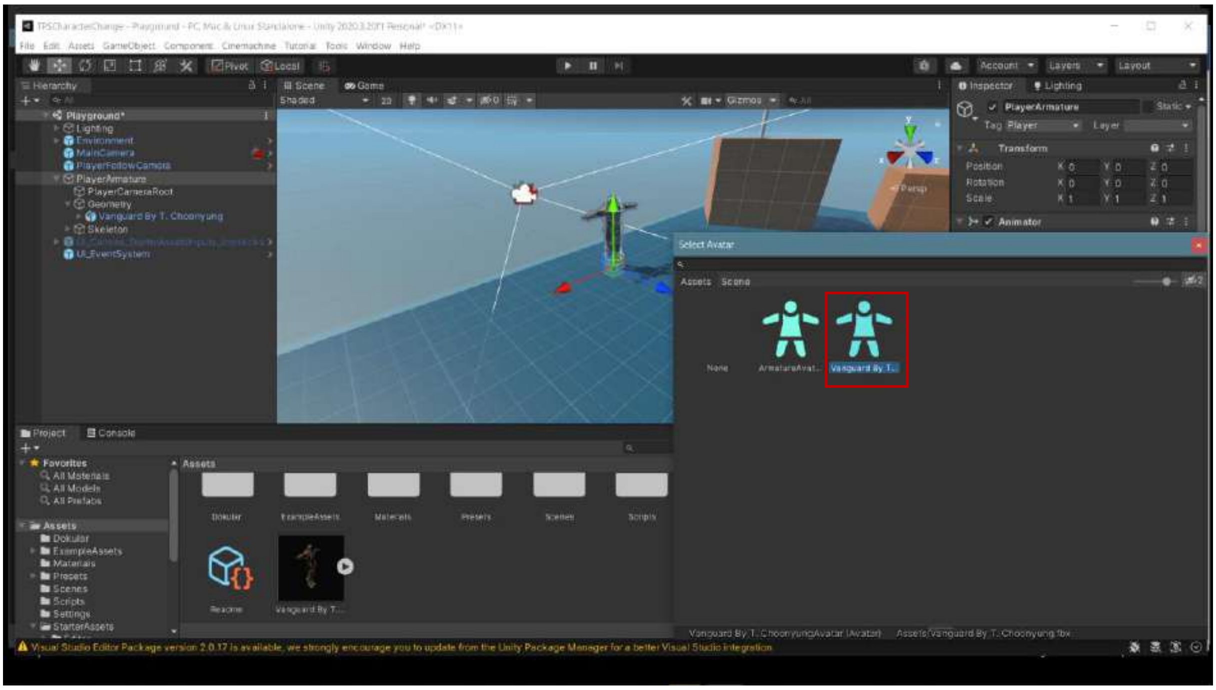


Finally, we need to change the avatar. To do this, open the list by pressing **PlayArmature>Inspector>Animator>Avatar**.



Select the **avatar** to which we added the **FBX** file.

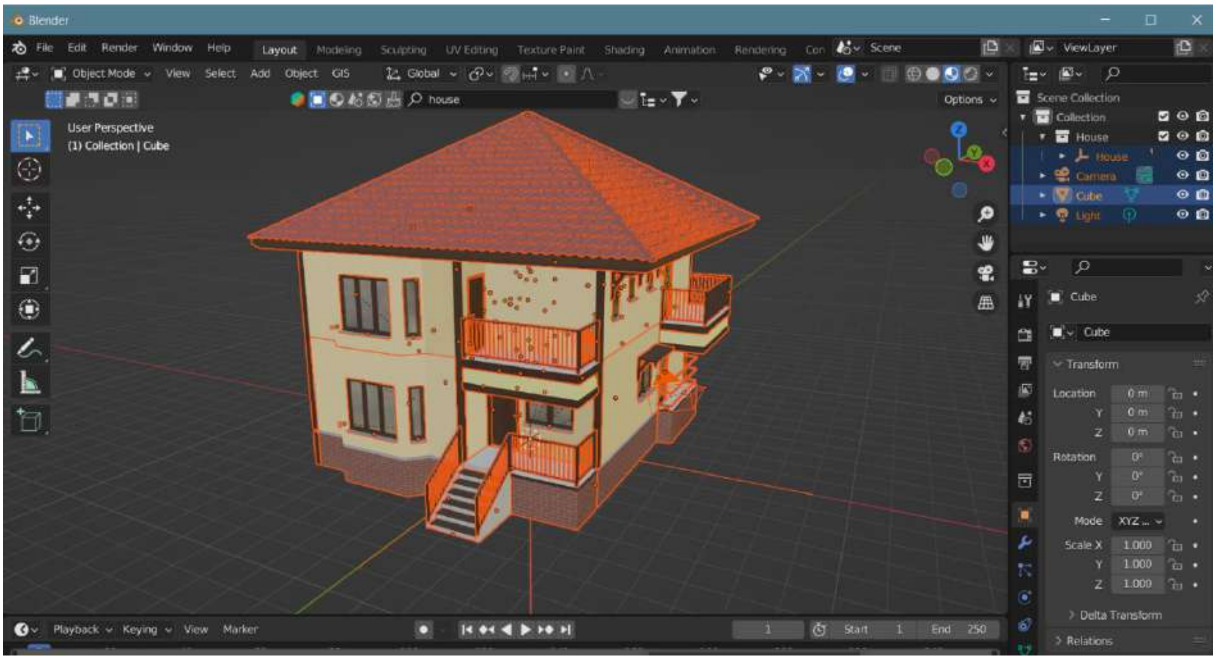
Now, in the game mode, there is the character we selected instead of the robot.



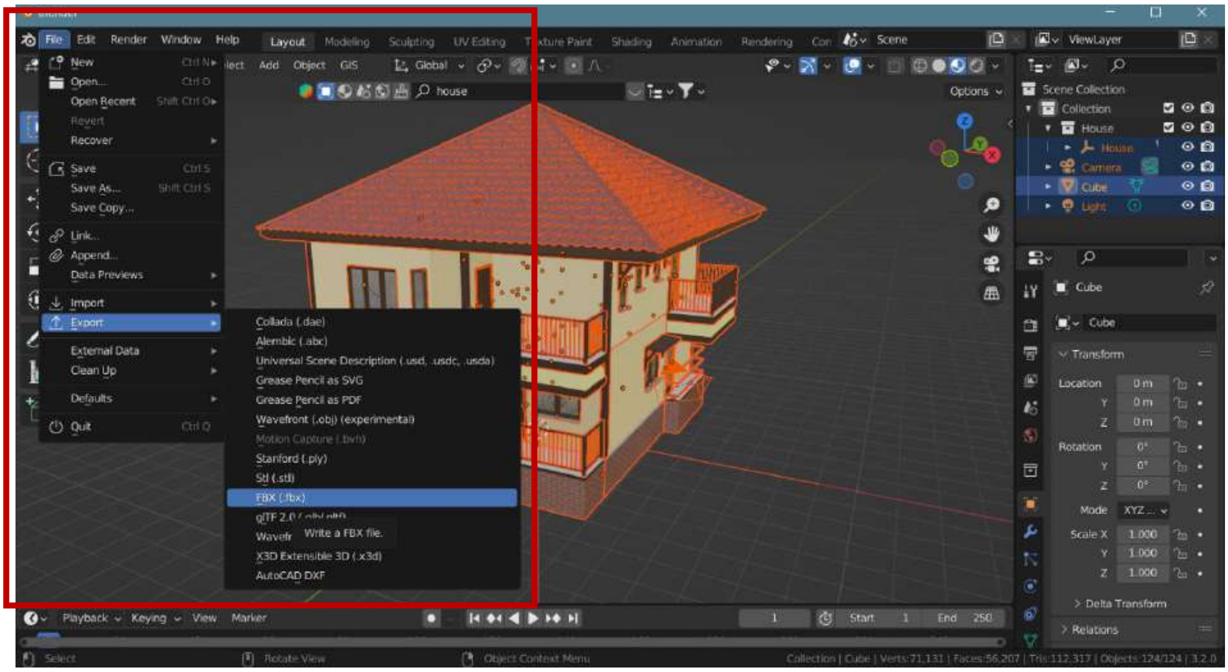
Similarly, a **mine worker** and the environment can also be used for the TPS application.

9.4.Transferring Blender Designs to Unity

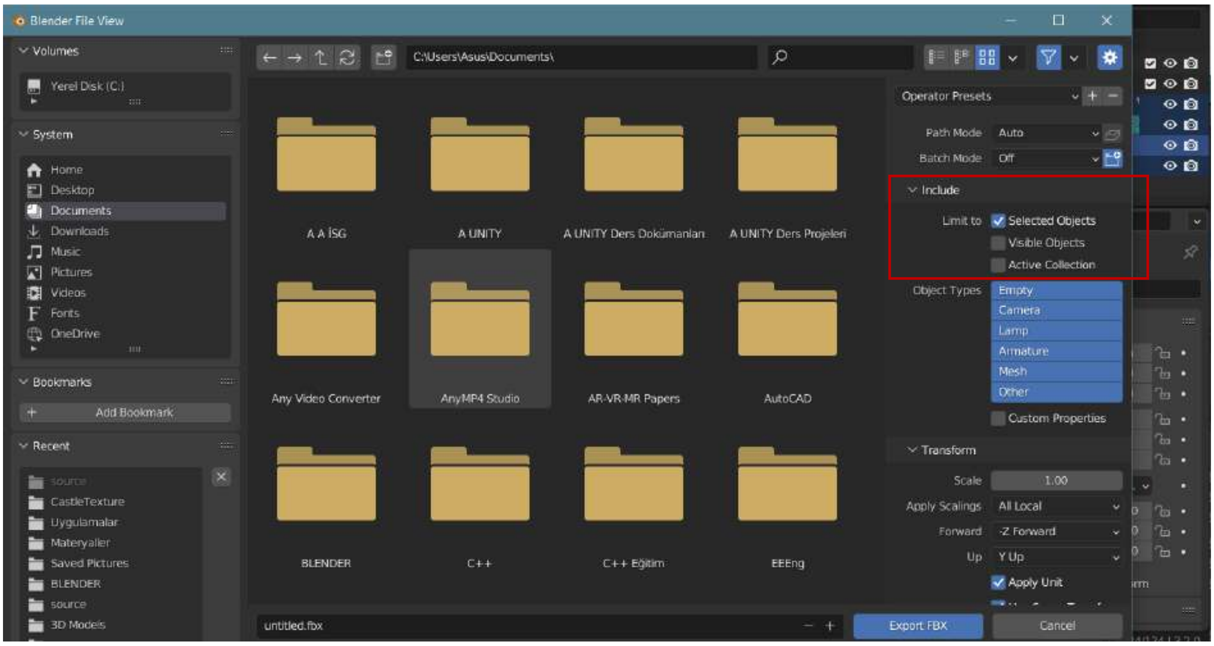
Blender is one of the most important design and animation packages for Unity. However, the files may not be fully transferred every time. In order to ensure the visibility of textures and colors when transferring Blender scenes or objects to Unity, the entire scene developed in Blender or certain object(s) is first selected.



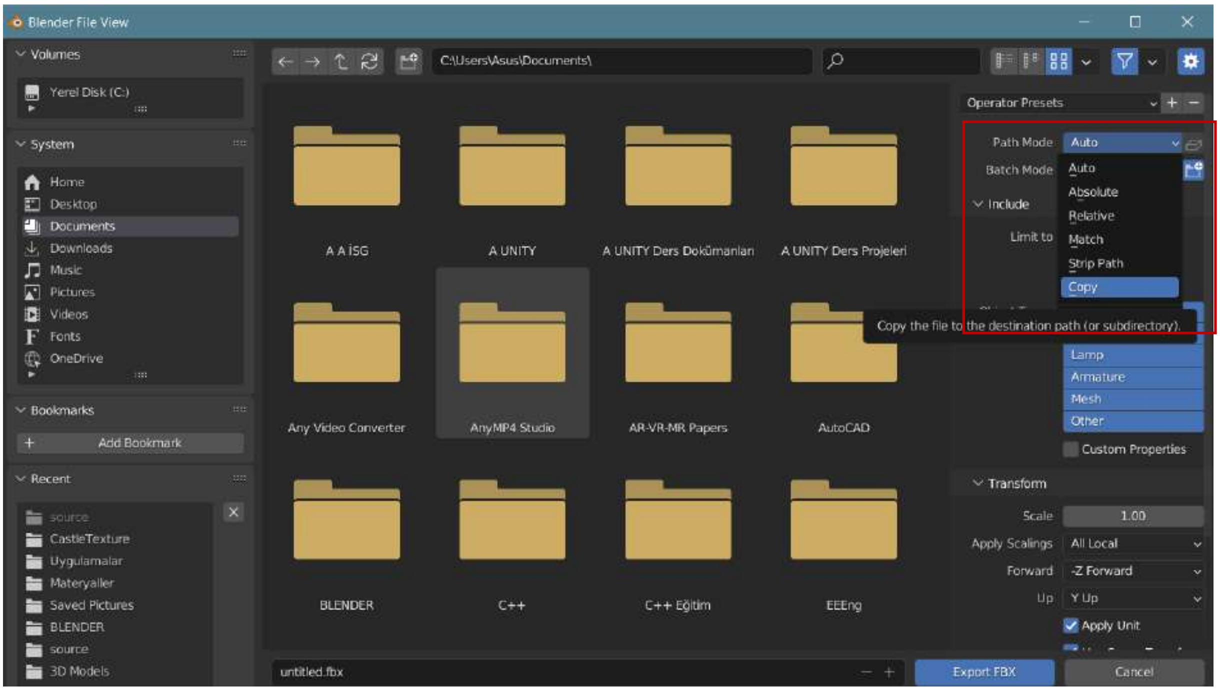
Then, click on **File->Export->FBX (.fbx)** option.



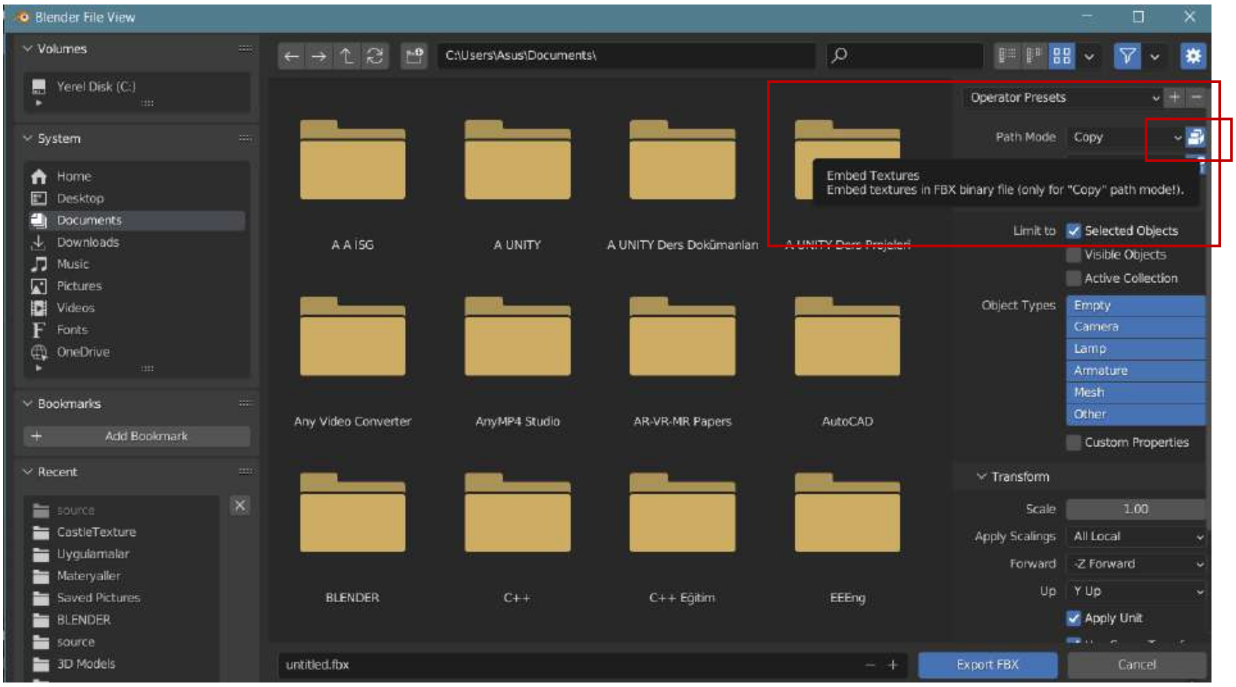
In the window that opens for recording, the following operations are performed in order: **Include->Limit** to section is clicked and **Selected Objects** is activated. Thus, the scene or object(s) we selected are included in the export process.



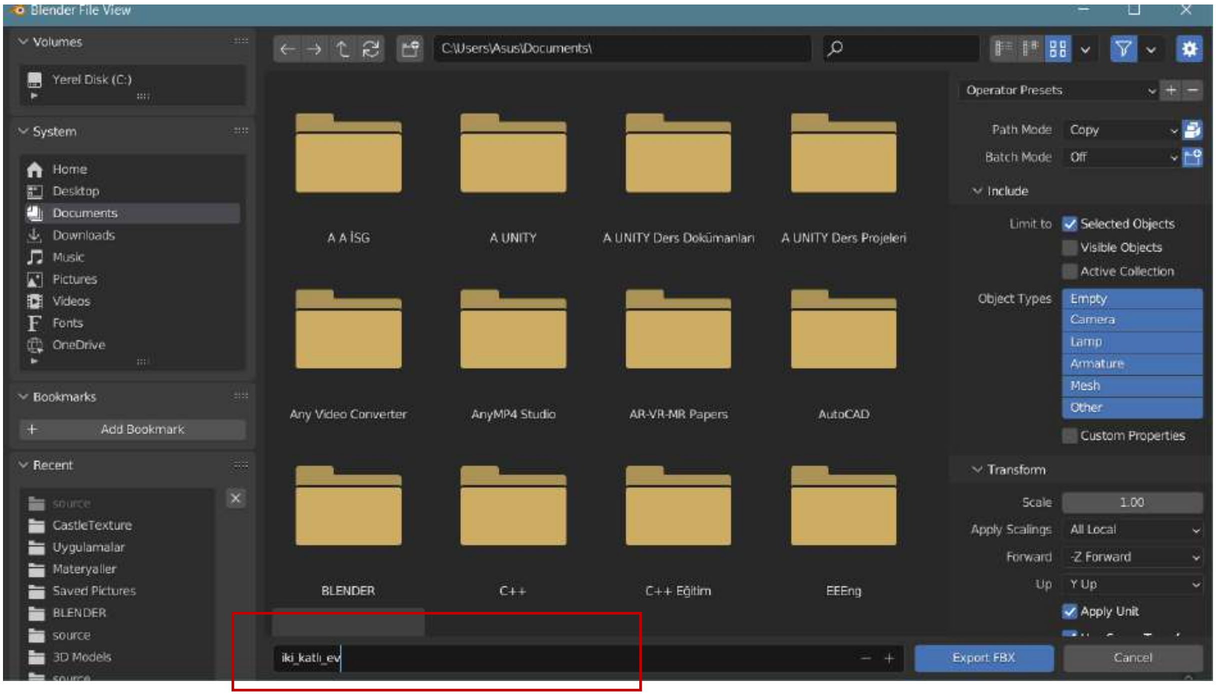
Copy is selected under the **Path Mode** section.



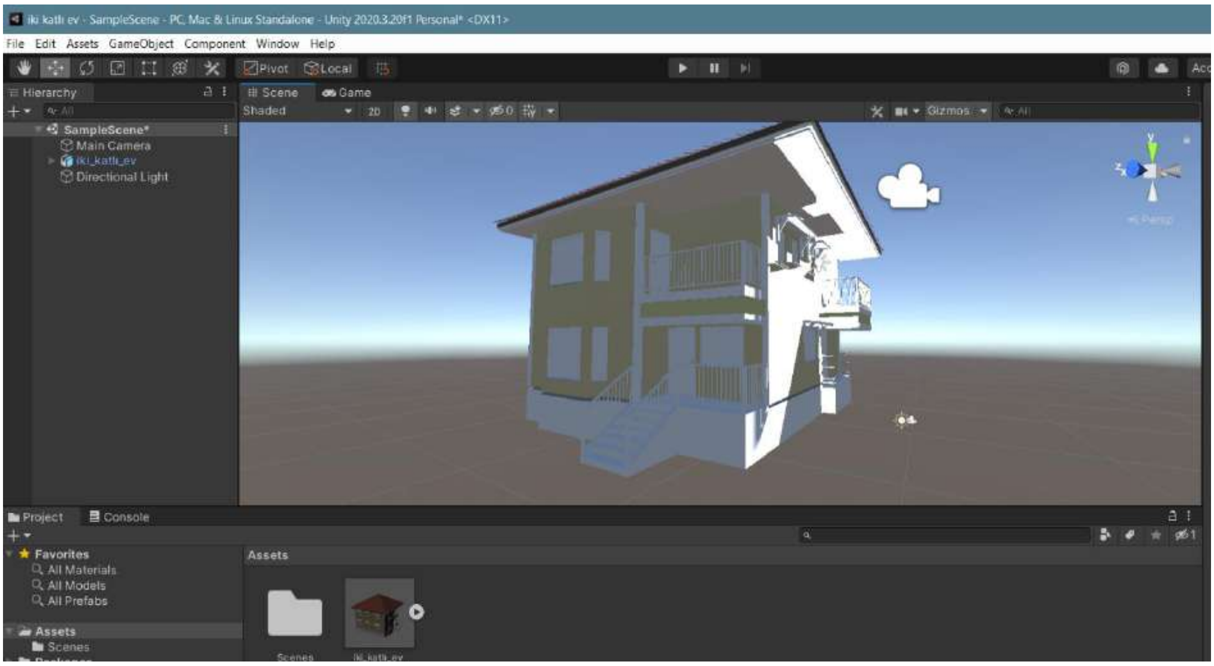
Again, the **Embed Textures** box next to **Path Mode** is selected and activated.



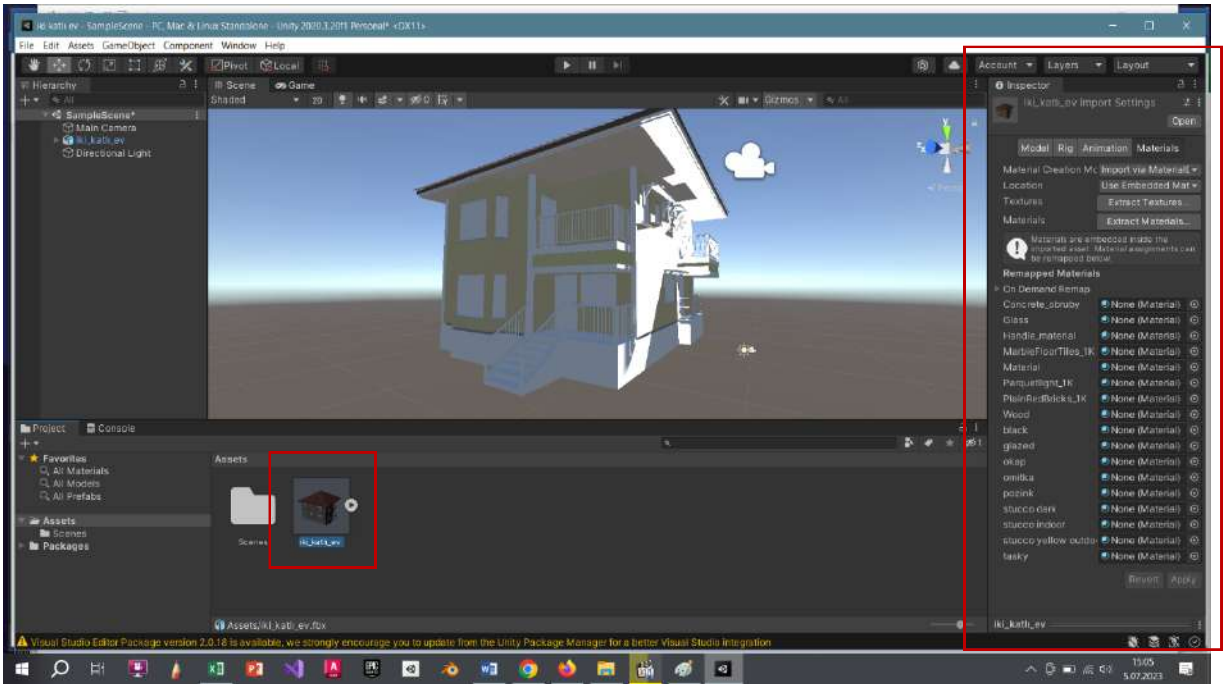
Then, give our file a name under the folder we specified and save it.



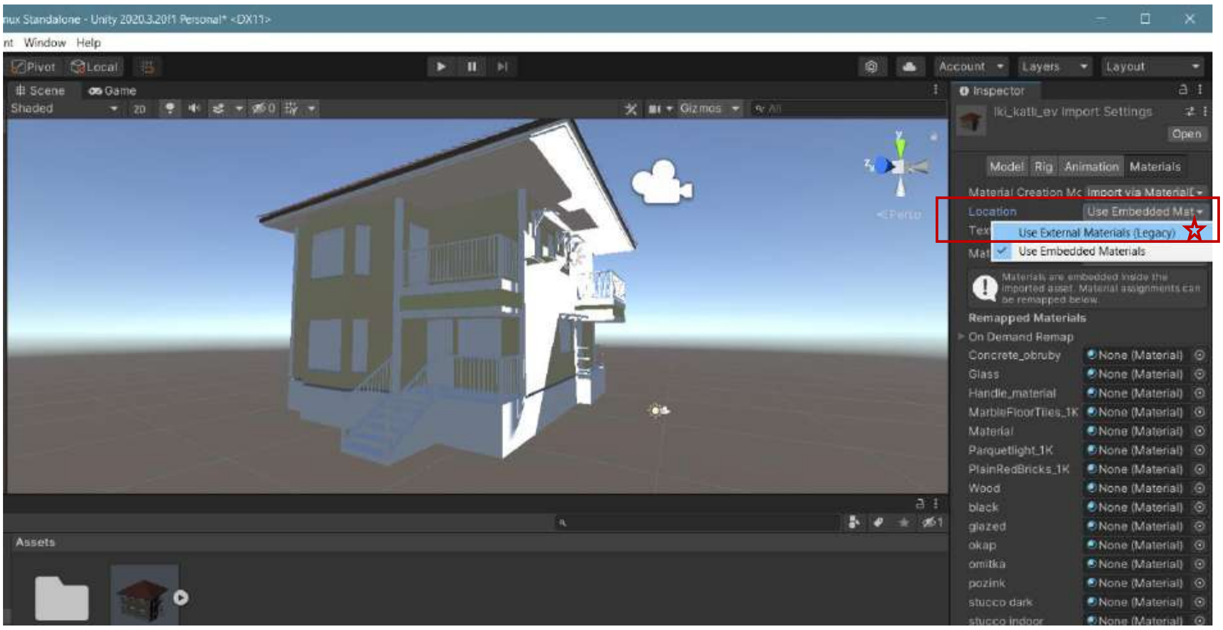
Now, come to our Unity scene. Drag the **FBX** file to the Assets section.



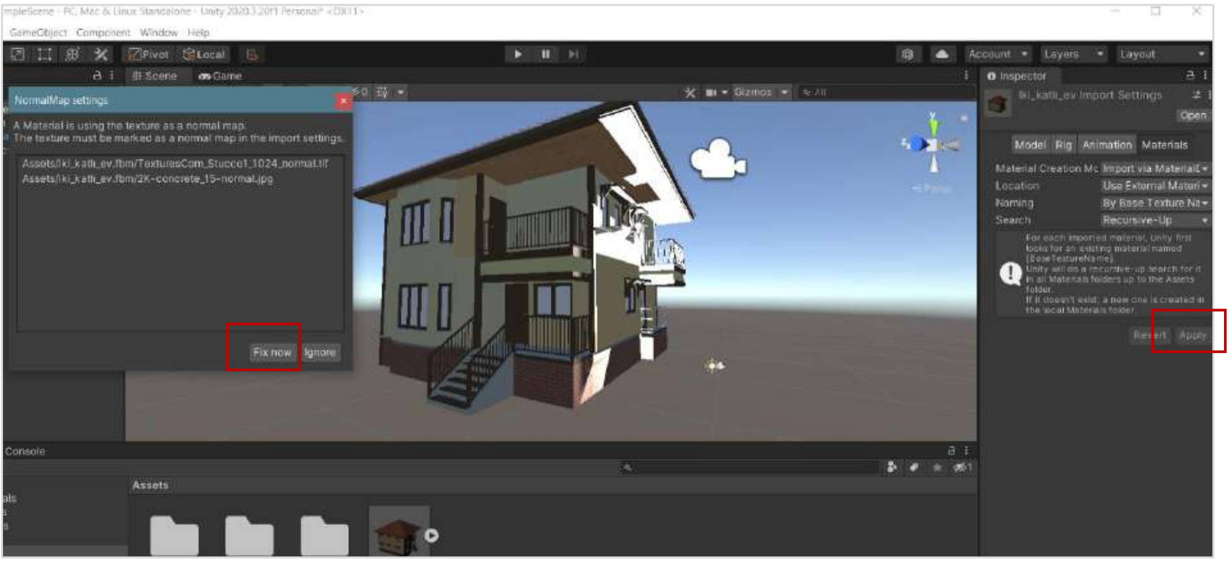
It seems that the textures and coatings have not arrived completely. Now, select our **FBX** object in the Assets section and look at its settings in the **Inspector** section.



Select **Materials->Location->Use Embedded Materials (Legacy)**.



Then, click **Apply**. In some cases, click **Fix Now** in the window that can be opened for customization purposes.



As you can see, the textures, colors and textures in Blender have been transferred.

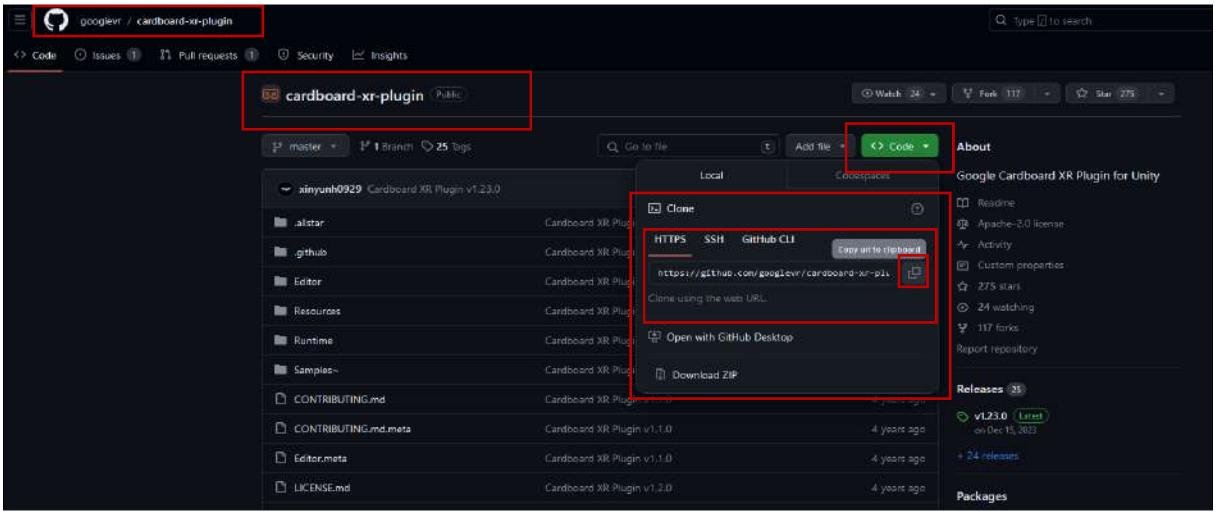
10. VIRTUAL REALITY – VR APPLICATIONS

10.1.Virtual Reality (VR) Application for Cardboard Devices

It is possible to say that **cardboard**-type **headsets** are the most economical VR headsets for experiencing virtual reality applications. These **stereoscopic** (superimposing 2 images) based devices have a chamber where mobile phones can be placed. Therefore, the main application output platform is mobile devices such as mobile phones and tablets.

It is possible to print cardboard-type devices with Unity. For this purpose, the **Google Cardboard XR Plugin** provides a template that will make the process easier.

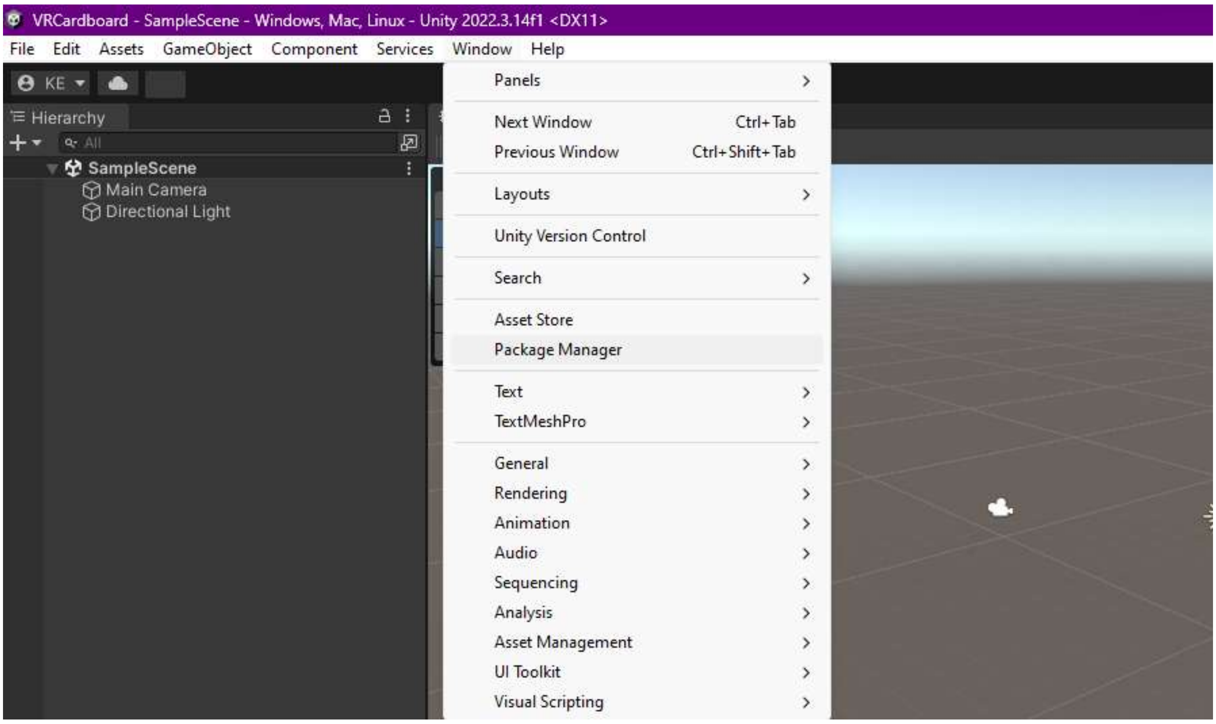
Let's open the address https://github.com/googlevr/cardboard-xr-plugin on the **GitHub** site where the plugin is located. Copy the link by clicking on the link in the **HTTPS** in the window that opens in the **Code** section.



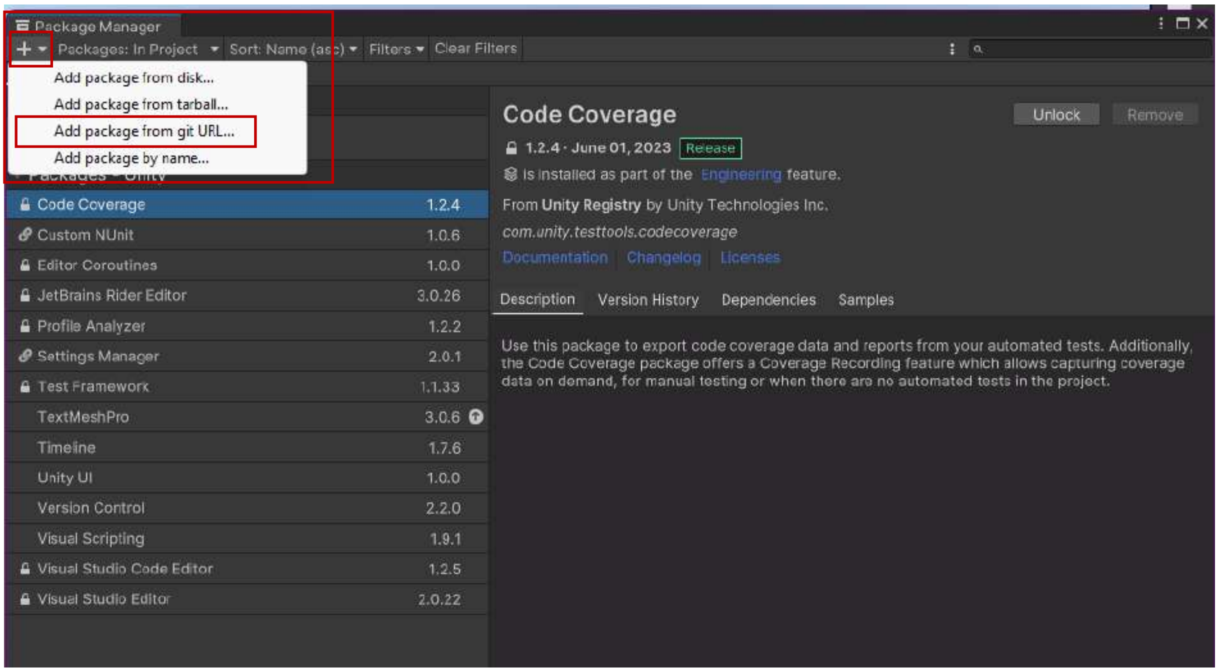
After this preliminary preparation, develop a simple Google VR Cardboard project in Unity.

In **Unity 2021.3** or later (here 2022.3.1.14f), create a 3D project. We named our project **VRCardboard** in this application.

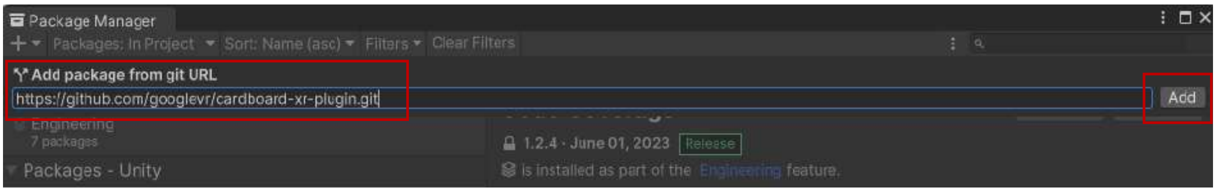
Go to **Window>Package Manager**

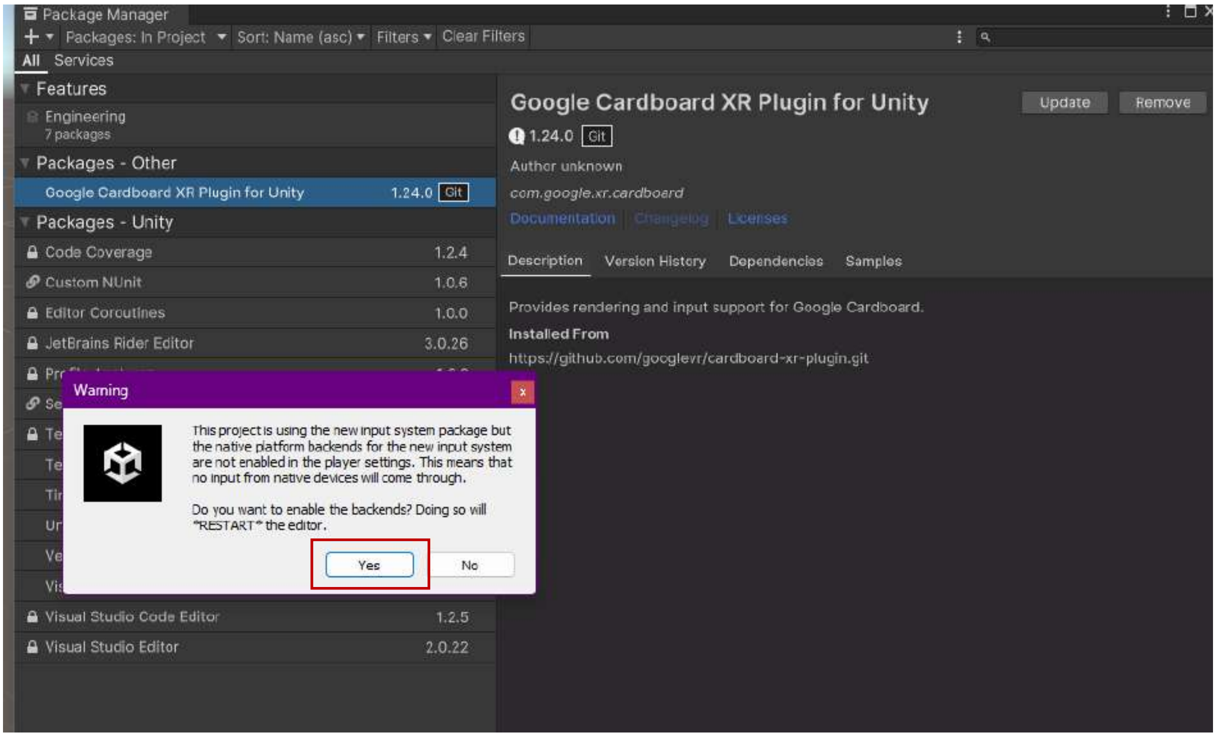


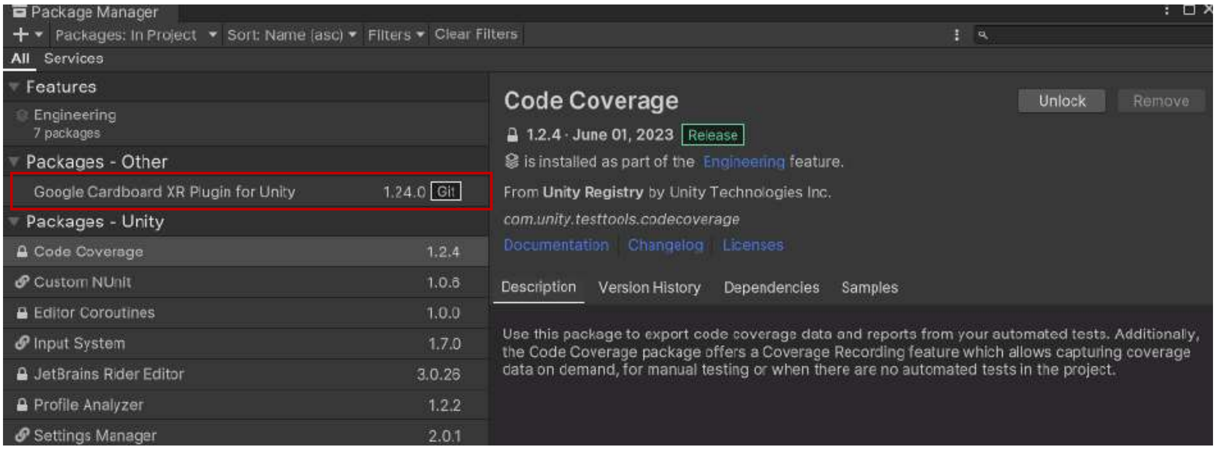
Open the at the menu by pressing the **+** key here. Select the **Add package from git URL**... line.



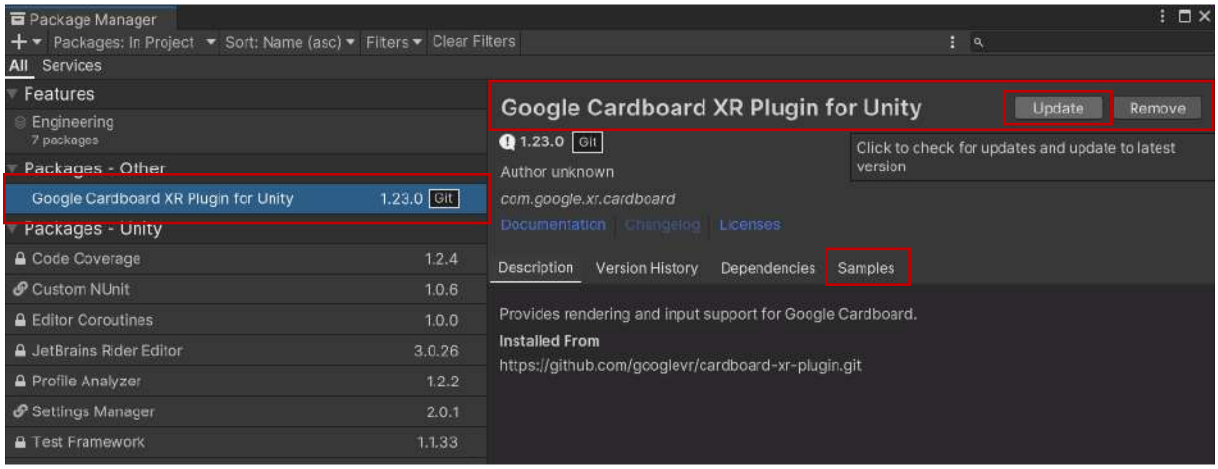
Paste the link we just copied here and press the **Add** button.



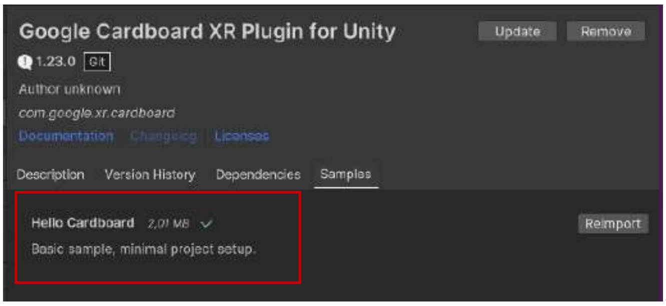




The **add-on** is now installed in our package manager. If there is any **update** suggestion, we can do it by clicking **Update**.

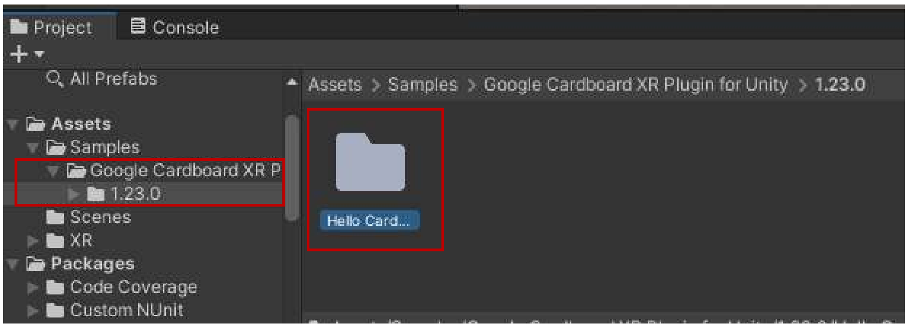


Since the preset scene, which is a **template** for the package, is located in the **Samples** section, open it and **import** the sample scene named **Hello Cardboard**.



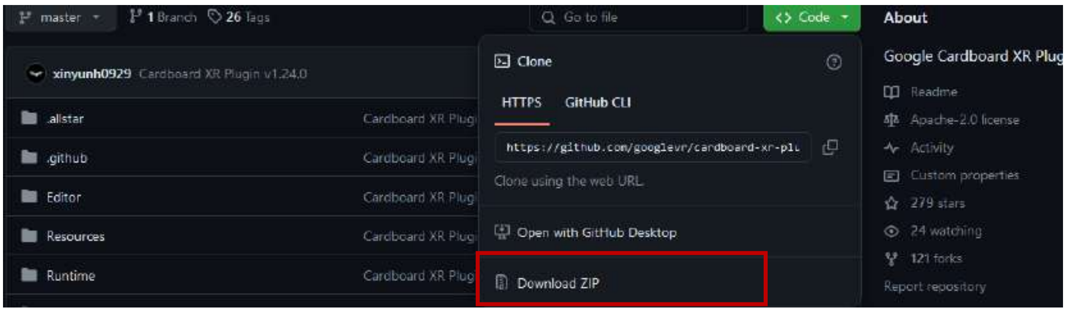
.

After this process, we can see that the folder is placed in the **Project** section.

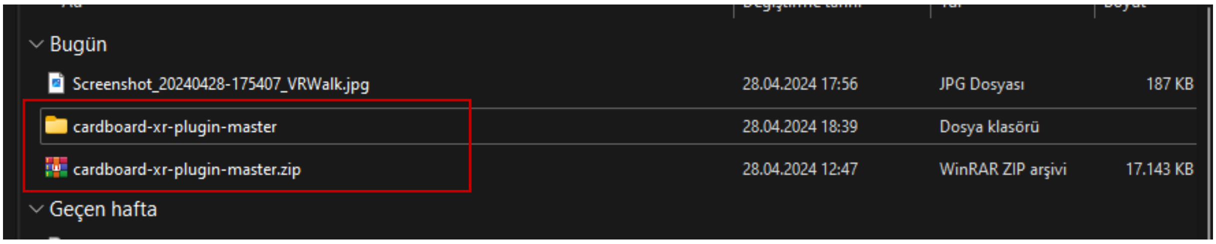


We can close the package manager.

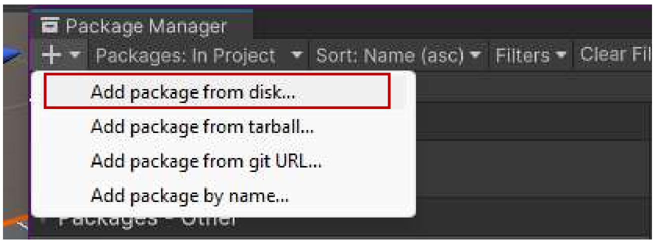
Some computers may not be able to do this, so another way to get the package is to download it as a **ZIP** file.



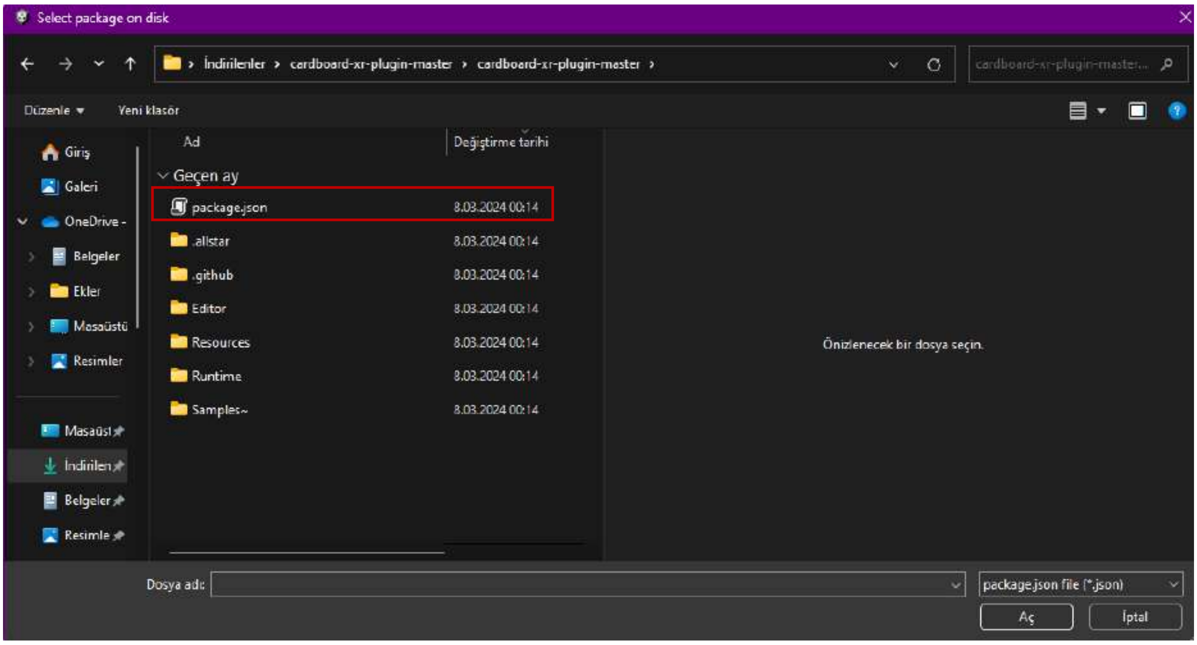
After the compressed file is downloaded to the hard disk, open it and extract it.



In the **Package Manager**, press **+** and click the **Add package from disk** option.



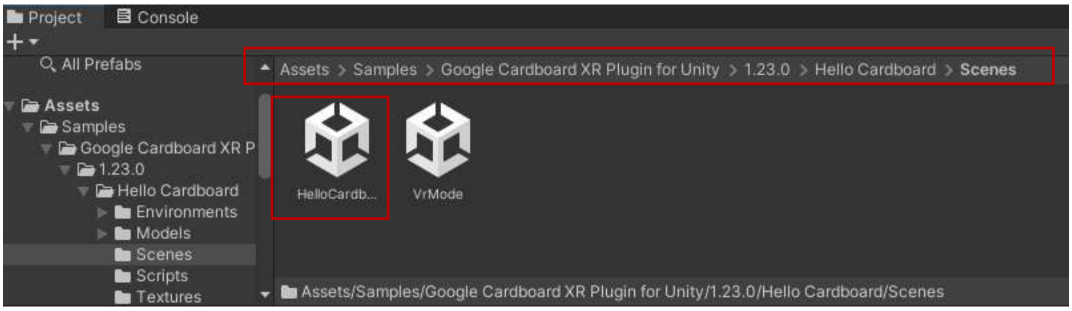
The file type Unity is looking for here is **JSON**. Therefore, we can include the **Google VR** package in our project by selecting the **<u>package.json</u>** file.



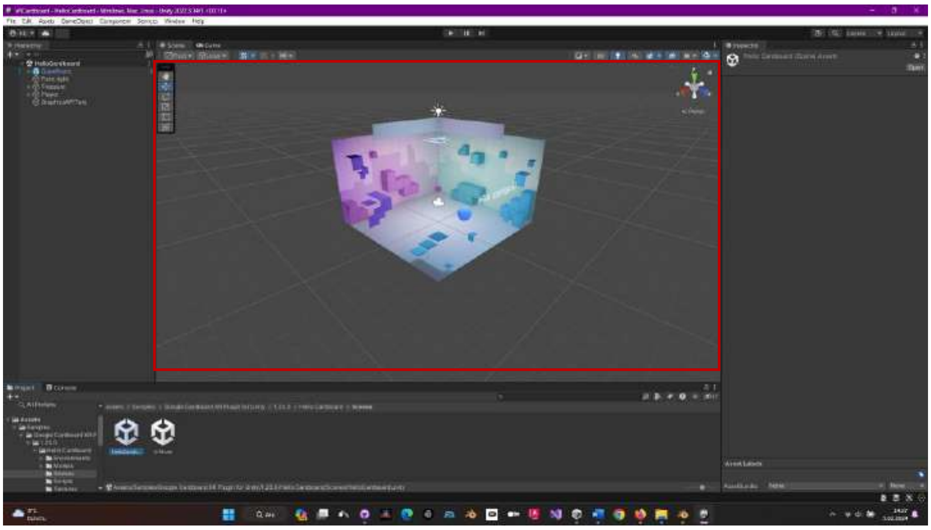
The next part continues as described above. Restarting, Update, Sample addition operations are the same.

10.2.Configuring the HelloCardboard Scene

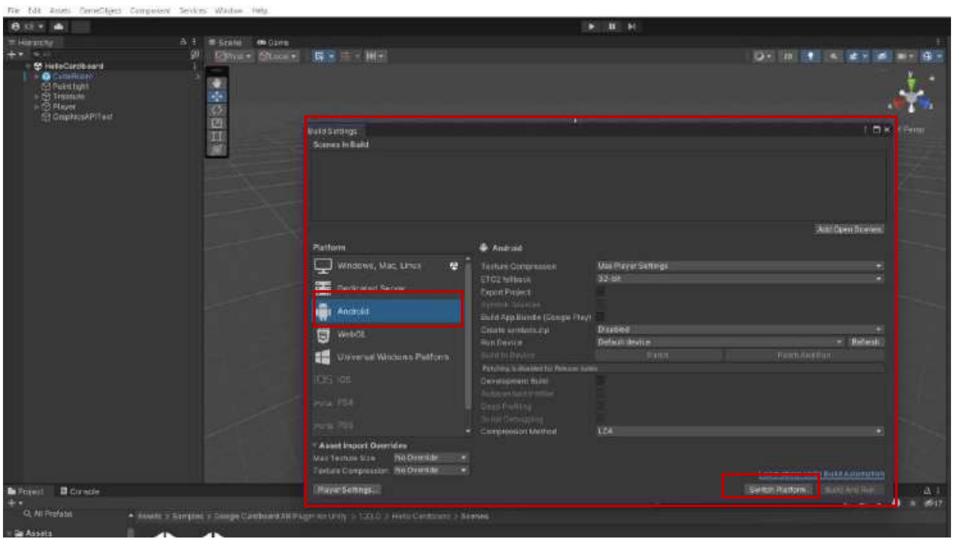
To open the sample scene, go to **Assets/Samples/Google Cardboard/<version>/Hello Cardboard/Scenes.**
Let's select **Add Open Scenes** and select H**elloCardboard**.



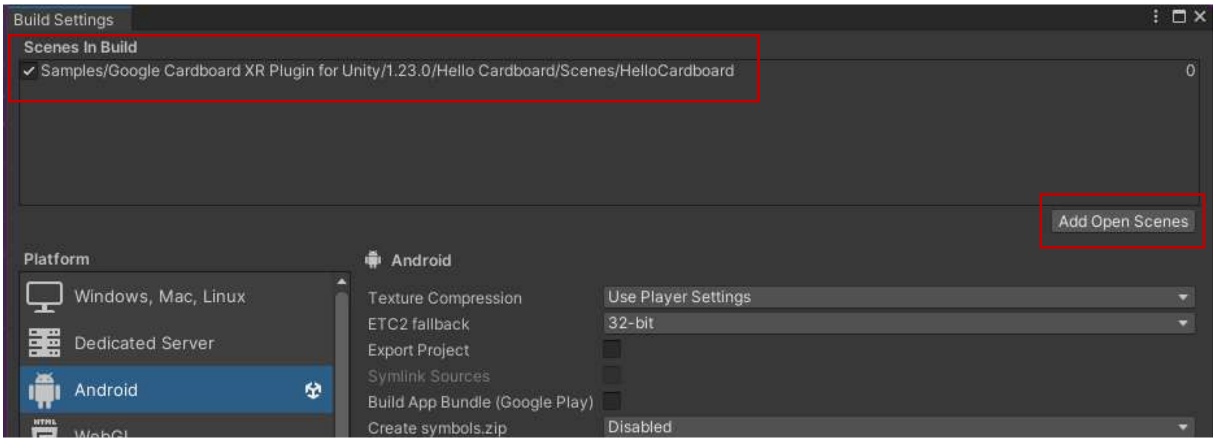
With this selection, the ready scene will appear on the screen. A **VR room** is seen as the content.



Now, let's change our platform to **Android** (IOS for Apple). Select **File>Build Setting>Android** and click **Switch Platform**.

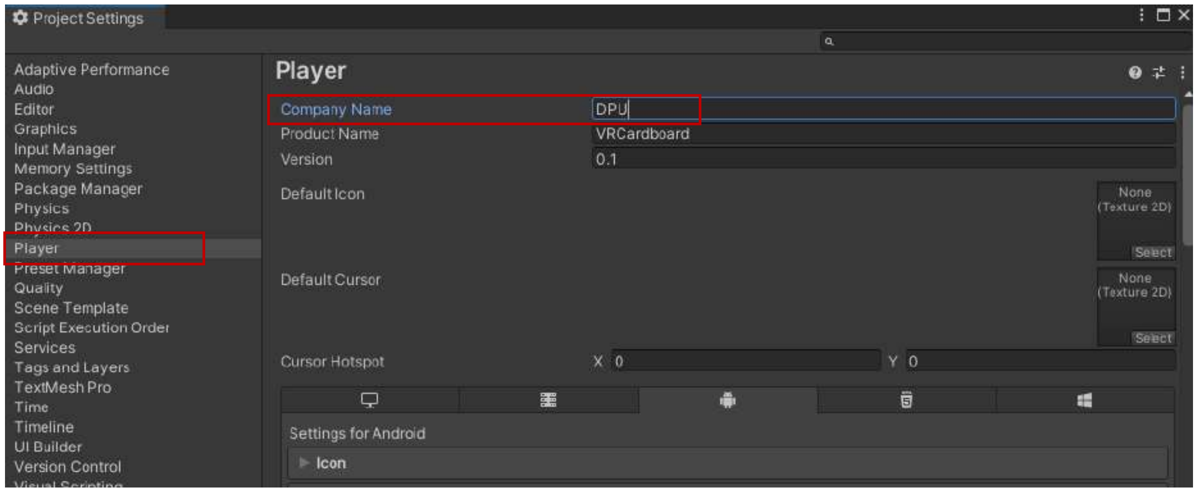


Let's add our current scene to the **Scenes** in **Build** section by clicking **Add Open Scenes**.



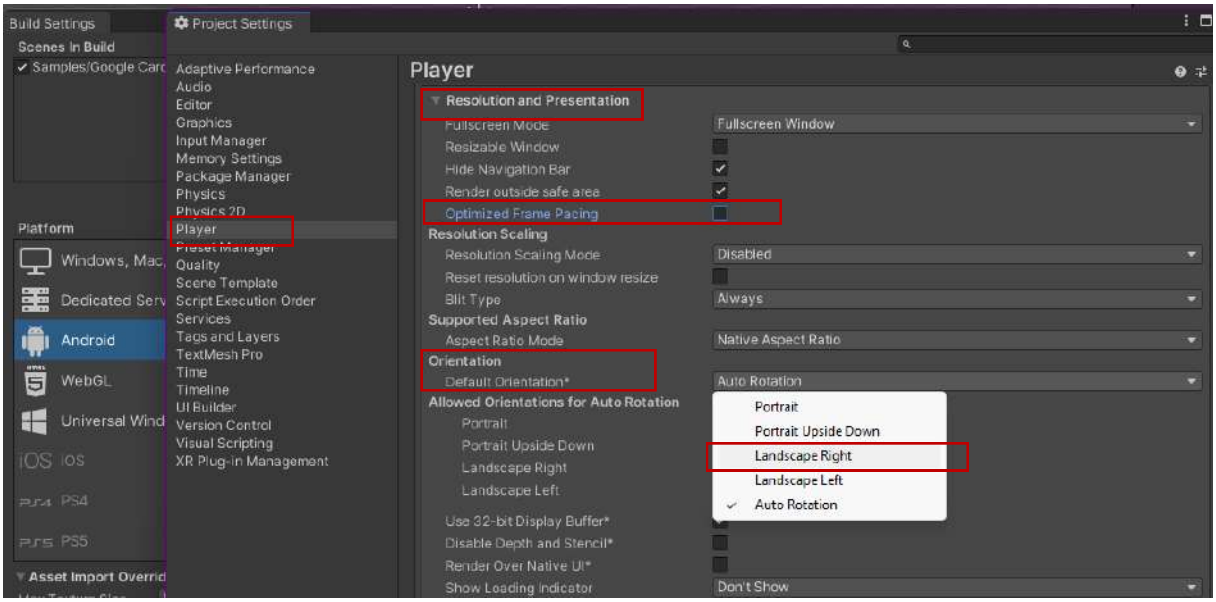
At this stage, we need to make a series of settings. In the **Build Settings** window, click on **Player Settings**. The titles in the settings may vary depending on the version. In this study, version 2022.3 is taken as basis.

In **Player>Icon** section, **Company Name** and **Product name** can be changed. Here the company name is changed to **DPU**.

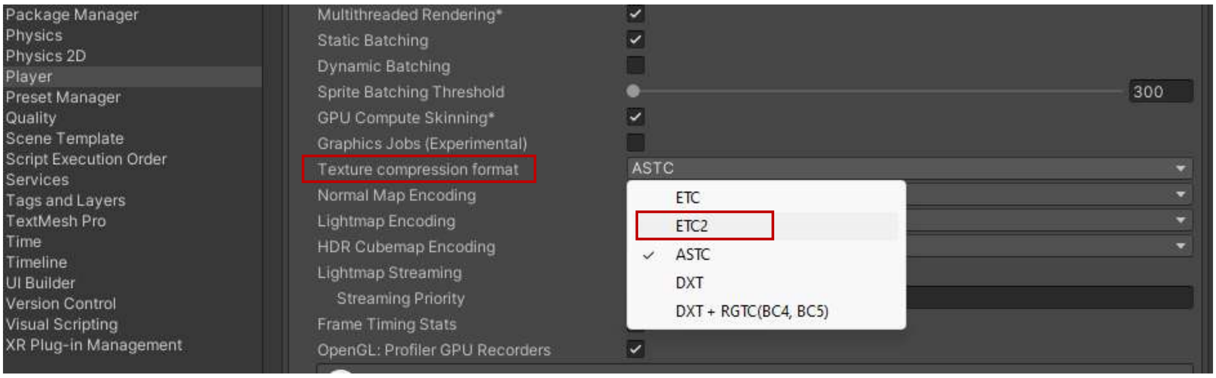


In the next step, go to the **Player Settings>Resolution and Presentation** submenu. Uncheck the **Optimized Frame Pacing** box.

In the options under **Orientation**, instead of **Auto Rotation**, you can select **Landscape Right** or **Landscape Left**. Here **Landscape Right** is selected.



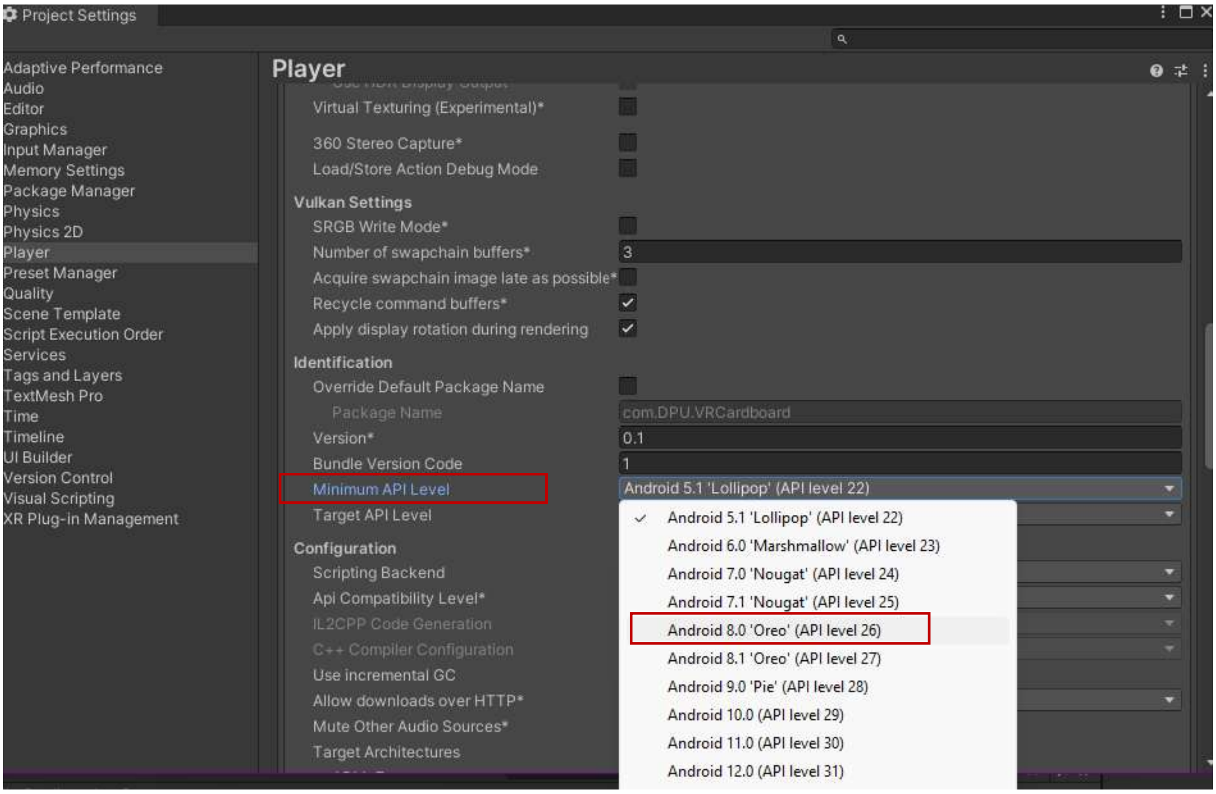
Now, go to the **Other Settings** section related to the **Player**. Select **ETC2** as the **Texture compression format**.



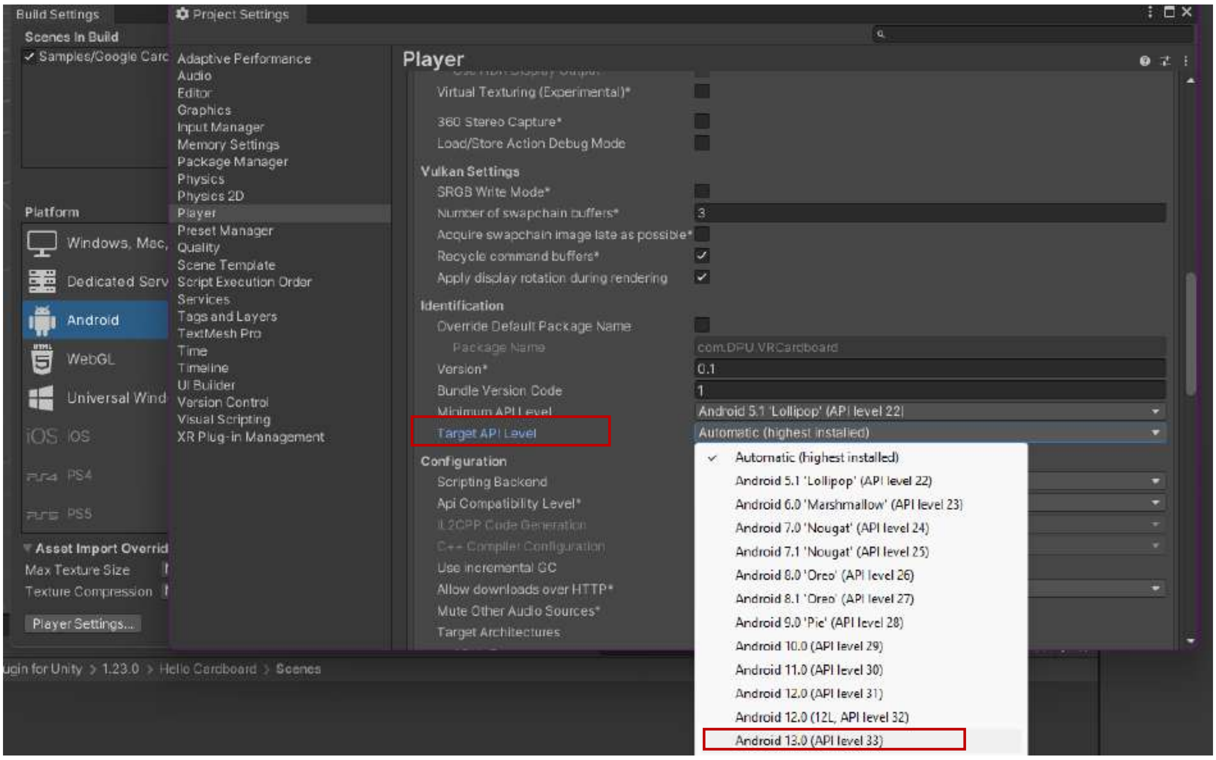
Vulkan Settings> Apply display rotation during rendering box to disable it.

Check if the company name is the same as above.

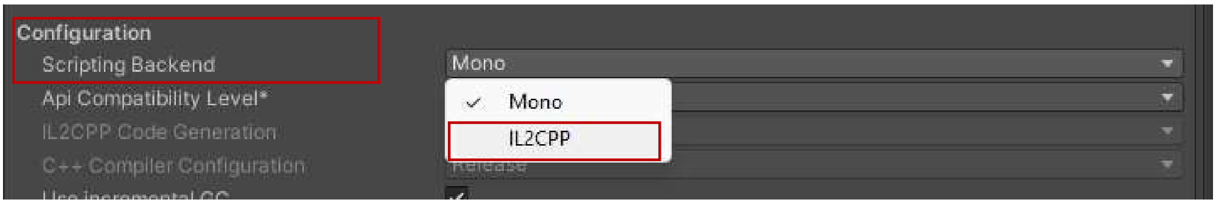
Set the **Android** level as **Minimum API Level>Android 8.0 'Oreo' (API Level 26)**.



Select **Android 13.0 (API Level 33)** for **Target API Level**.

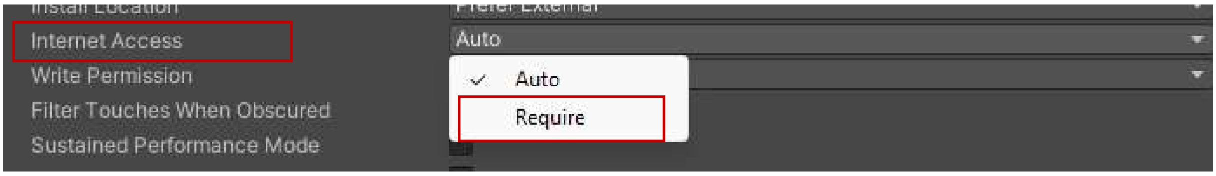


On the same page, set the **Configuration>Scripting Backend** selection to **IL2CPP**.



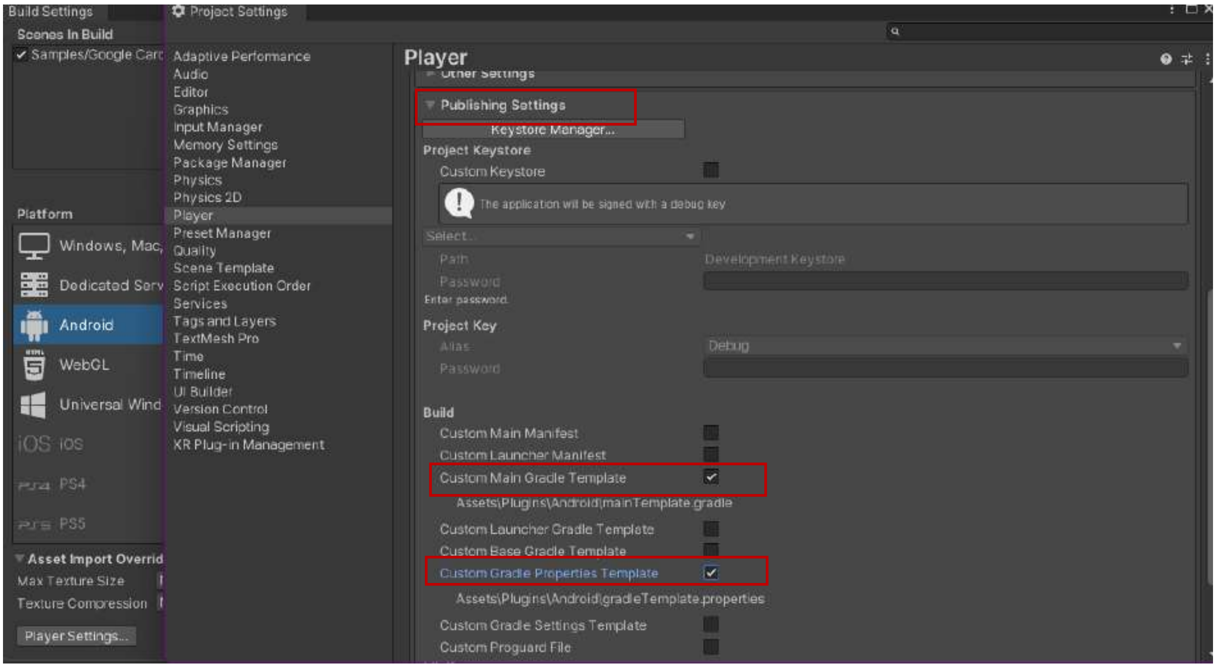
Click the **ARM64** box in the **Target Architectures** section.

Set the **Internet Access** section to **Require**.

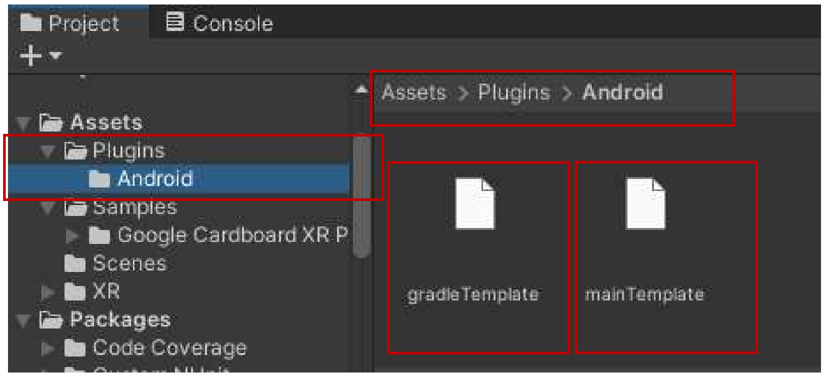


Note: If the Unity version is 2023.1 or later, select Activity in the Application Entry Point and deselect GameActivity.

Open the **Player>Publishing Settings** window. Here, let's select the **Custom Main Gradle Template** and **Custom Gradle Properties Template** boxes.



This last operation caused the **Plugins** subheading to open in **Project>Assets**. When we look here, we see that two files named **gradleTemplate** and **mainTemplate** were added under the **Android** folder.

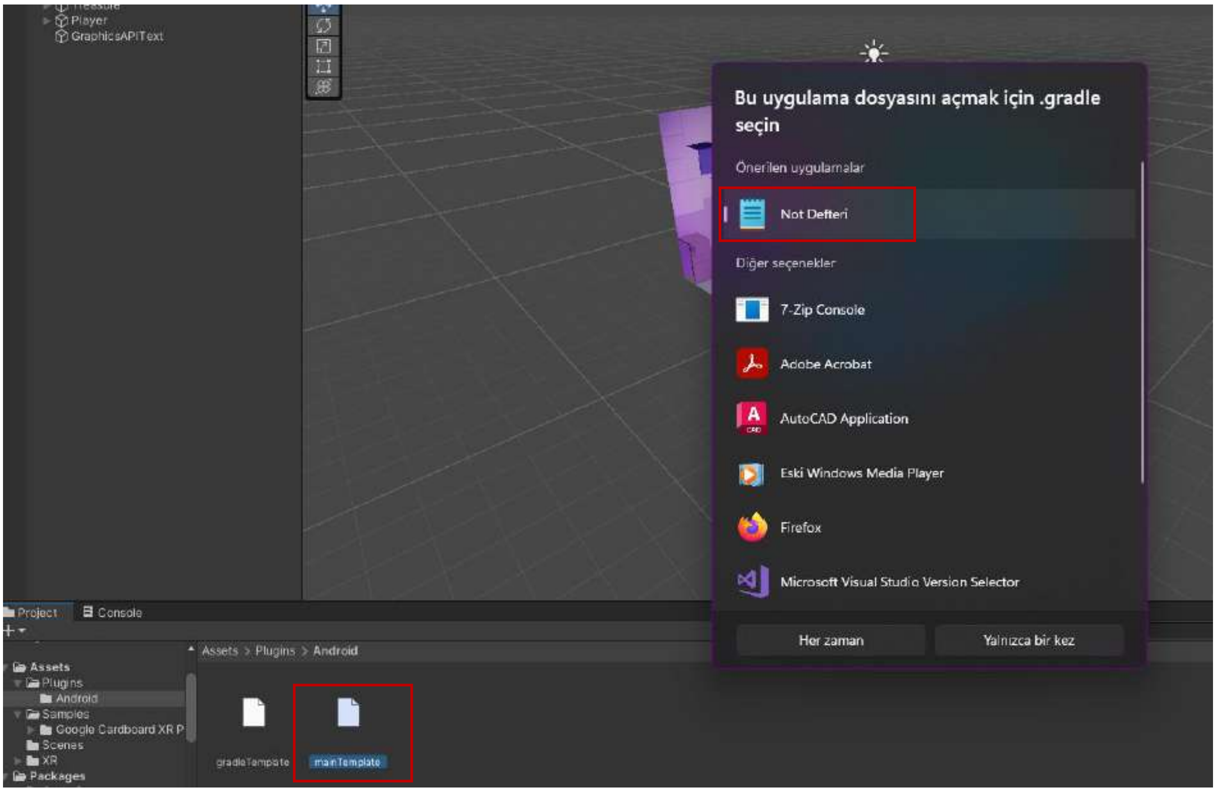


Now, an addition needs to be made to these files. The text of these two code snippets, which can also be copied from the page https://developers.google.com/cardboard/develop/unity/quickstart?hl=en and how to add them are listed below.

First code block of four lines:

```
implementation 'androidx.appcompat:appcompat:1.4.2'
implementation 'com.google.android.gms:play-services-vision:20.1.3'
implementation 'com.google.android.material:material:1.6.1'
implementation 'com.google.protobuf:protobuf-javalite:3.19.4'
```

Let's copy it from here. Let's double-click on the **mainTemplate.gradle** file to open it. We will be asked to choose an editor. **Notepad** is sufficient for this job.



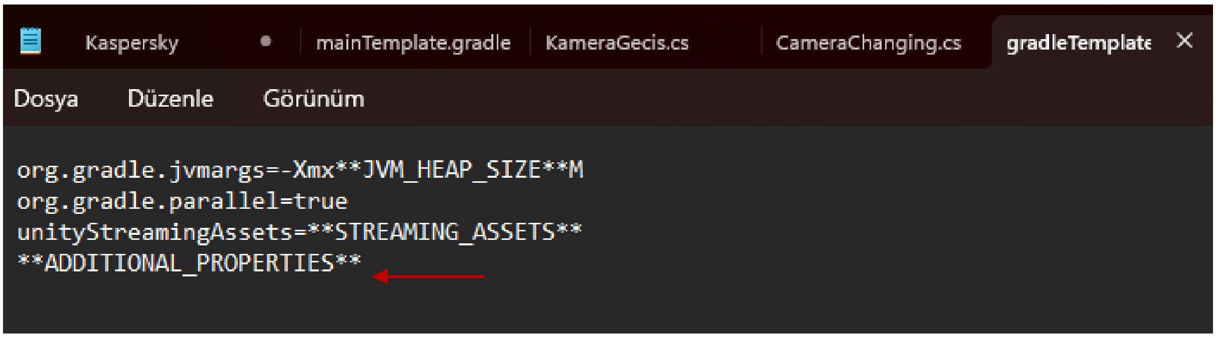
Once the file is opened, we need to determine where to copy the copied code block.

```
apply plugin: 'com.android.library'
**APPLY_PLUGINS**

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
**DEPS**}

android {
    ndkPath "**NDKPATH**"

    compileSdkVersion **APIVERSION**
    buildToolsVersion '**BUILDTOOLS**'

    compileOptions {
        sourceCompatibility JavaVersion.VERSION_11
        targetCompatibility JavaVersion.VERSION_11
    }
```

Here, paste the codes to the place indicated by the arrow above the ****DEPS**}** expression and save it.

```
apply plugin: 'com.android.library'
**APPLY_PLUGINS**

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
  implementation 'androidx.appcompat:appcompat:1.4.2'
  implementation 'com.google.android.gms:play-services-vision:20.1.3'
  implementation 'com.google.android.material:material:1.6.1'
  implementation 'com.google.protobuf:protobuf-javalite:3.19.4'

**DEPS**}
```

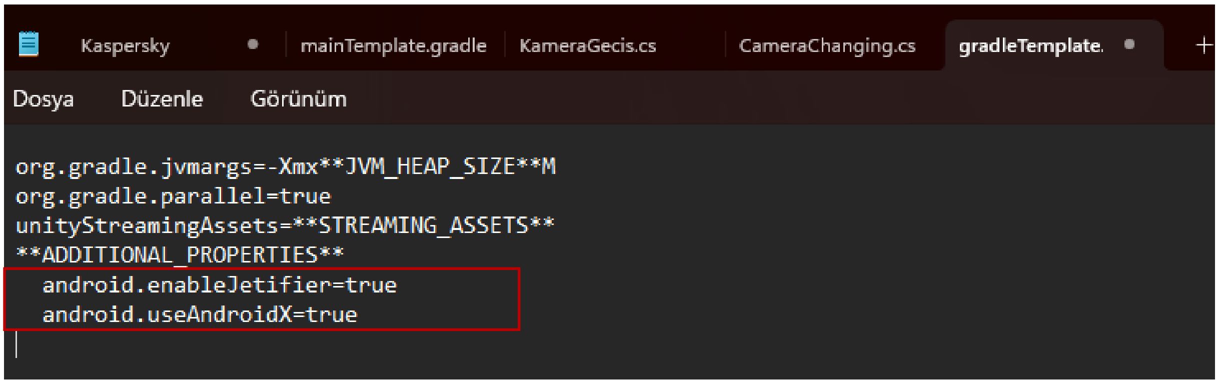
Open the second file, i.e. **gradleTemplate.gradle**, in the same way, for example, with **Notepad**. As you can see, this file has relatively short content.



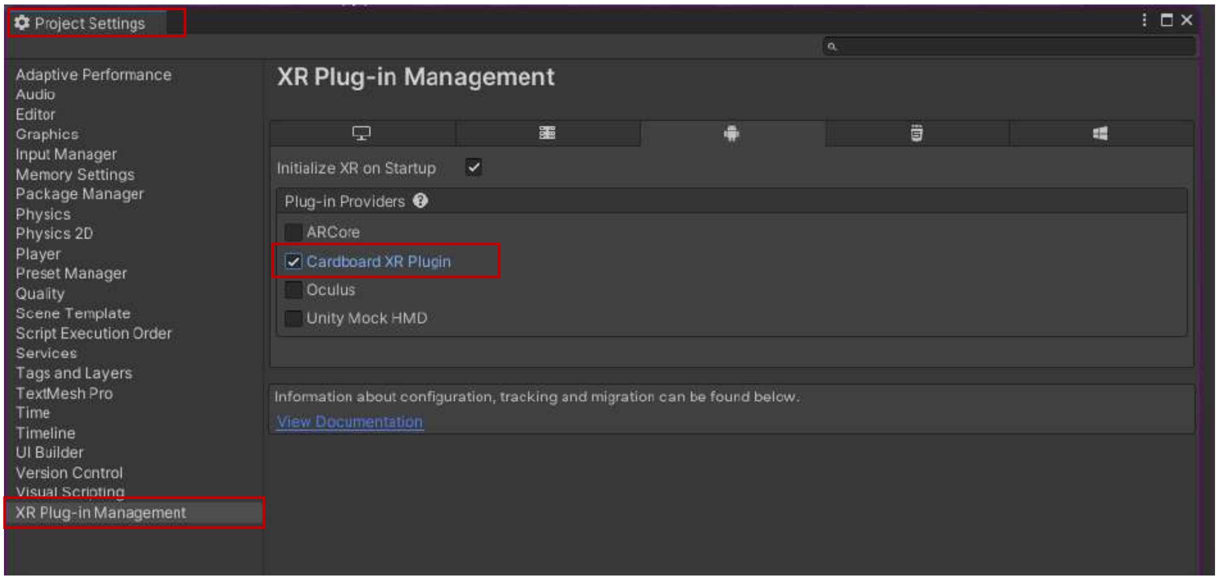
Copy the two lines of code given below.

```
android.enableJetifier=true
android.useAndroidX=true
```

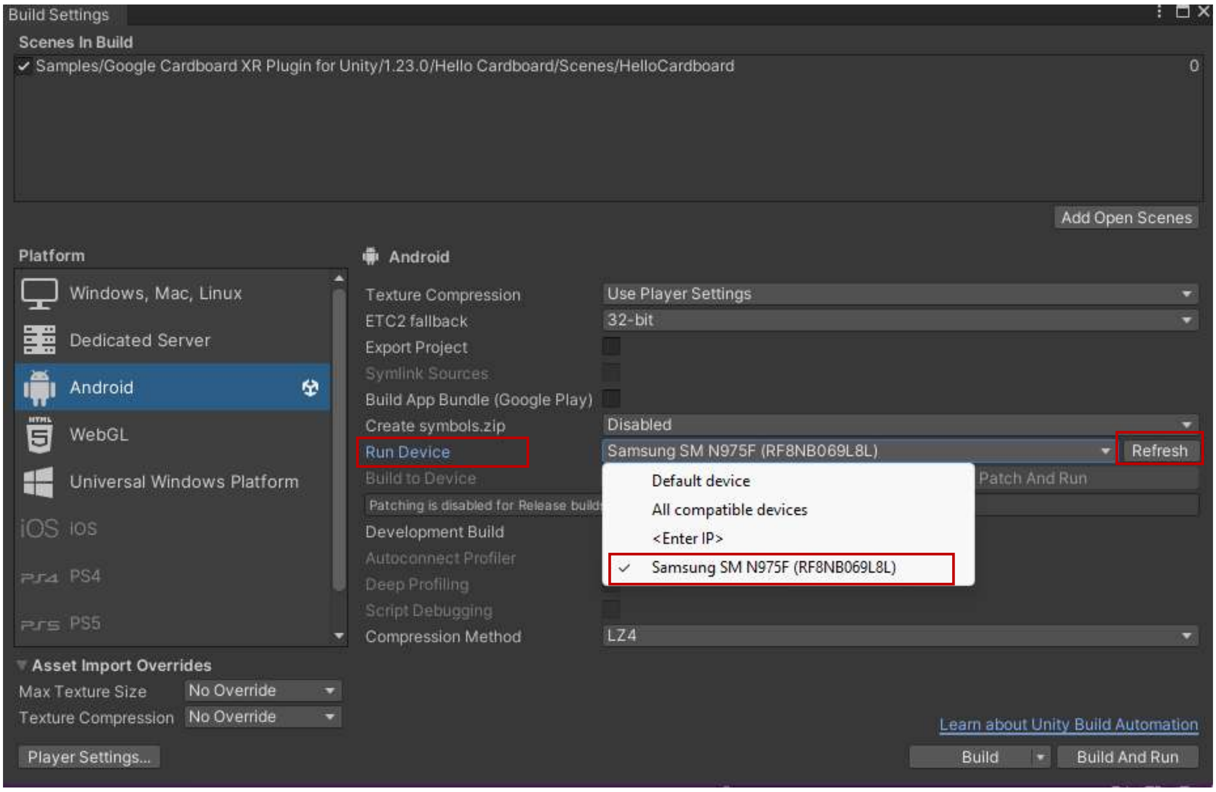
Paste it to the location shown in the file and save it.



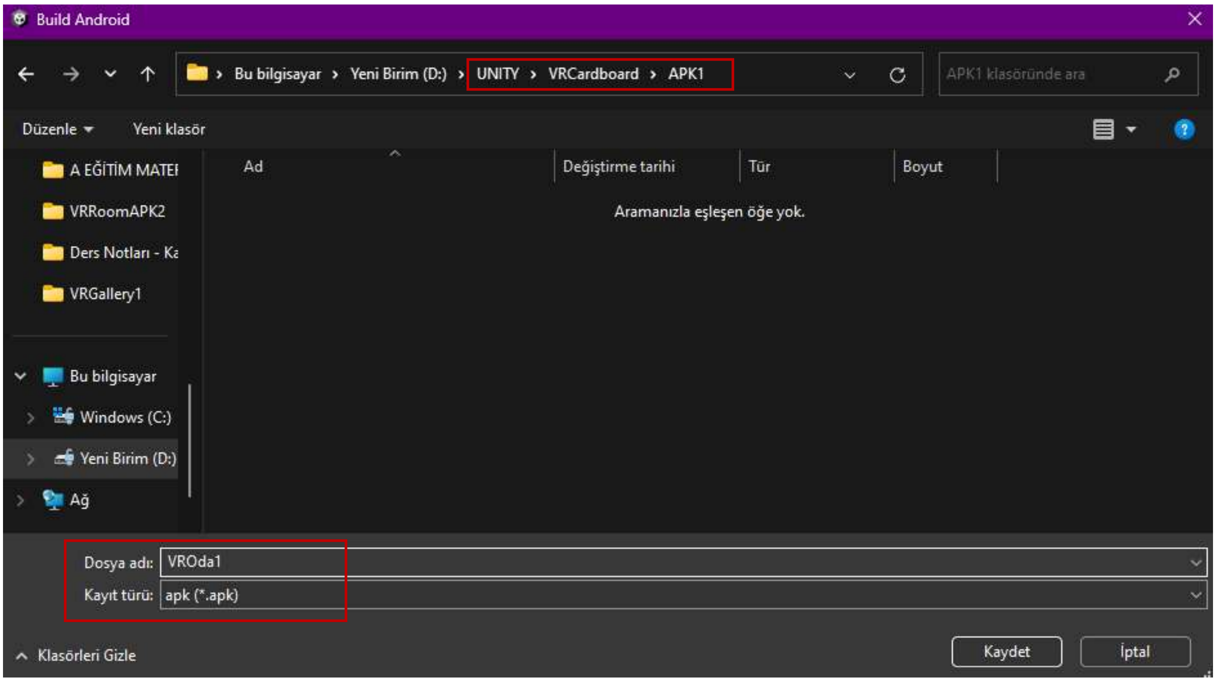
Go to **Project Settings>XR Plug-in Management** and select the **Cardboard XR Plugin** box.



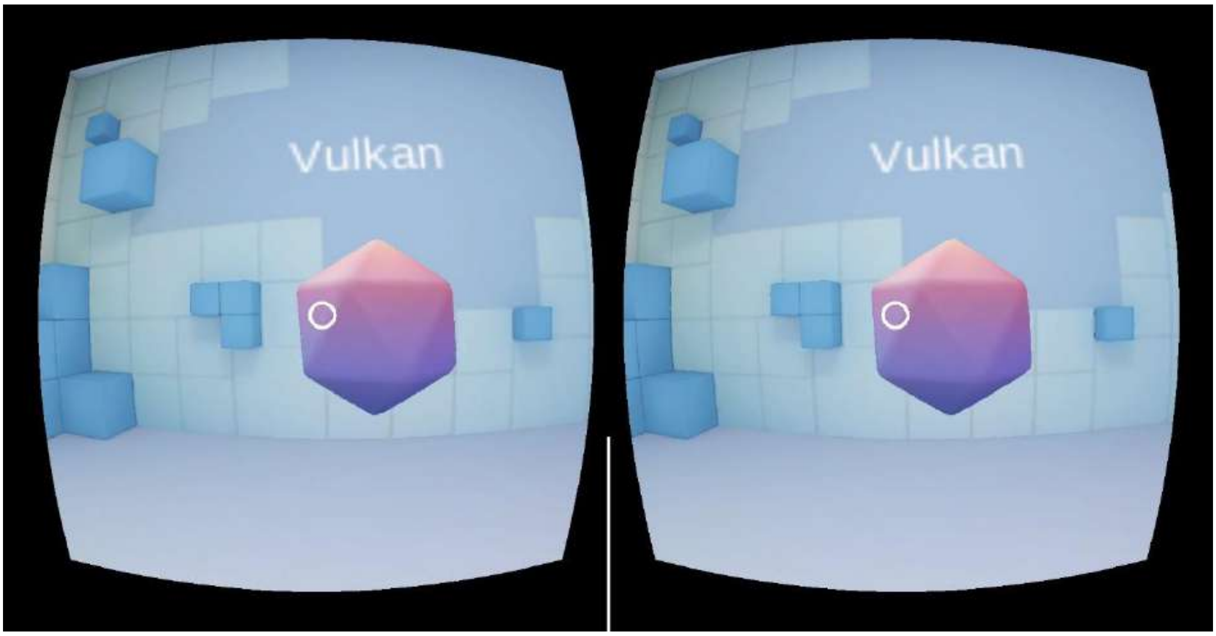
Come to **Build Settings**. Connect our smart mobile device with developer features enabled to the computer. Here, **Refresh** the **Run Device** section and select the **mobile device**.



In the window that opens with the **Build and Run** key, define an **APK** file name (**VROda1**) by opening a new folder (here **APK1**) or directly in the current location.



After installing the project as an **APK** file on the phone, the application with two stereoscopically symmetrical images will start working.



A **3D room** experience can be achieved by attaching a **cardboard** headset. The headset moves, and the Treasure changes color with the **Reticler Pointer** and can be transported with the button under the headset.

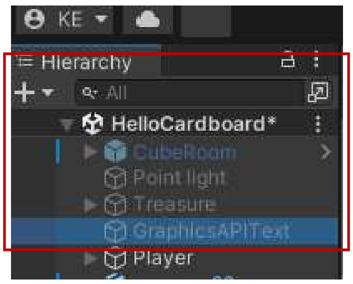
At this point, we can remove the room and objects on the stage and position a 3D environment of our design on the stage. In this way, we can achieve 3D experience for many designs.

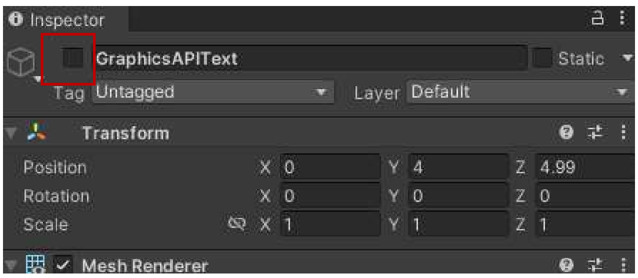
There are tutorials on the internet that show what happens when the aimer hovers over an object. Actions that will happen when the object is in the **On Click** state can be done with **C# coding**.

10.3.Moving in a VR Scene

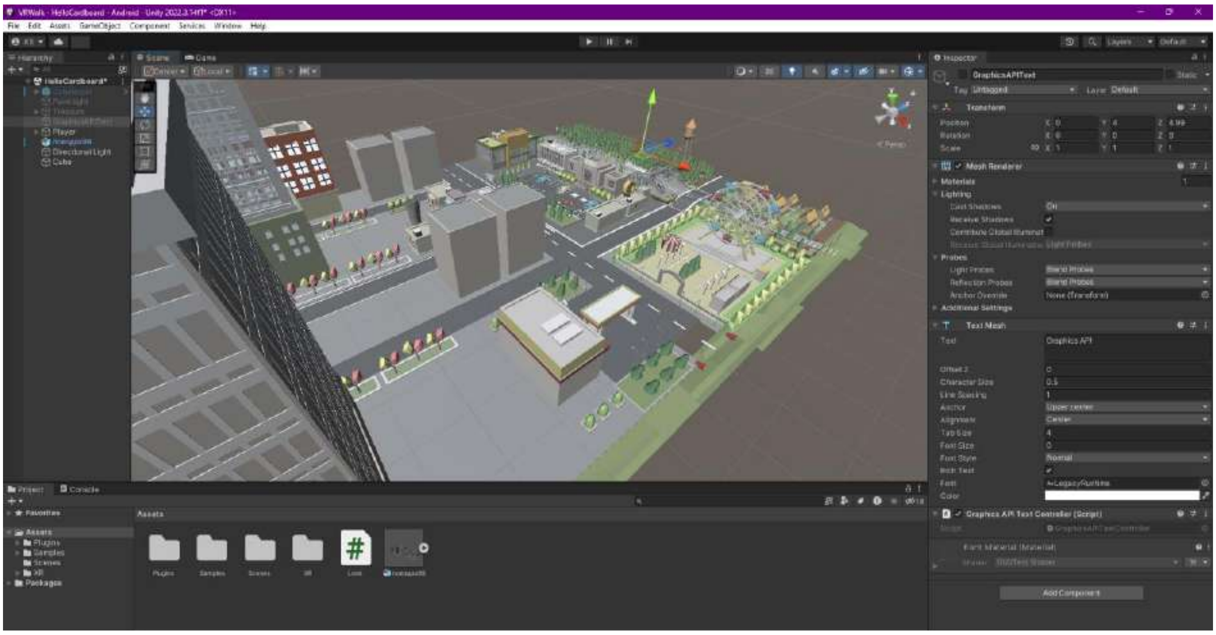
Cardboard devices, except **Google Daydream**, usually do not have a control device. Therefore, the application is passive in a space and is done by looking around. So, is there no way to move around in the scene? Yes, there is! This section explains how this can be done with **C# Script** codes.

We have achieved 3D experience with the **VR Room APK** file and a cardboard-type headset. Now, first deactivate the ready scene, namely the **Cube Room, GraphicsAPIText, Point Light** and **Treasure** objects, by deselecting them in the **Inspector**.

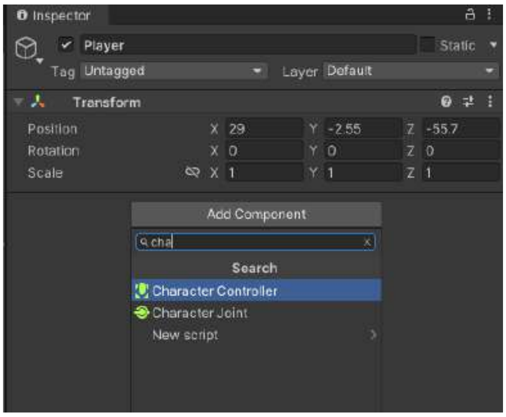




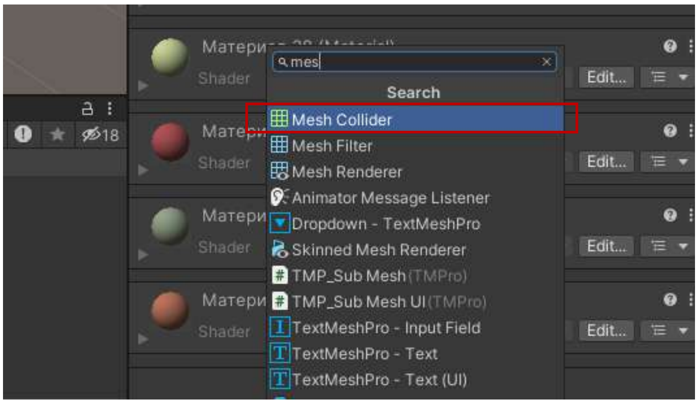
In the **Asset** section, download the **Sketchfab>Low Poly City, fbx** file that we used before and drag it to the stage.



Position the **Player** object in the **Hierarchy** in the city. Add the **Character Controller** component to the **Player** with **Add Component** in the **Inspector**.



To prevent the player from passing through the city and falling under it, select the city object and add a **Mesh Collider** with an **Add Component** in the **Inspector**.

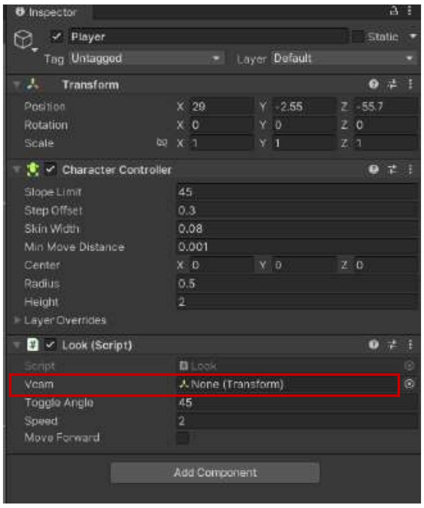


Also, create the **C# Script** file named **Look.cs**, whose codes are given below, in the **Assets** section and write the following script.

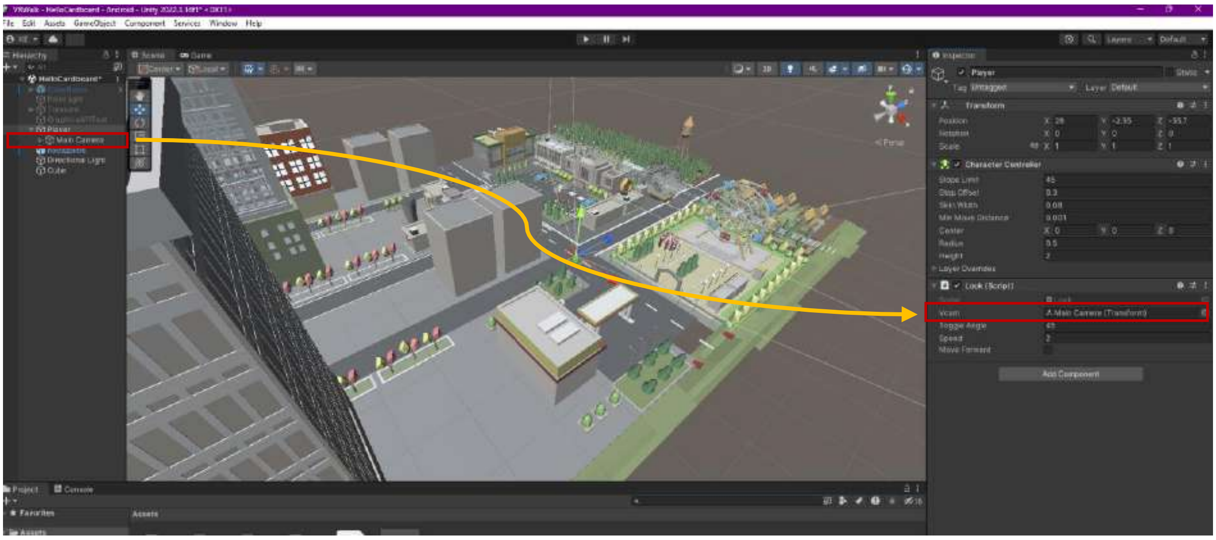
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Look : MonoBehaviour
{
    public Transform vcam;
    public float toggleAngle=45.0f;
    public float speed = 2.0f;
    public bool moveForward;

    private CharacterController cc;

    // Start is called before the first frame update
    void Start()
    {
        cc = GetComponent<CharacterController>();
        //DontDestroyOnLoad(this);

    }

    // Update is called once per frame
    void Update()
    {
        if (vcam.eulerAngles.x <= toggleAngle && vcam.eulerAngles.x < 90.0f)
        {
            moveForward = true;
        }
        else
        {
            moveForward = false;
        }
            if (moveForward)
            {
                Vector3 forward = vcam.TransformDirection(Vector3.forward);
                cc.SimpleMove(forward * speed);
            }

        }
    }
```

The code file ensures that the **Player** stands still when the camera is viewed 45 degrees up and down and moves forward at angles between them (*toggleAngle=45.0f*).

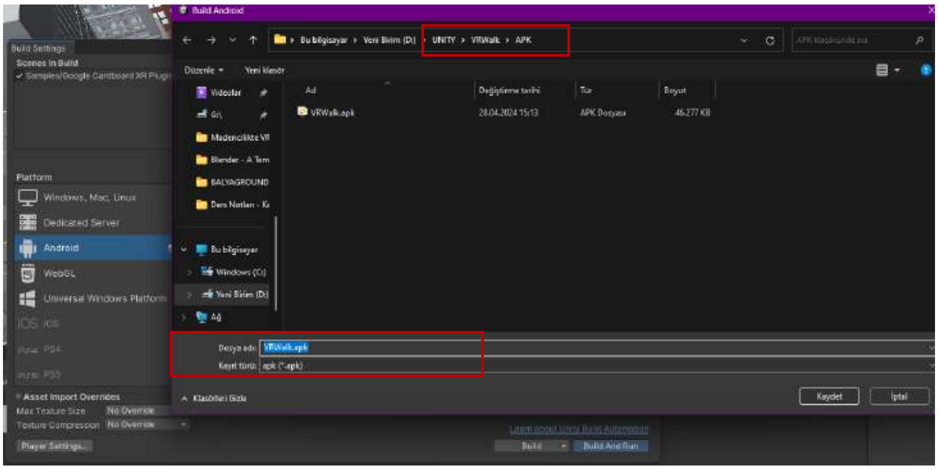
In the **Inspector**, when you look at **Look (script)**, you will see that the **Vcam** section is empty and says **None (Transform)**.



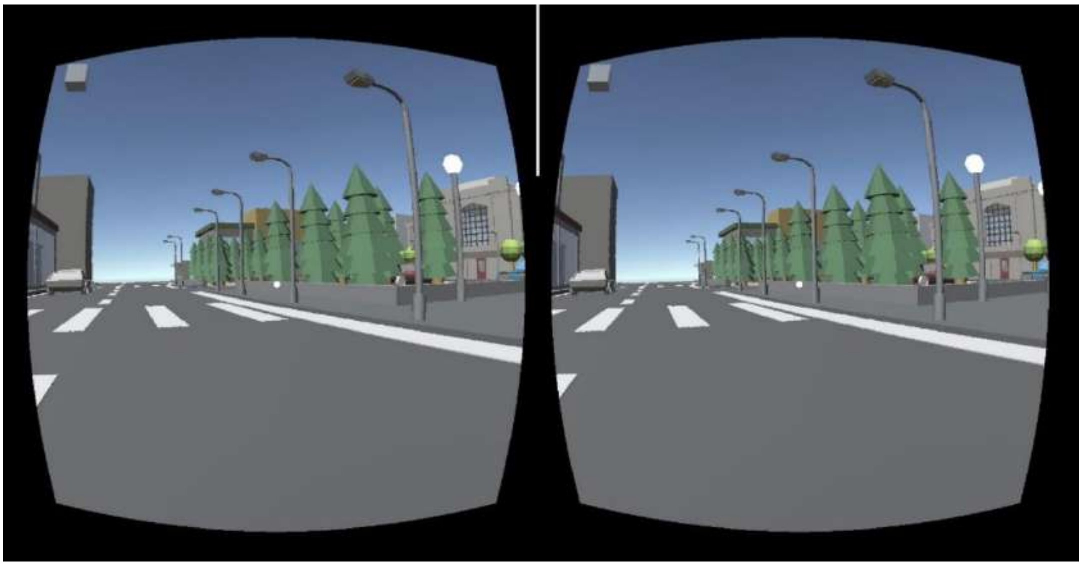
Drag and match the **Main Camera** here, which is located under the **Player** in the **Hierarchy**.



Go to **Build Settings**. While our phone is connected, create our **APK** file by pressing **Build and Run**.



We can open the application on our phone and do the first check. We can test the motion control by turning the mobile device in all directions.



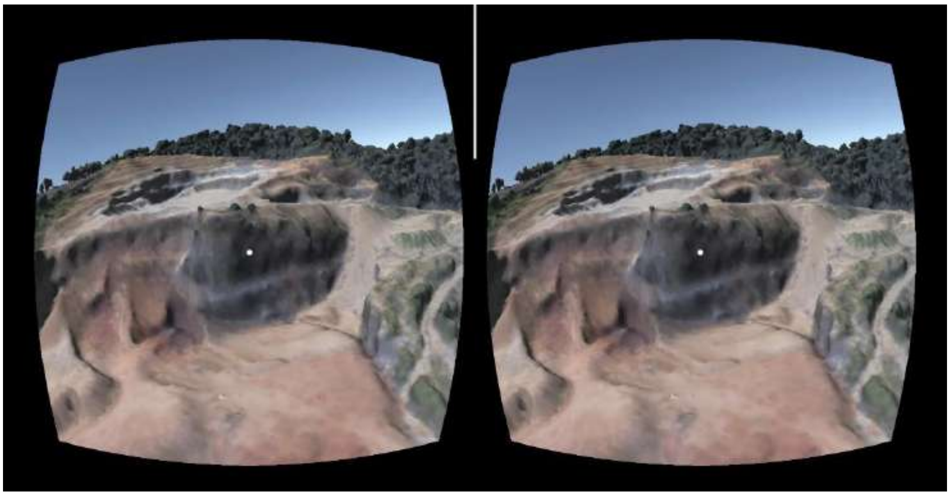
As a result, we can observe the result in 3D by placing our phone on our cardboard headset. We can control it by moving our head left and right and up and down and experiencing the VR application.

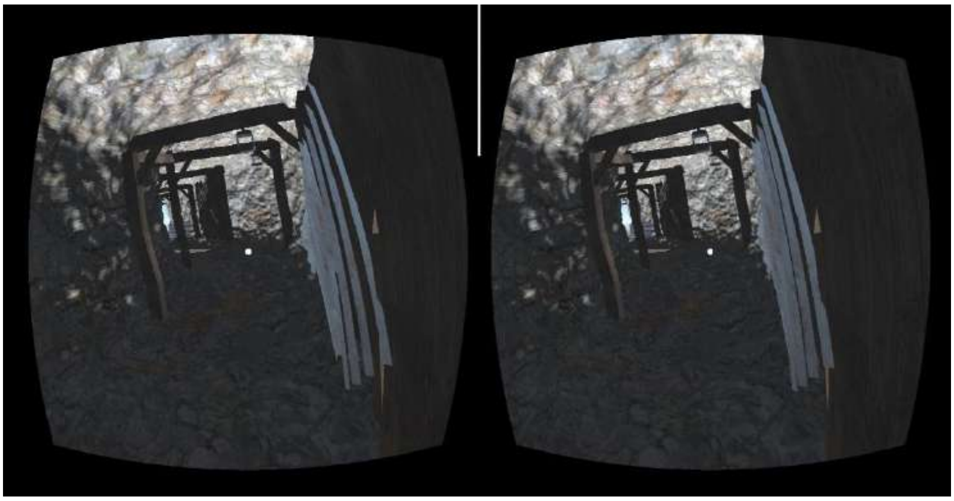
Some phones may require the **Cardboard Google LLC** app to be installed from **Google Play** for this app to work.

On Google Play, with keywords like **VR Player** and **VR Converter**, we can find applications that convert our mp4 etc. files to VR (for example **VR Video Converter** and **VR Game**).

There are also many videos in VR standards on YouTube. For example, a search with keywords such as VR Video can find many videos suitable for VR viewing and can be watched in 3D on a VR headset.

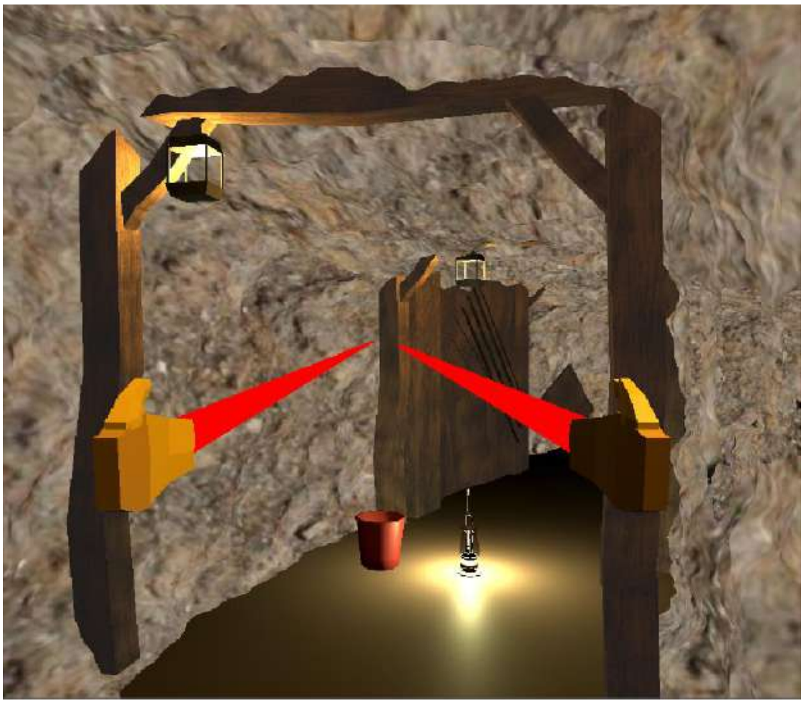
Similarly, for open pit and underground mine gallery scenes, it is possible to deploy on mobile device and visualize in Google Cardboard headset.





We can walk through and control movements in the scenes by using the C# codes.

Although the Meta Oculus Quest 2/3 tutorial is not taken here, some outputs are given to show the ability to have VR deployments for mining cases. Here, Meta and Unity MRTK (Mixed Reality Toolkit) templates are utilized.



11. AUGMENTED REALITY – AR APPLICATIONS

11.1.Augmented Reality – AR

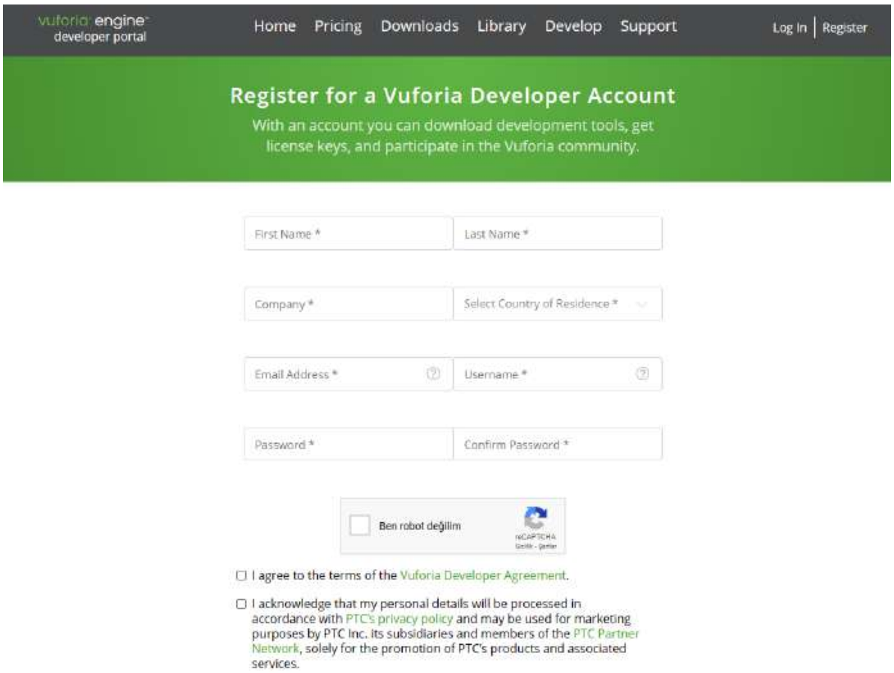
Augmented reality can be thought of as the merging of virtual assets with the real world. If there is the possibility of interactive management and intervention with these assets, the term **Mixed Reality** is used for this concept.

For augmented reality, special glasses called **Smart-Glass**, which are usually expensive, are used. However, smartphones or tablets that everyone can have can also be used for this purpose, although they do not provide the same experience.

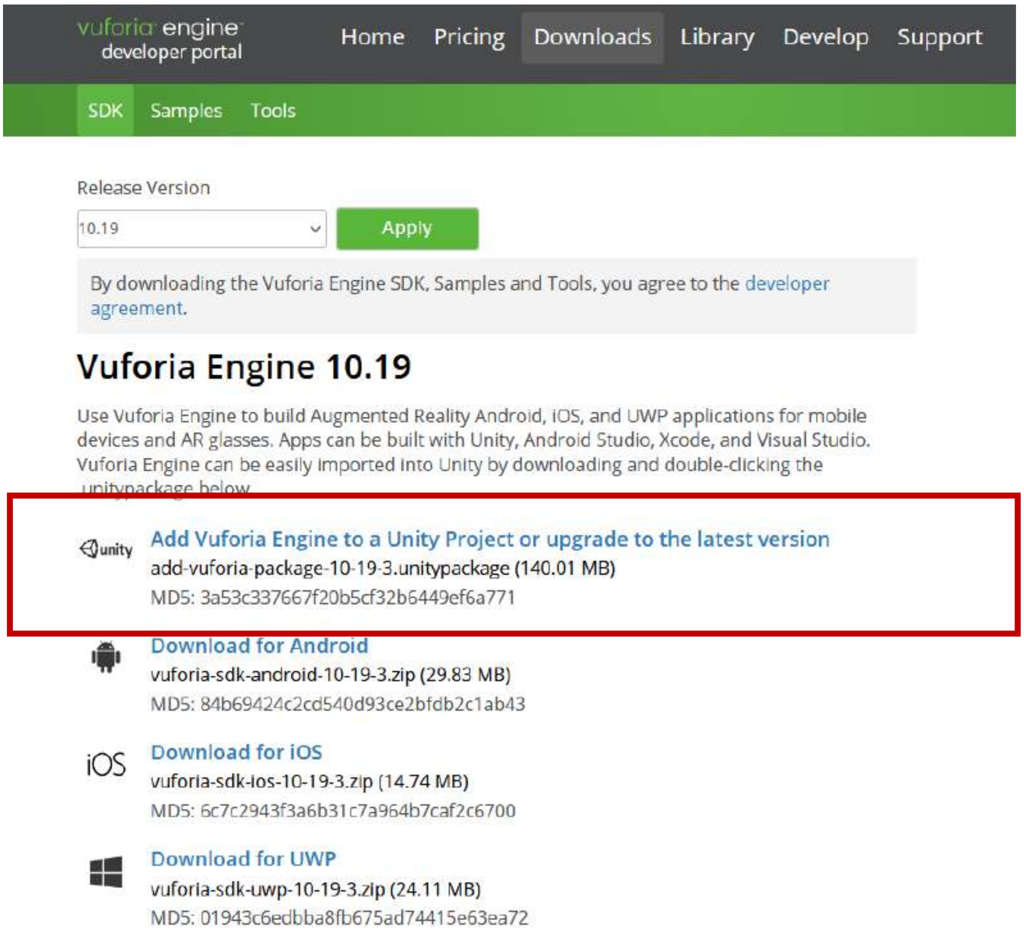
Various engines can be used to develop AR applications with Unity. Examples include **AR Foundation**, **AR Core, AR Kit**, and **Vuforia**. Due to its practical use and reason, this section will show the **Vuforia** application.

11.2.Vuforia AR Engine

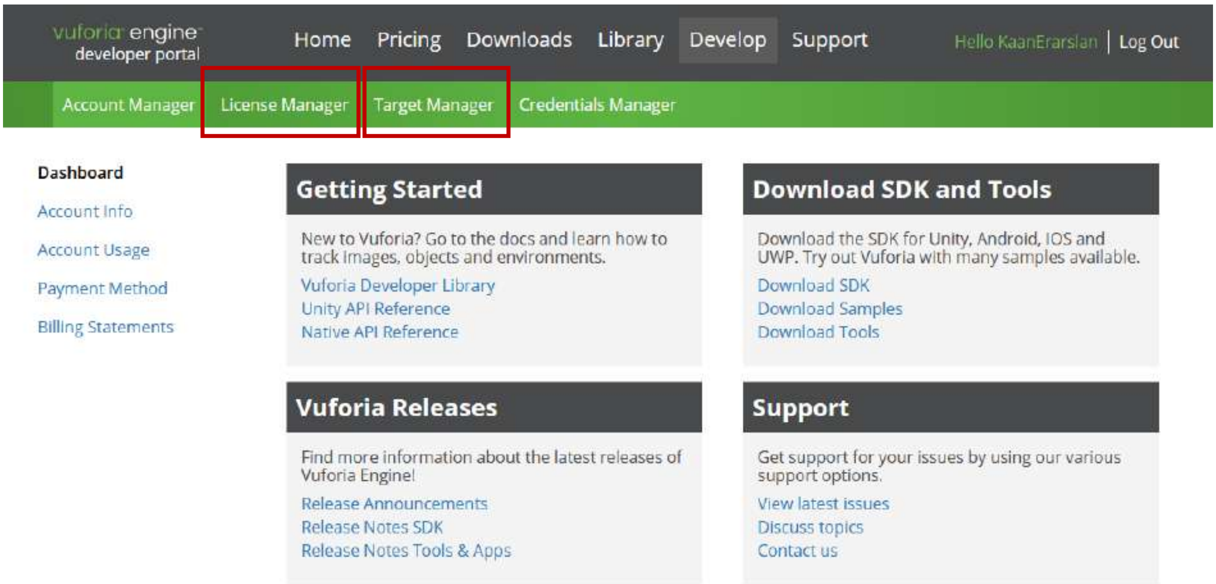
This software has **Asset** downloads that work integrated with the **Unity Asset Store**. For this purpose, an account must first be opened on the **Vuforia** website. Using the same email address as Unity will be advantageous.



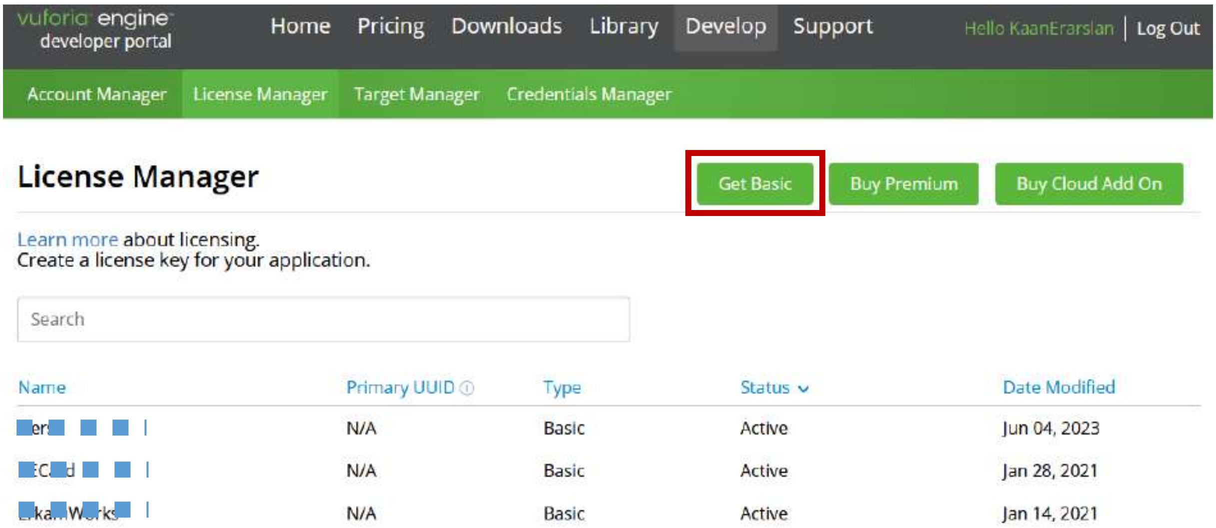
Vuforia's Unity app is located under the **Downloads** menu, but it is not required for this project.



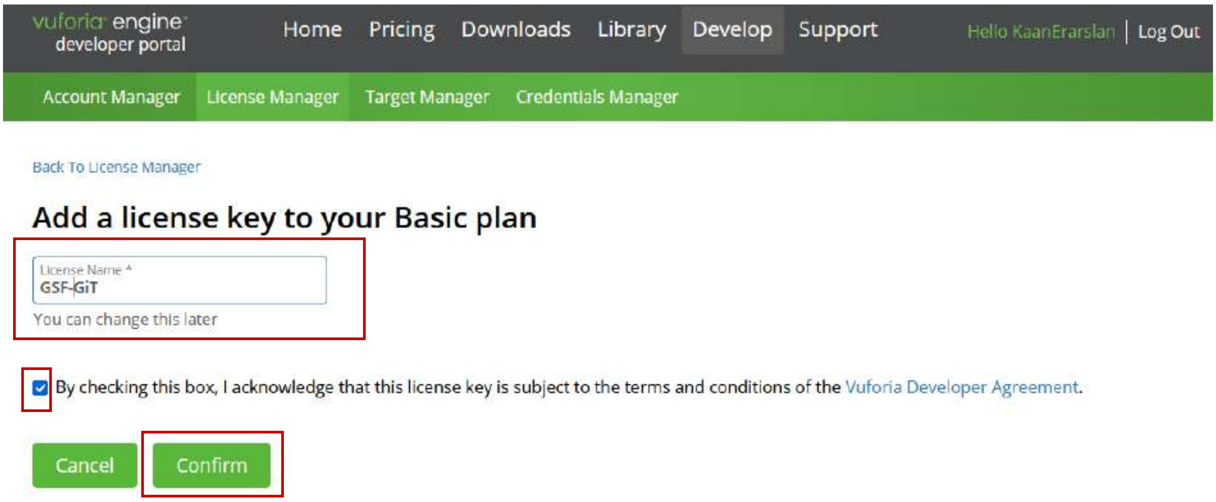
Here it will be necessary to use the **License Manager** and **Target Manager** titles for preliminary preparation.



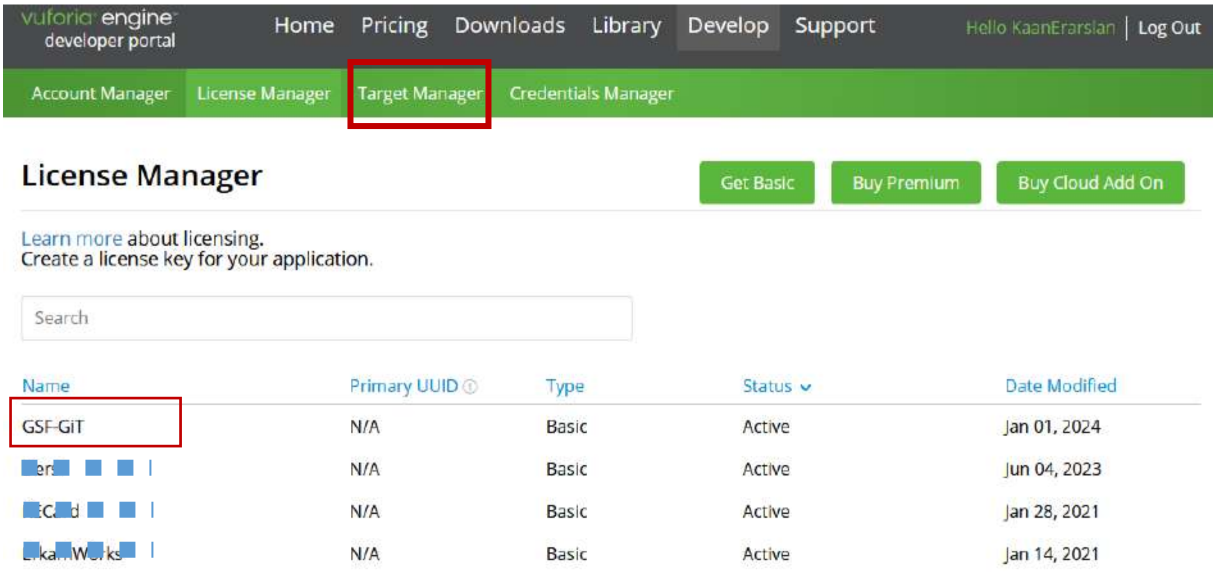
First, let's select this menu and use the **Get Basic** button in the window that appears to create a **Database License** with the **License Manager**.



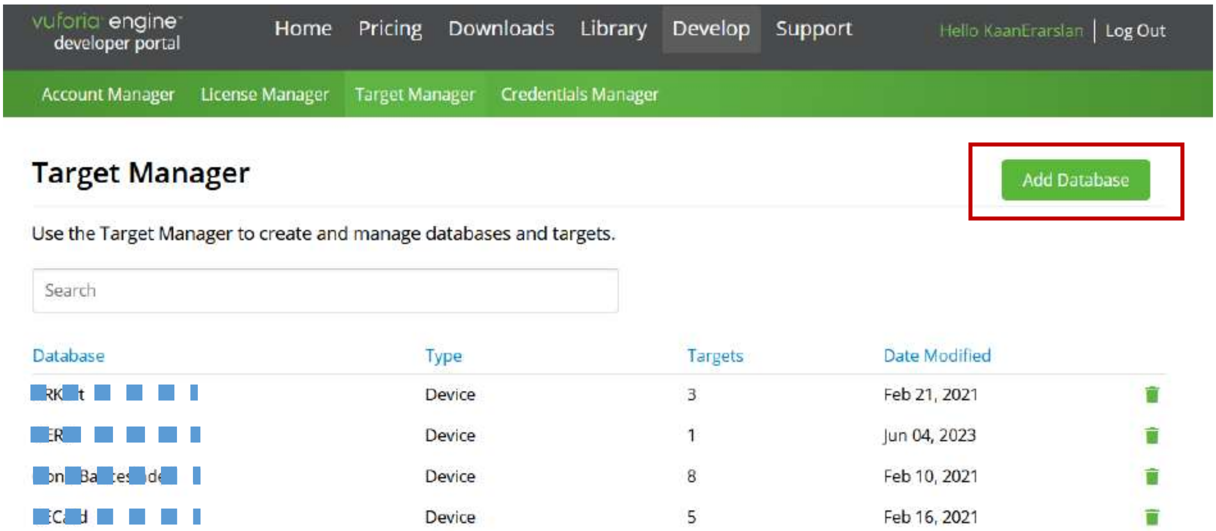
In this window, specify the **License name** and check the agreement box. Continue by clicking **Confirm**.



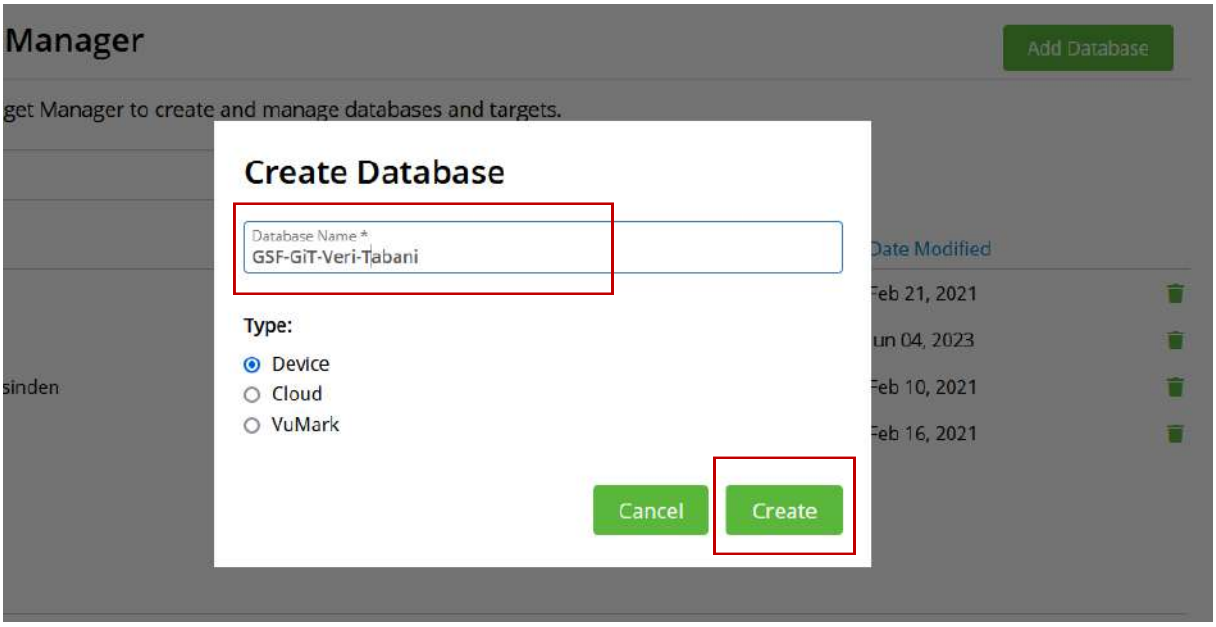
Now, a **license** has been opened with the name we created on our list.



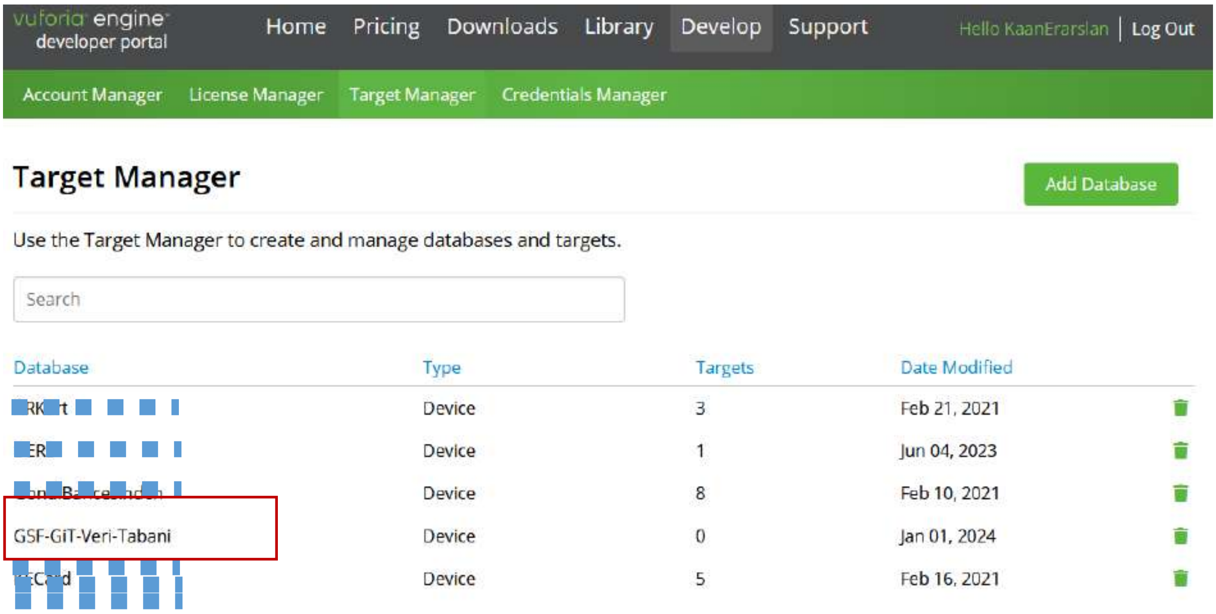
Next step, we can go under the **Target Manager**. Here, click **Add Database** to create a database.



Enter the name of the database; here, do not use special characters (such as some Turkish characters) and spaces like in web page names. **Device** option is suitable as **Type**. After naming, create our database with the **Create** button.



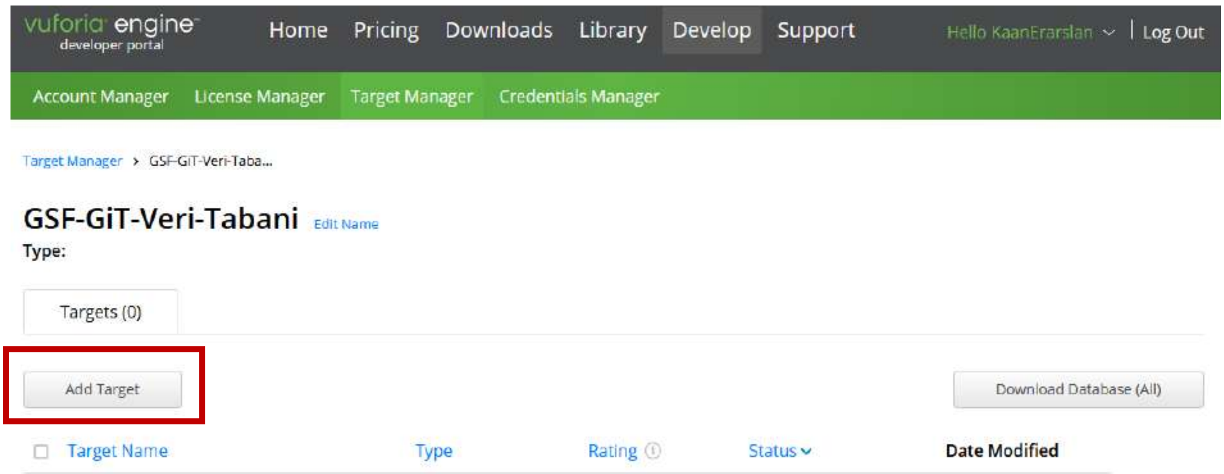
Our database will be visible in the list.



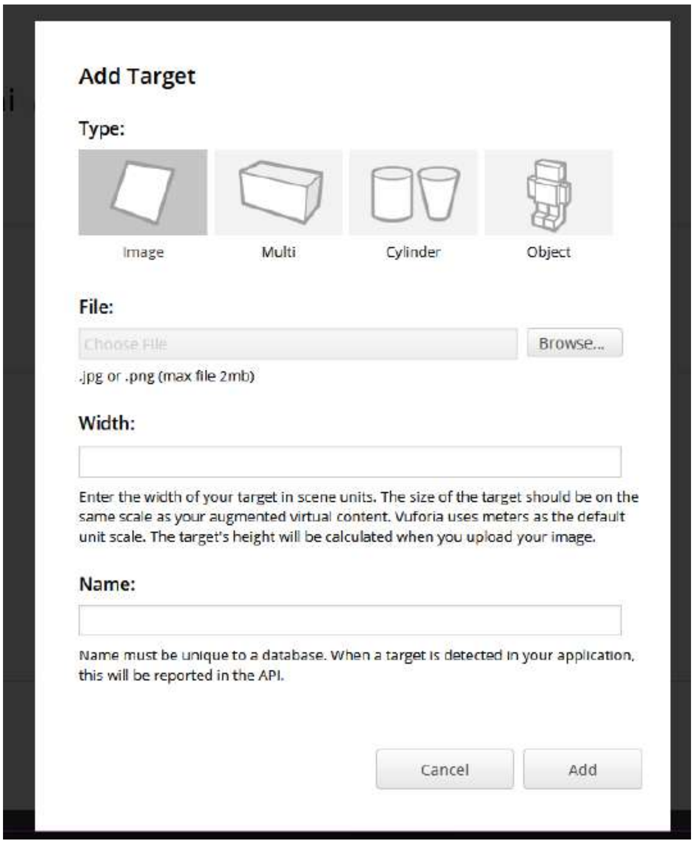
Now, prepare to add an object to the database that opens empty, or you can also use a ready-made image file.

Here, we set our target image as the **GSF Faculty** logo. Take the high-resolution file from the faculty page to the hard disk. File name: **GSF-LOGO.png**

To add this image to our database in Vuforia, all we need to do is type the phrase "**GSF-GiT-Database**" that we just created in **Target Manager** and whose name we see in the list. The window that opens allows us to upload various visual objects to the database. This object can be a **JPEG** or **PNG** file. It is important to note that the quality of the visual object must be high. For **Image Target** type applications, **Add Target** is selected to upload the visual file we plan to upload.

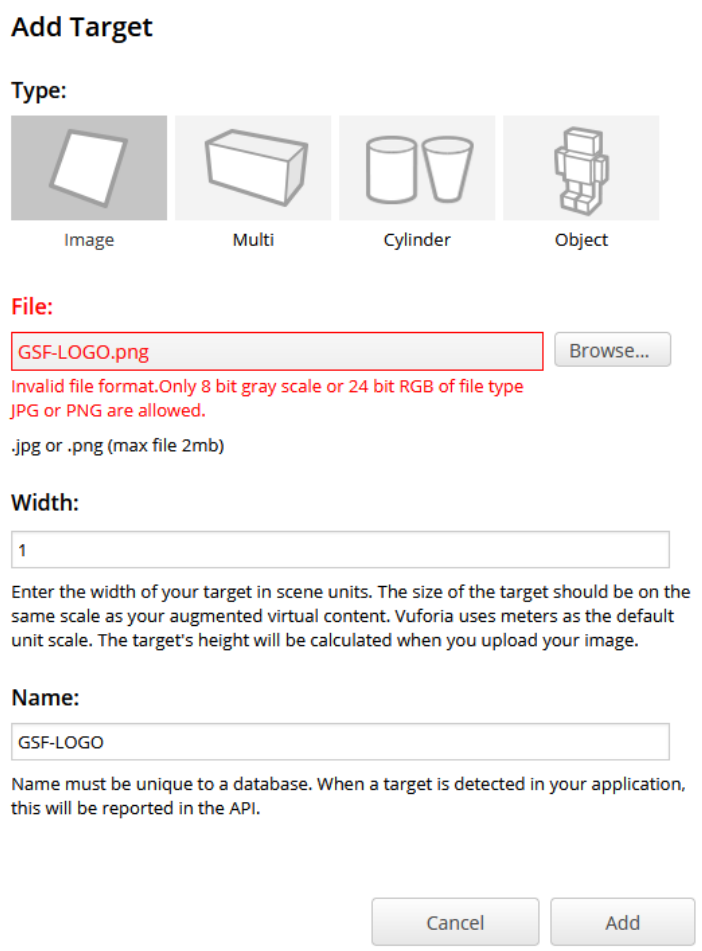


The window that opens offers options to upload files to the database.

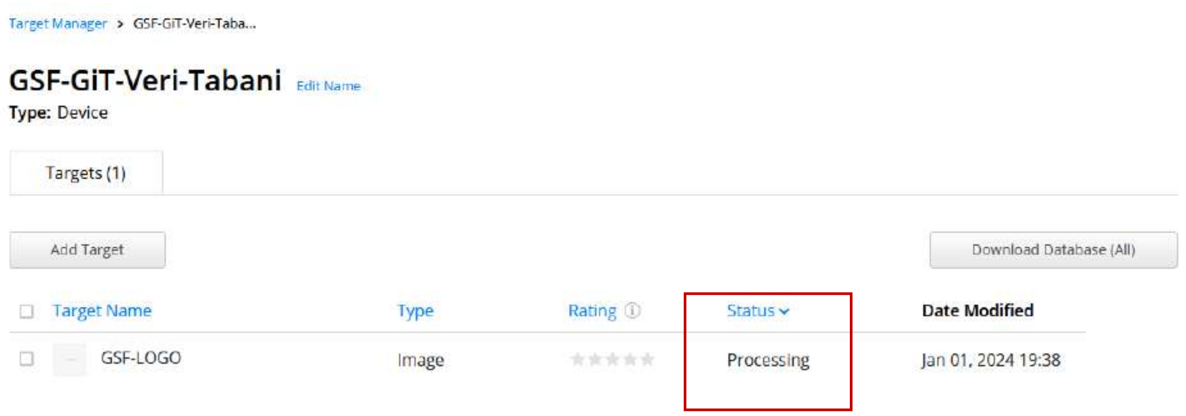


Now, upload our target file to the **File** section with **Browse**. The file name will be seen as **GSF-LOGO.png**. Write **1** to the **Width** section. **GSF-LOGO** automatically appears in the **Name**. Add it with the **Add button**.

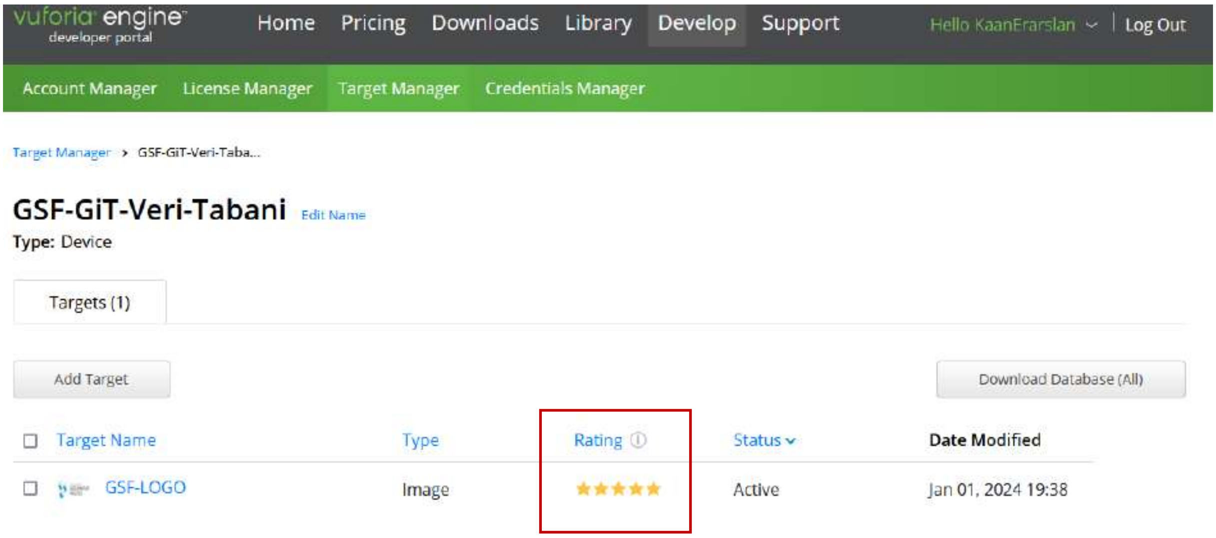
You may be asked to change the properties of the file:



To overcome the problem, the file was saved in **JPEG** format with **Paint** and accepted as a problem-free file. Our file appears in the database list that opens. At this stage, **Vuforia** performs the test that appears with the phrase "**Processing**" to determine the file **quality**.

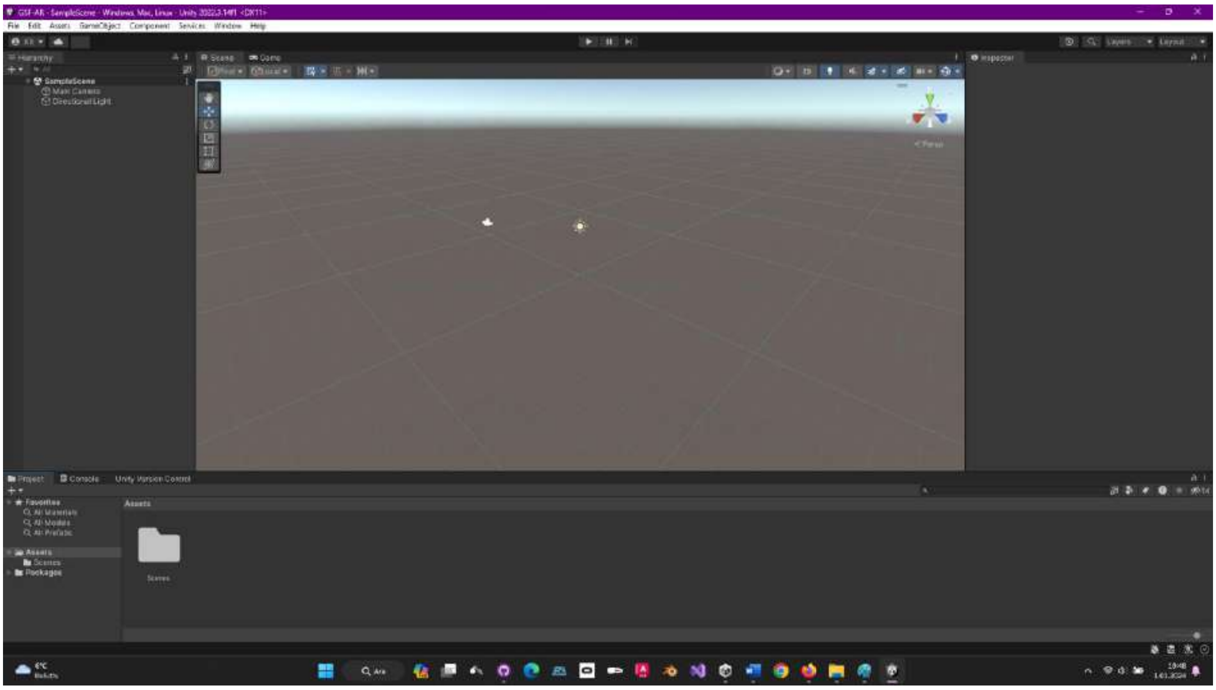


When we refresh the page after waiting for a short time, it displays the **quality level (Rating)** out of 5 stars. **5 stars** is the strongest considered file quality. The system can work up to **3 stars**, but it is likely to break in the mobile application.

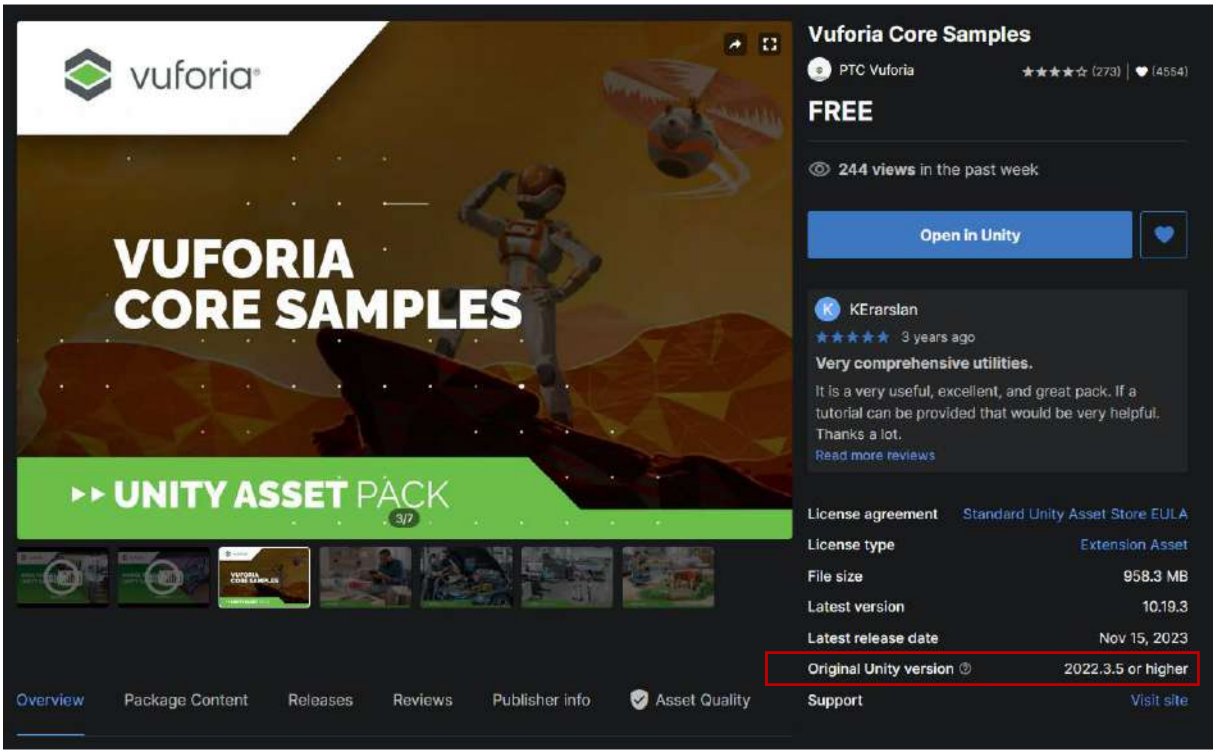


11.3.VUFORIA integration in UNITY project

Now, continue by opening a project in Unity with version **2022.3.5 or higher**. In this application, version 2022.3.14f is used.

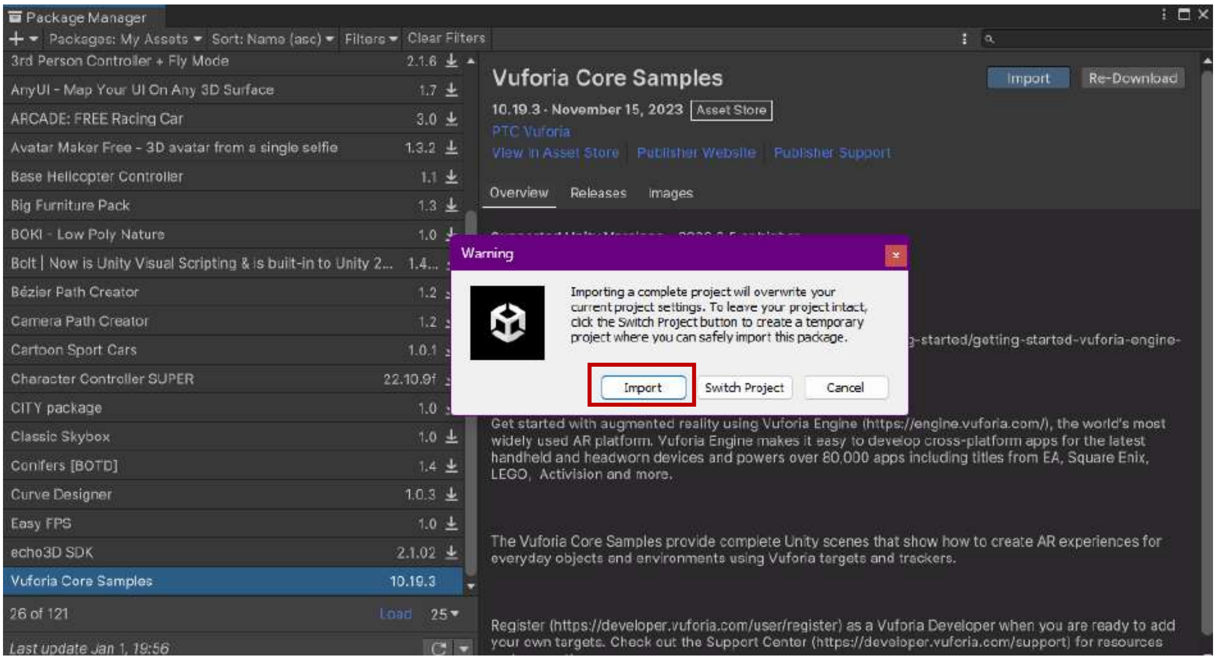


Log in to the **Unity Asset Store** with **Windows->Asset Store**. Reach the **PTC Vuforia Core Samples** section by typing **Vuforia Core Samples**. If you notice, Unity requires a version of **2022.3.5** or **higher**.

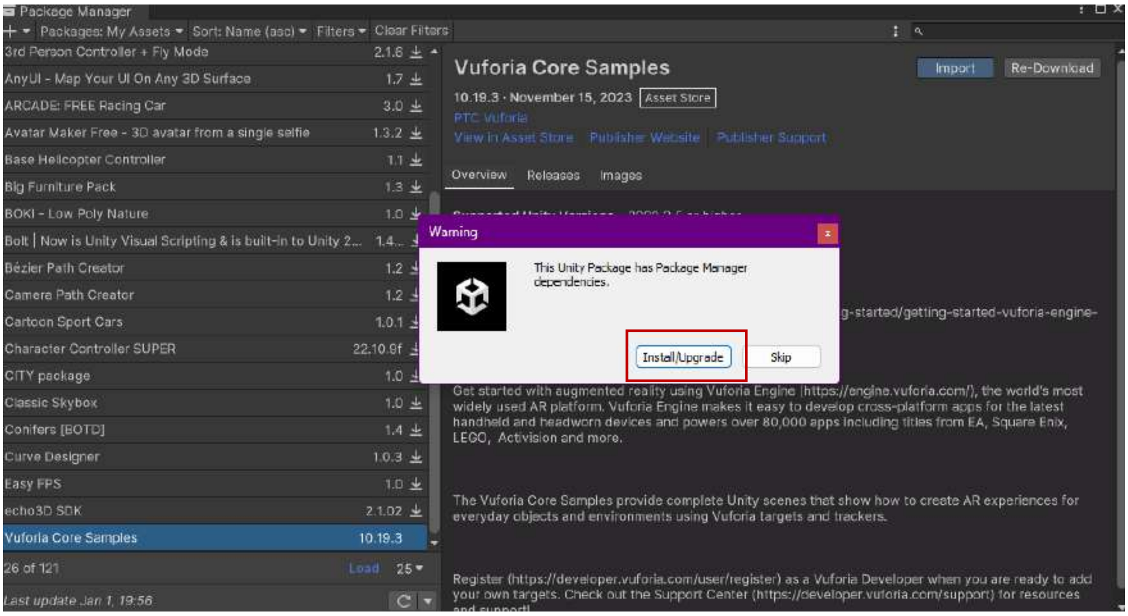


Now, add it to our assets ("**Add to Assets**") and open it in our project ("**Open in Unity**"). If we are using it for the first time, perform the **Install** process. Then **Import**.

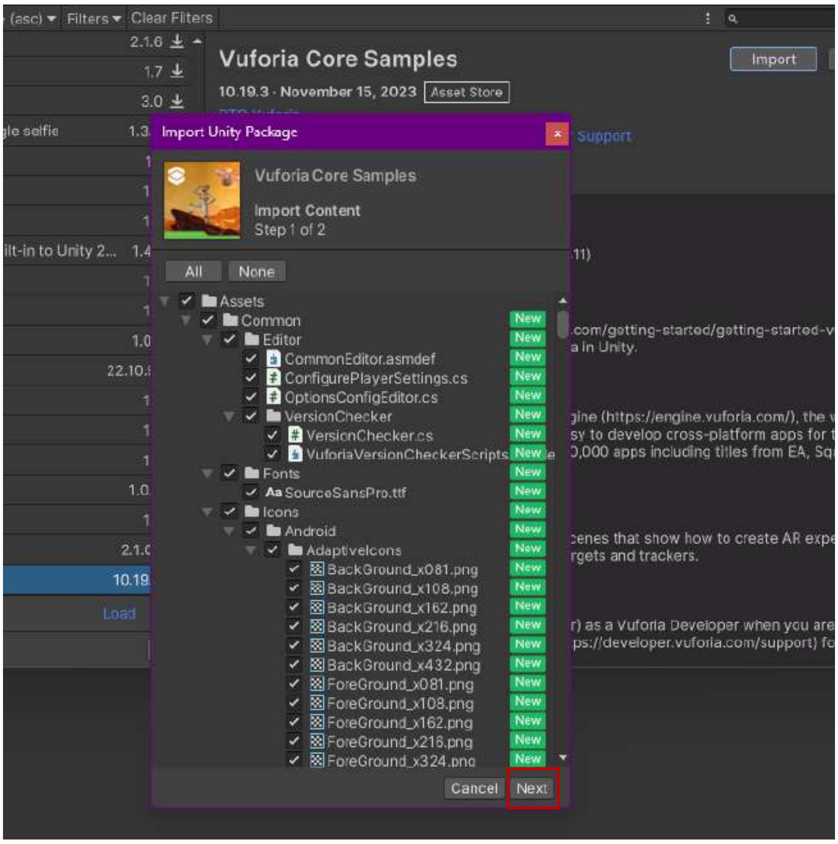
Add it to our assets ("**Add to Assets**") and open it in our project ("**Open in Unity**"). If we are using it for the first time, go on the **Install** process. Then, **import**.



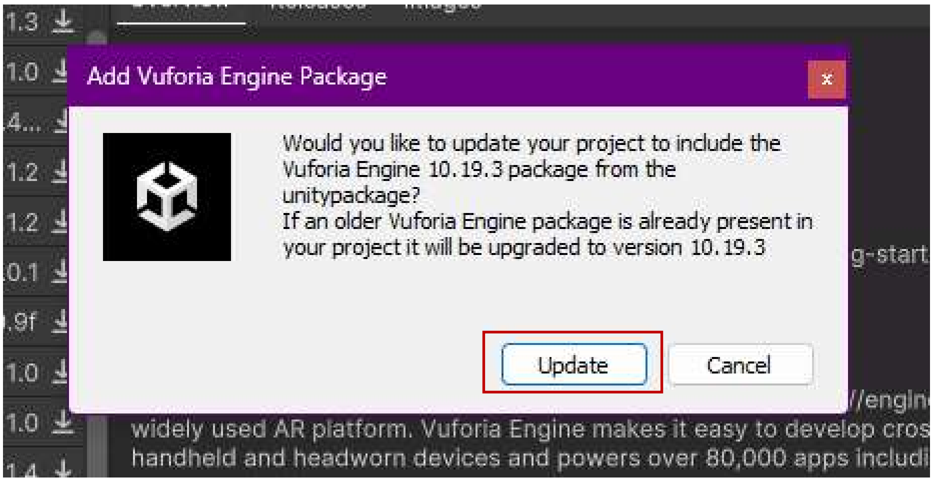
We can accept the warning that there are some updates.



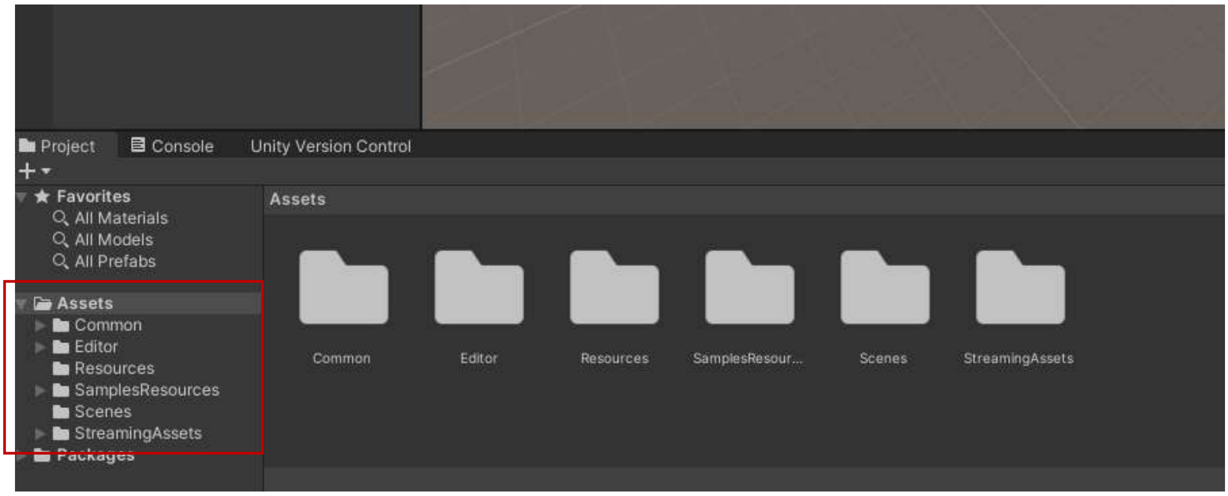
Get all the files by clicking **Next**.



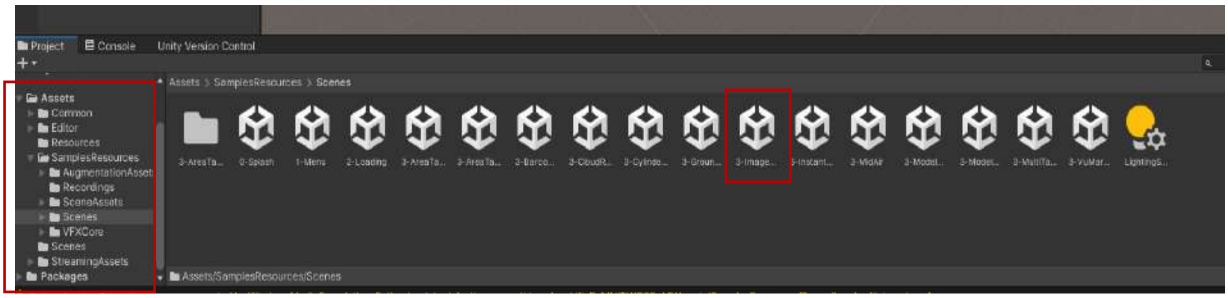
Confirm the information about the update with the **Update** button.



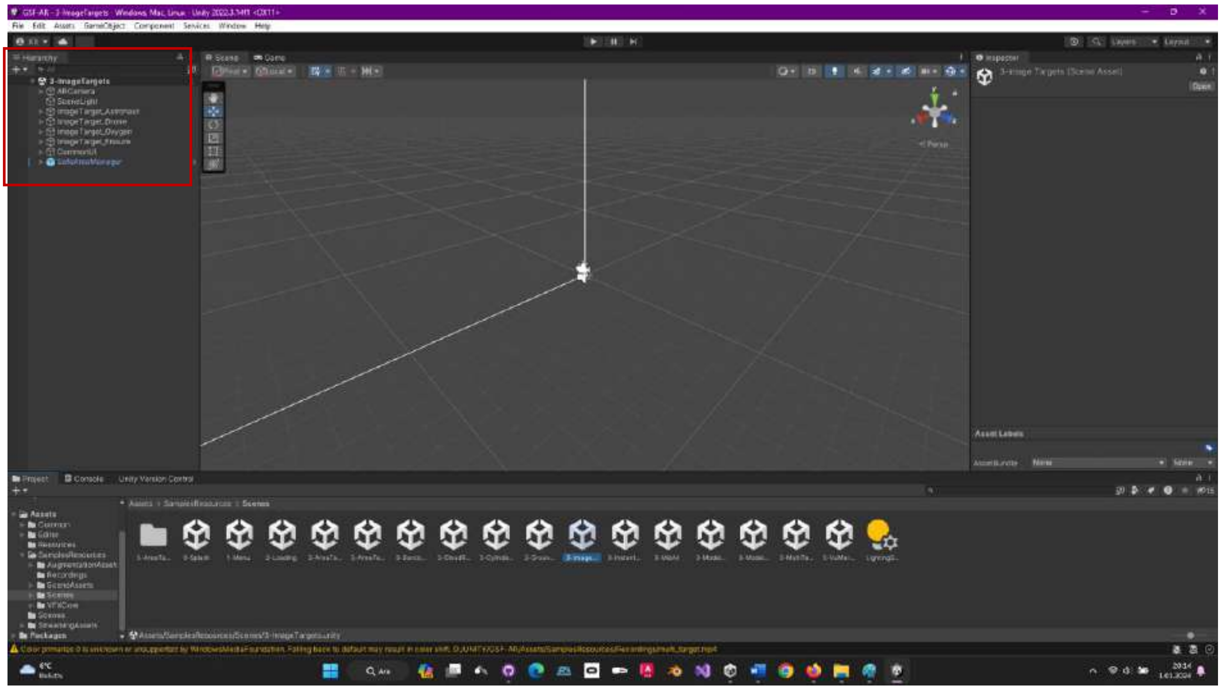
Go to our **Scene** in the project and see the changes under Assets.



When we go to **Assets->SampleResources->Scenes**, we see that there are many ready-made scenes. Here, **double-click** the **3-Image Target** scene.

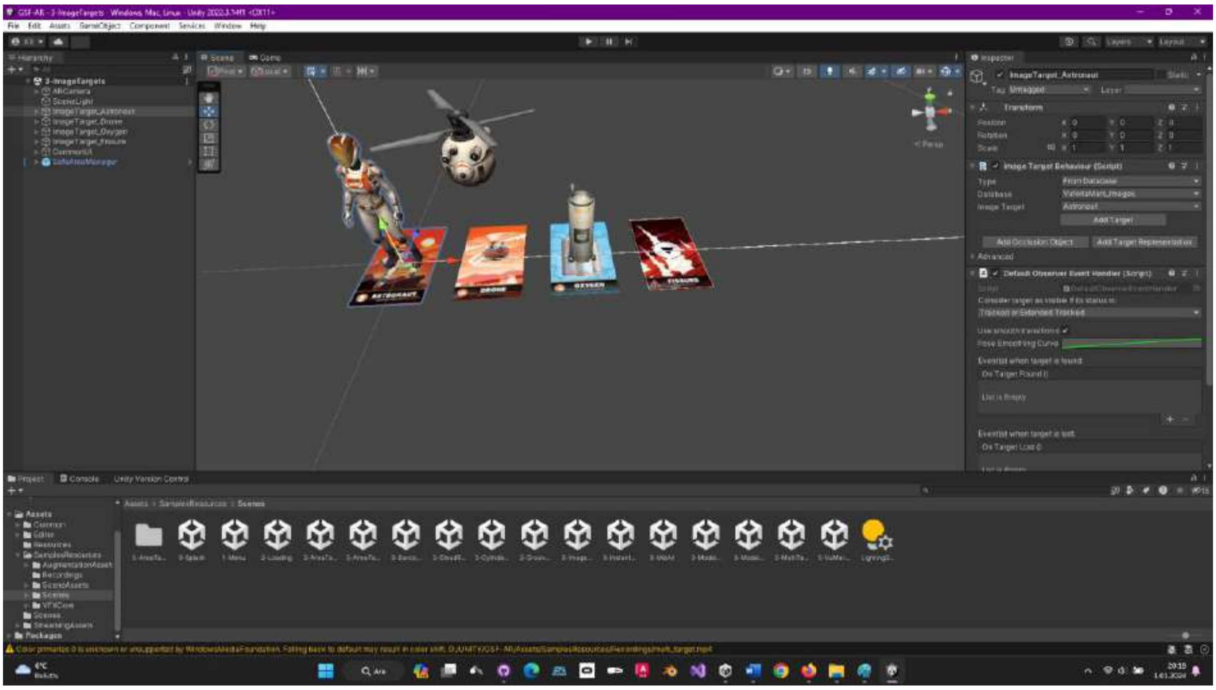


By **Image Tracking**, we will see that a **UI Canvas** has been opened. At the same time, new **GameObjects** have been opened in the **Hierarchy** window.



When we double-click on any of the objects starting with **Image**, sample objects in the 3D part of the canvas will be seen.

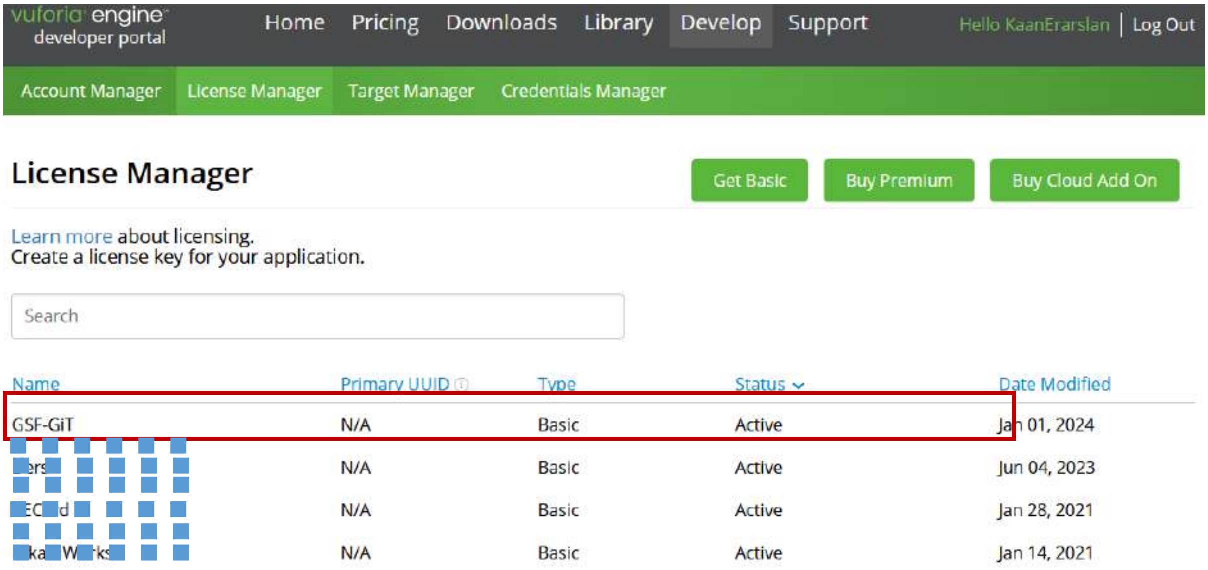
We aim to import the **GSF-LOGO** into our database, place the 3D object we want on it, and even play the video.



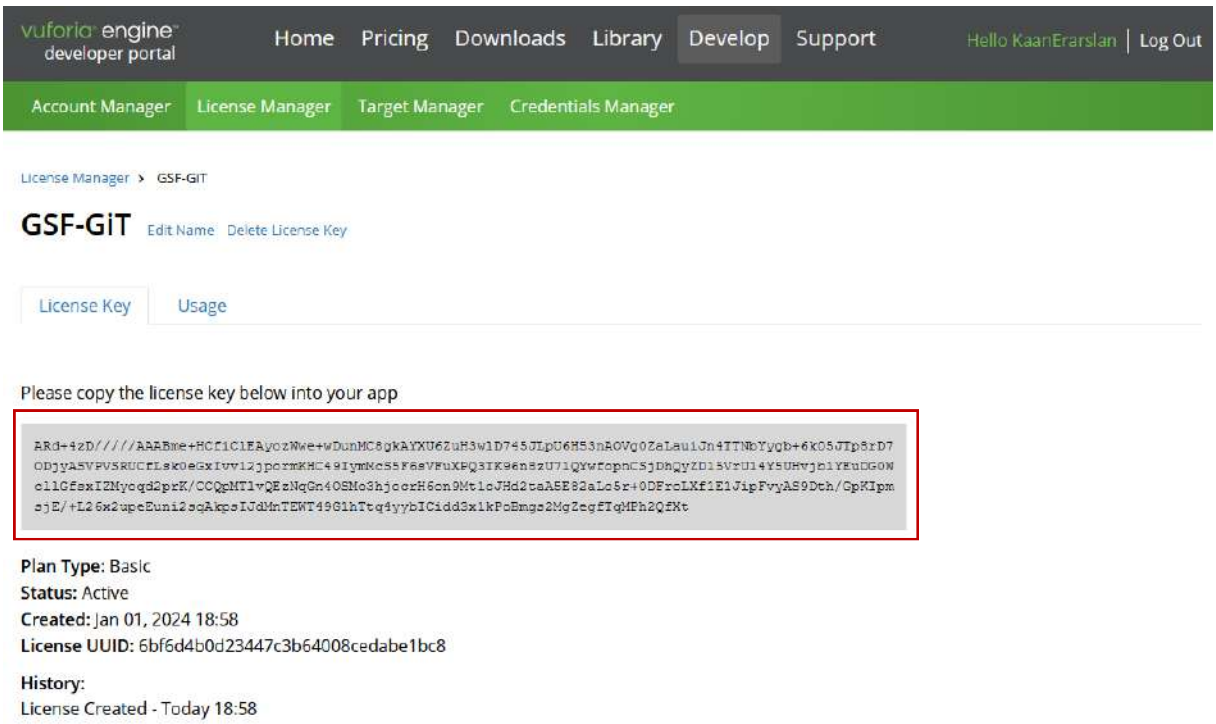
In order to perform these operations, we need to access the information and database in **Vuforia** and import it into the Unity project.

11.4.Getting the License Key and Downloading Database from Vuforia to Unity

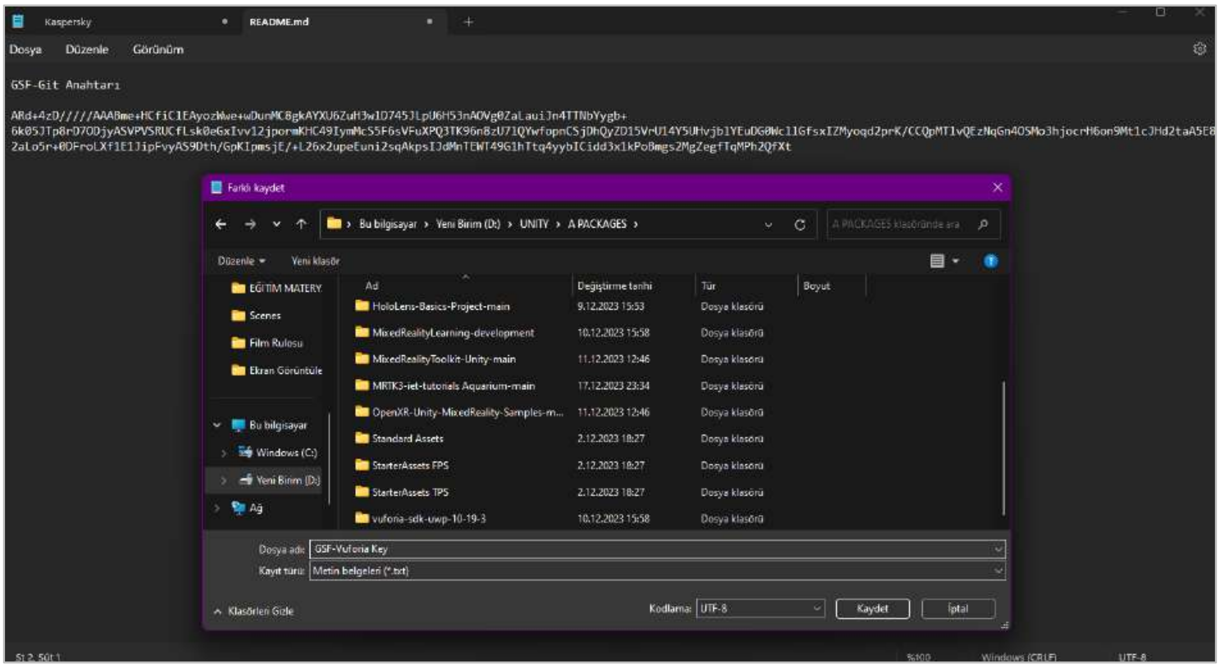
Let's go to the **License Manager** section in **Vuforia** and select **GSF-Git**.



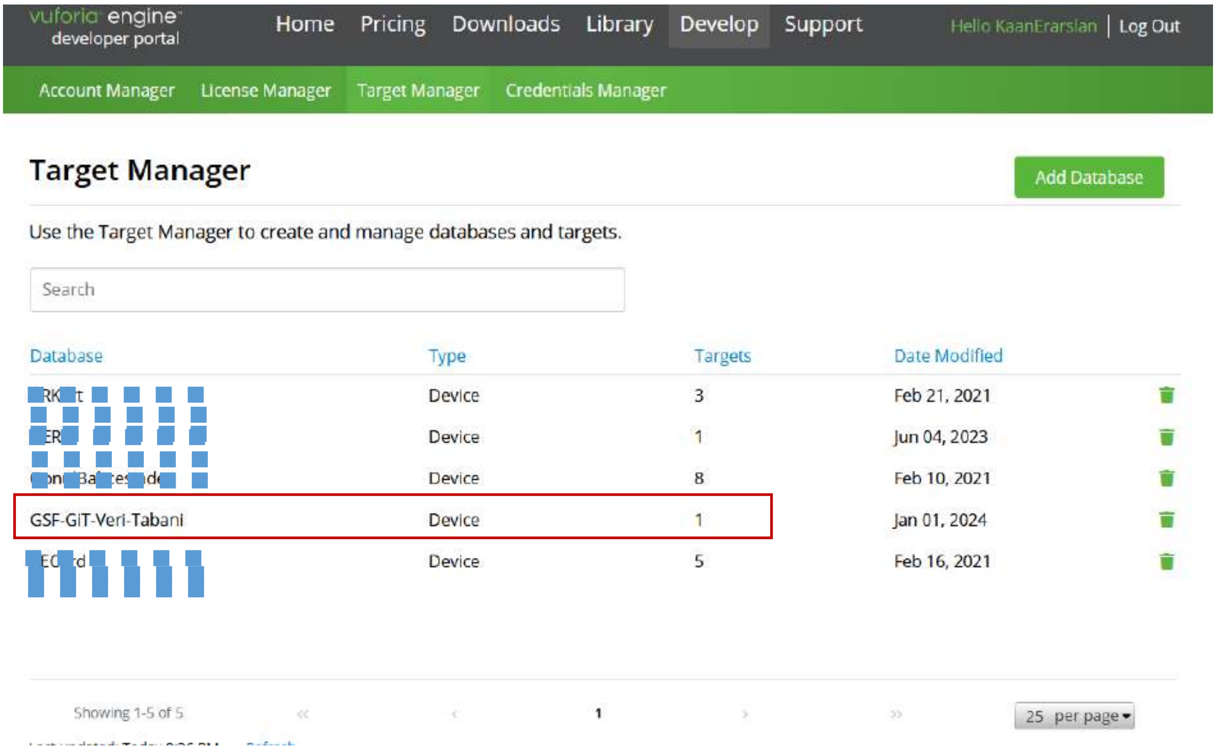
On the page that opens, there is a **license key** prepared for use. When we click on it with the mouse, this long key is copied to memory.



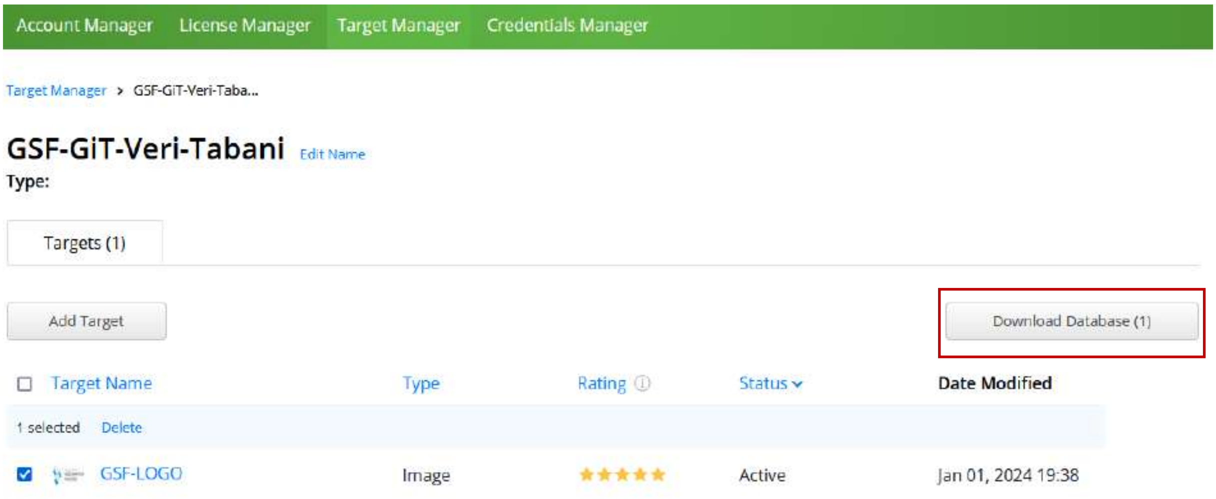
To use this key copied to memory without entering Vuforia every time, we can copy it to Notepad and save it as a text file.



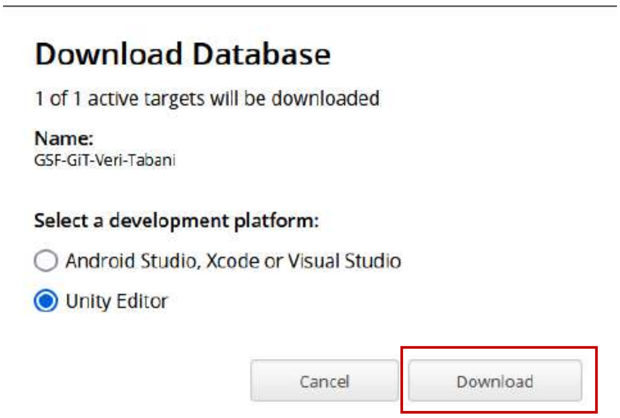
Now, go to the **Target Manager** in Vuforia. Our goal is to prepare and download the database for Unity.



Now, switch to our database named **GSF-Git-Database** with this key. To download the database that we want to download and have loaded one shape (**GSF-LOGO.png**) for now, select the box of our shape as **Target Name** and press **Download Database**.



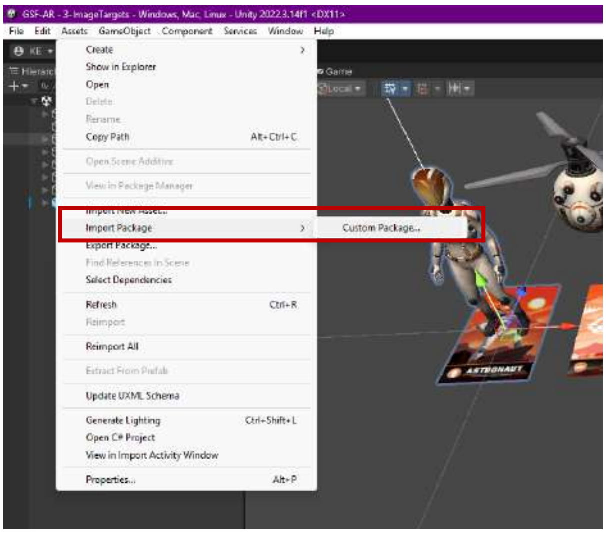
Here, Unity Editor should be selected from the options presented to us and **Download** should be clicked.



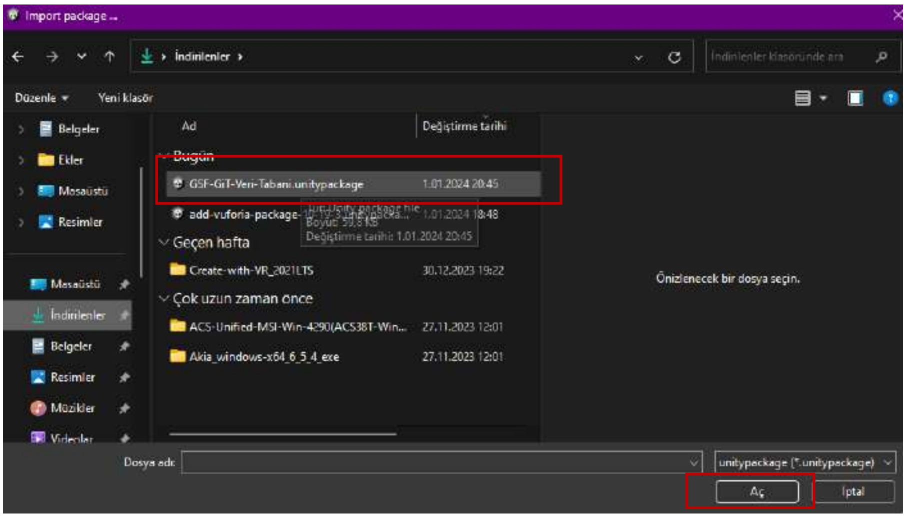
The database compilation process begins.

We can see the downloaded **unitypackage** file on the hard disk.

Go back to our Unity **scene**. Here we go to the **Assets->Import->Package->Custom Package** submenu.

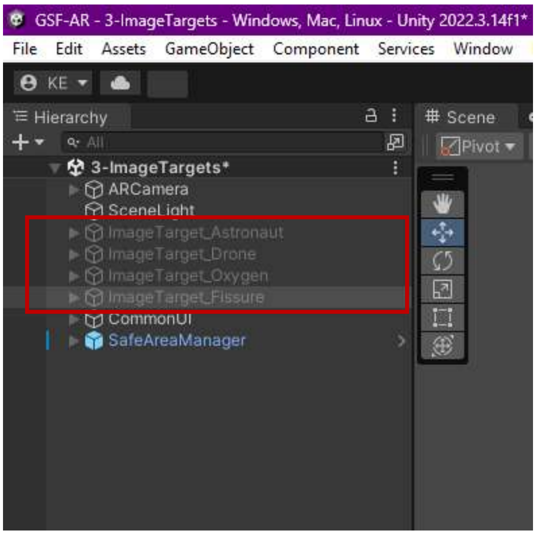


Now, add the package file we just downloaded from Vuforia to our project.



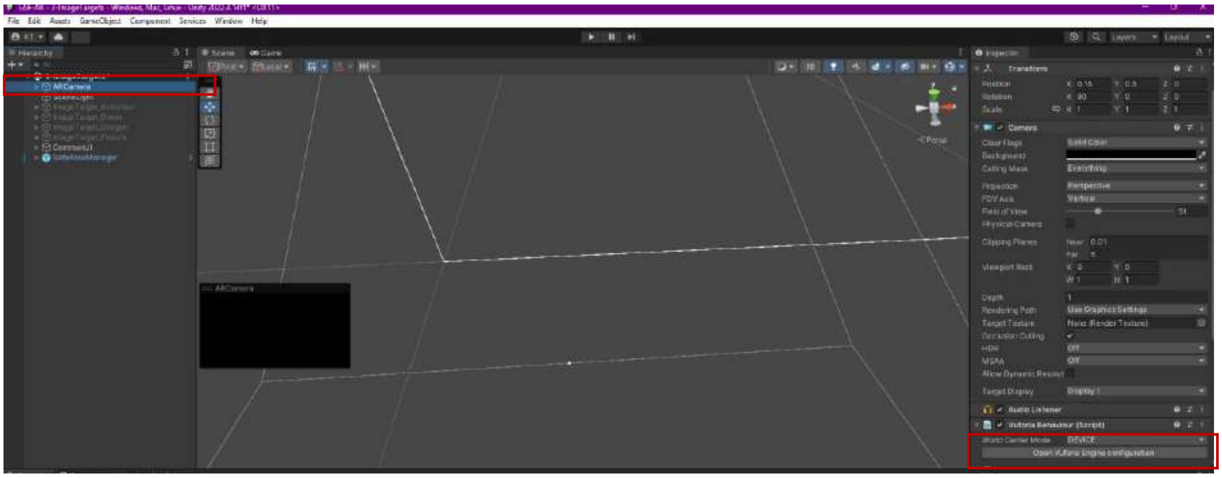
Then, **Import**.

Make the sample objects such as **Astronaut, Drone, Oxygen,** and **Fissure** in the **Hierarchy** section of our scene invisible by clicking on their boxes.

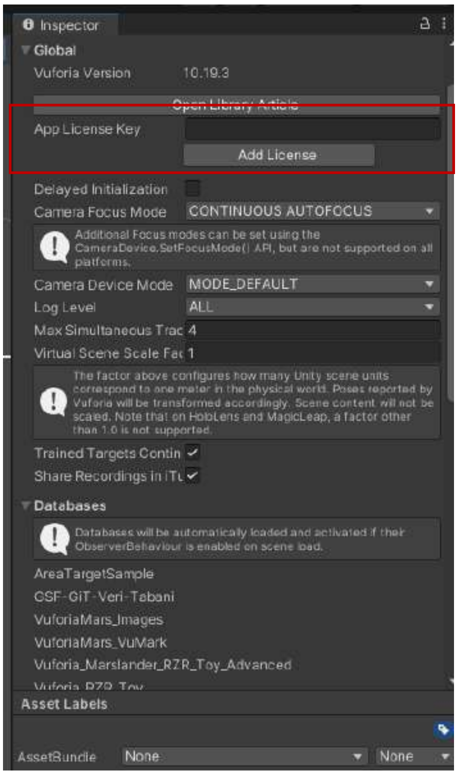


Now it's time to get our database object instead.

Go to **ARCamera** and open the settings in the **Inspector** section. Press **Vuforia Engine** configuration.

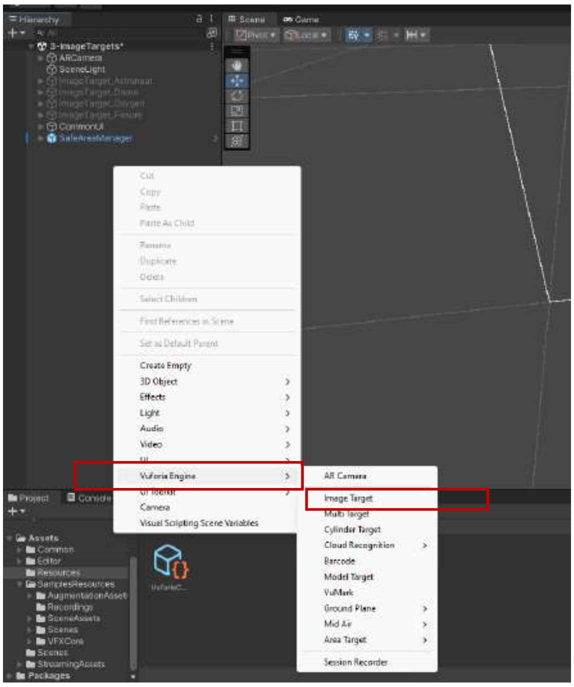


Here, a window will appear where we will enter the Vuforia **database key**. For this purpose, let's see the box called **App License Key**. As you may recall, the key that we memorized in Vuforia **License Manager** and that we can save in **Notepad** so that we can use it in our future work will be placed in this box.

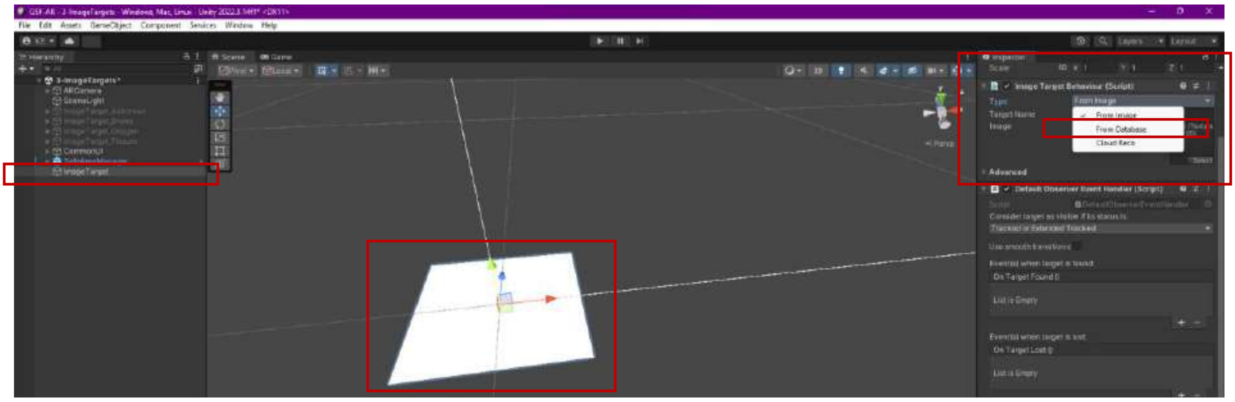


Paste the license key...

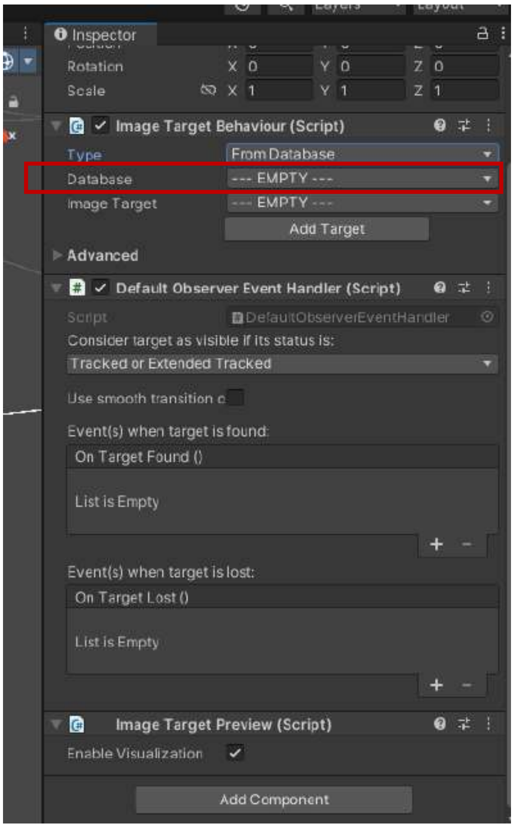
After removing the sample objects, we need to place our **Image** object. To do this, select **Vuforia>Image Target** from the menu that opens with the right click of our mouse in the **Hierarchy** section.



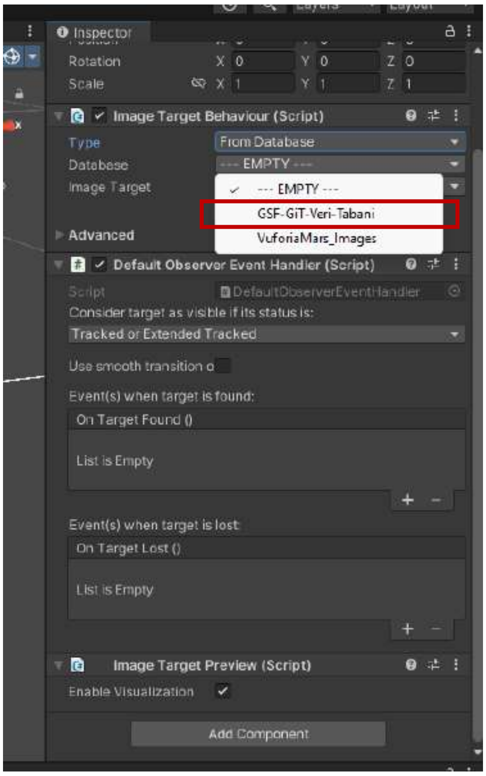
A plane-like object is placed in our scene. Open the menu to the **Inspector->Type** section of this object. Click on the **From Database** option in this menu.



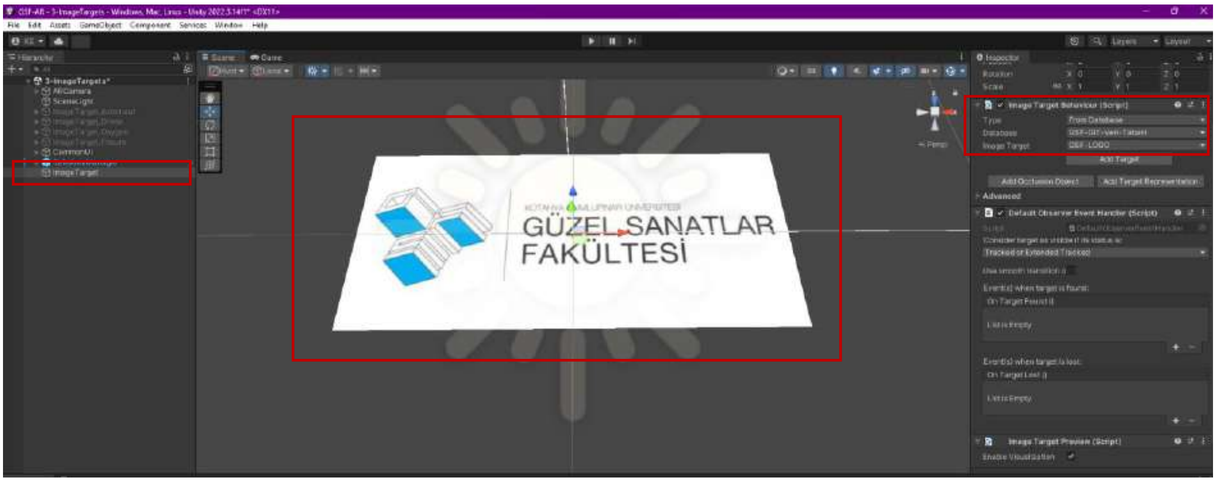
In the new settings that appear, there is a **Database** option under **Type**. We will import the database that we created in Vuforia and download it in **unitypackage** format here.



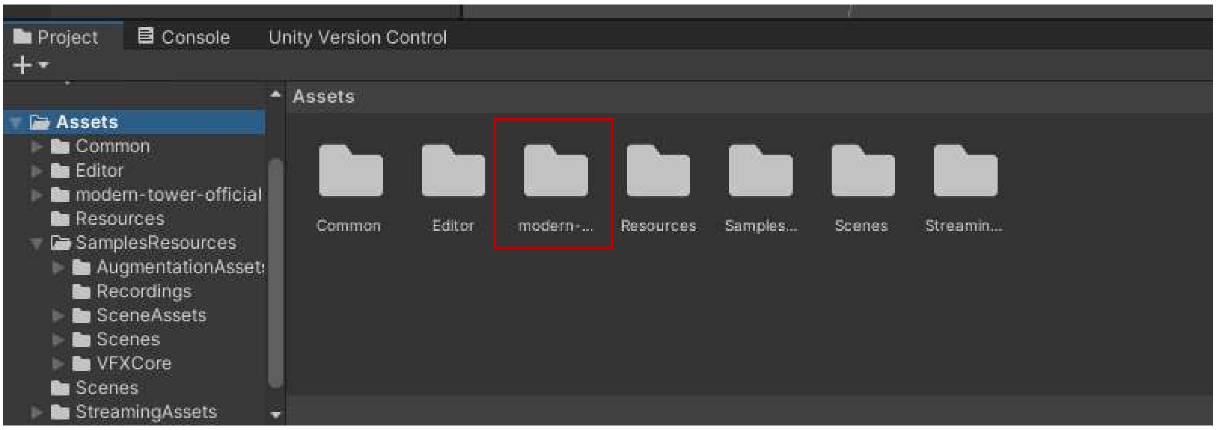
When you click on the **Database** section, you see our database in the list that opens. Select it…



With the selection, the **GSF-LOGO** figure will be placed on the **Image** on the stage.

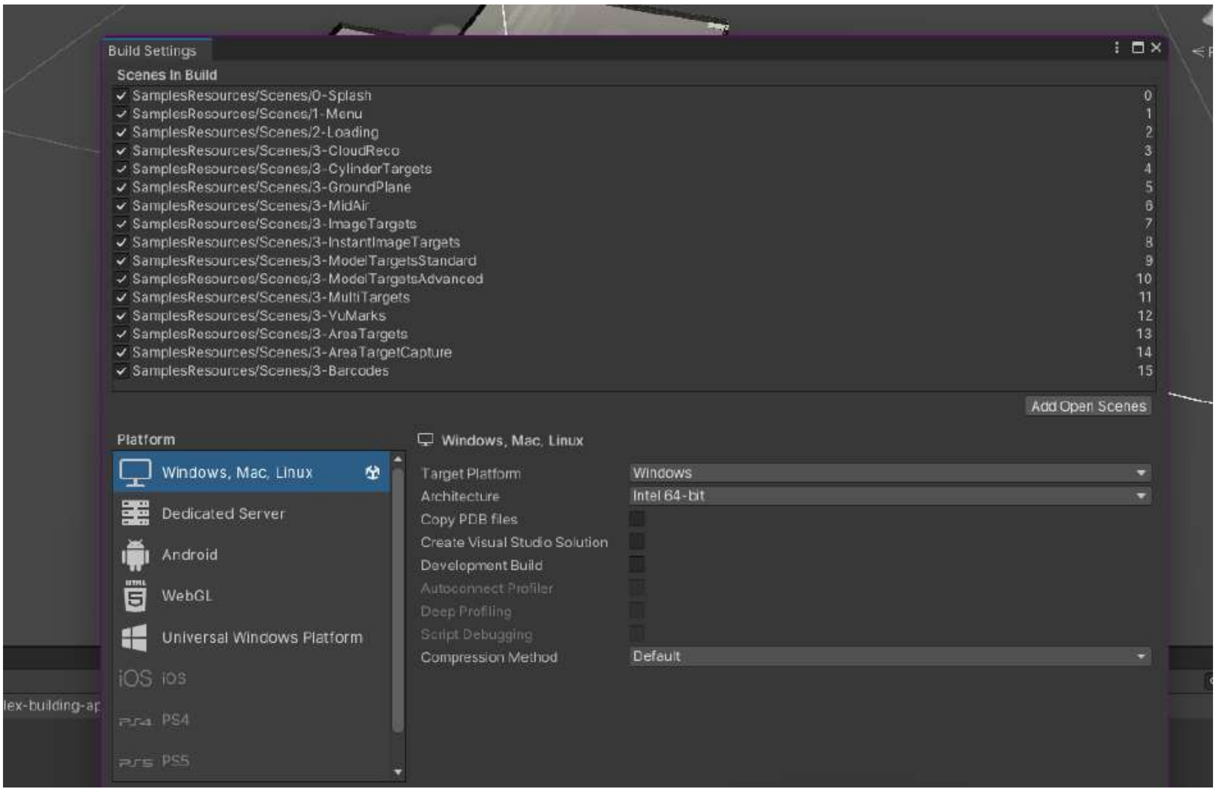


We plan to place a **3D school model** on the **Image** in the scene. We find a low poly building from **Sketchfab** and place it in the **Asset** folder.



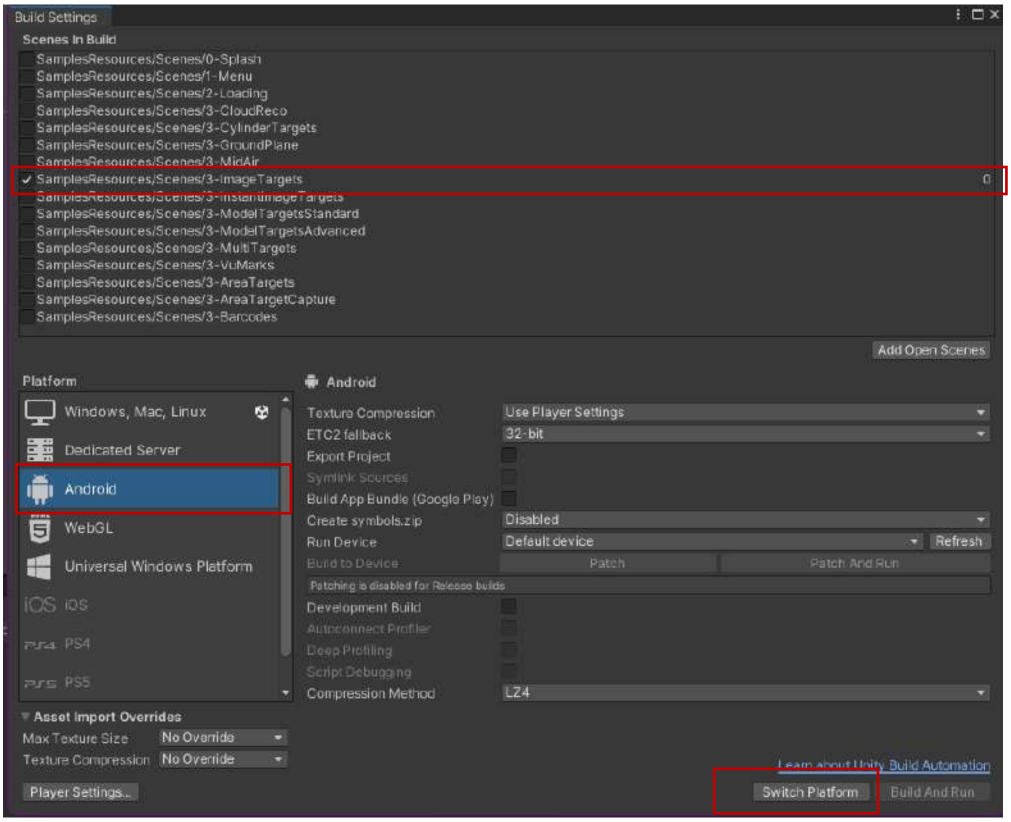
This 3D building model is placed on top of the **GSF-LOGO** plate taken from our database after various size reductions and positionings.

Now, it is time to deploy our mobile device. For this, the **Build Settings** window opens. There are many scenes listed in the **Scenes in Build** section.

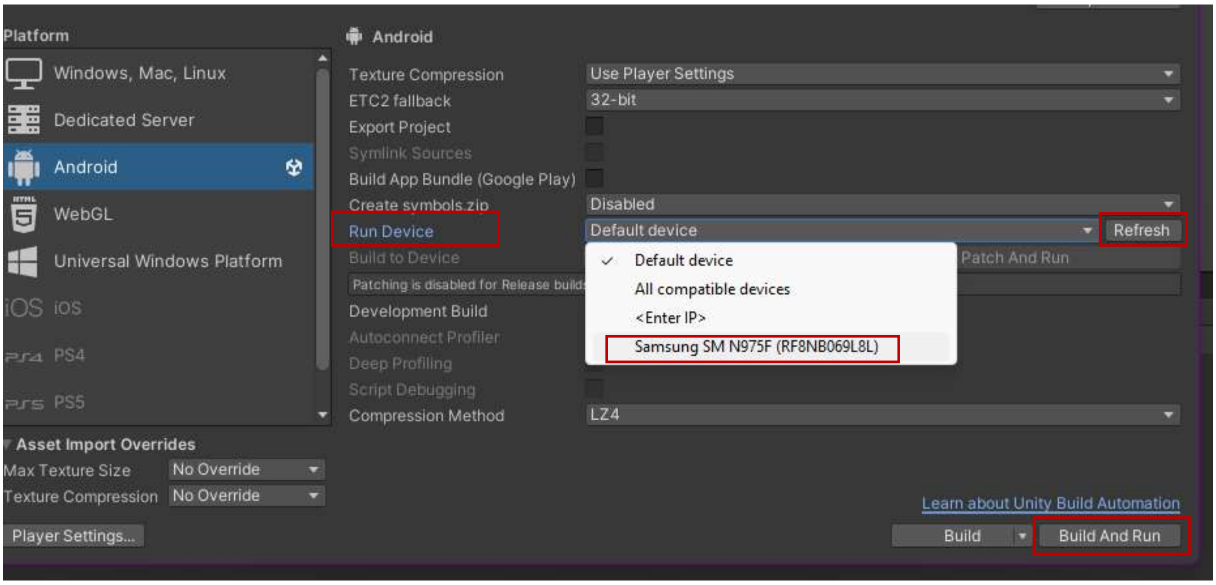


However, since our project is about **Image Target/Tracking**, the boxes of the ones other than **Image Targets** are checked and excluded from the process.

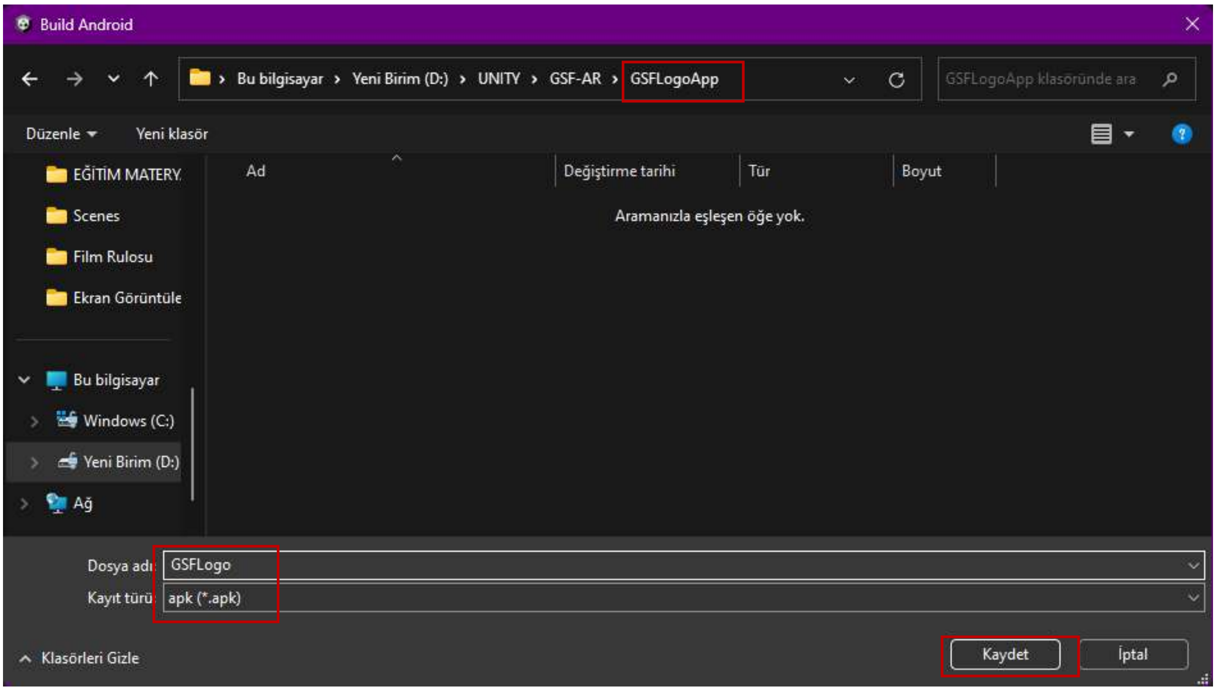
Also, since this will be a mobile application, the **Switch Platform** process is done by selecting the **Android** platform.



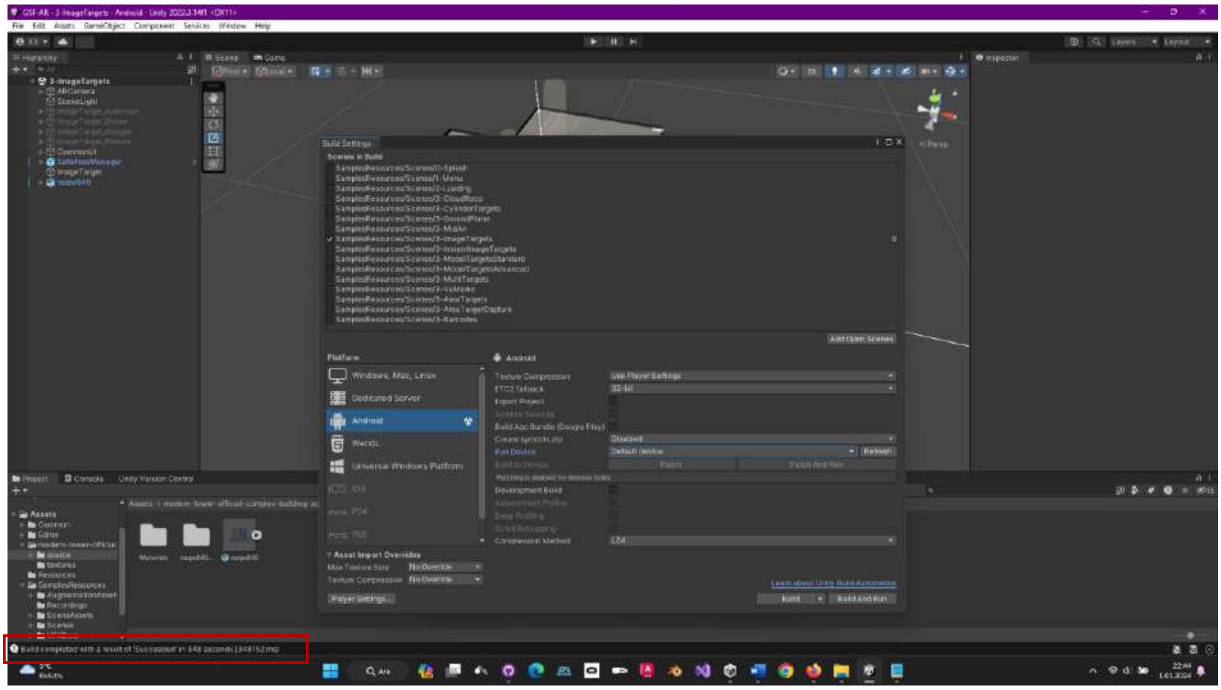
Now, we can connect our phone with developer features to the computer with a cable and **Refresh** the **Run Device** section.



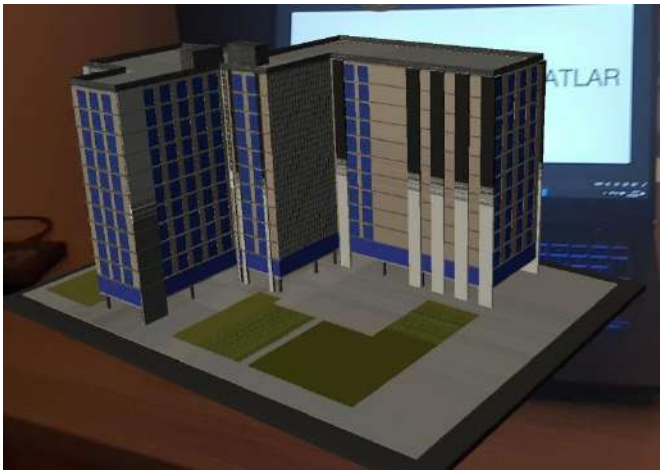
When we select the **Build and Run** section, in the window that opens, we create a new folder called **GSFLogoApp** with the right click of our mouse to register the application, and we specify the name of our **APK** file as **GSFLogo.apk**.



The printout process starts with the save button. It will take a few minutes to complete. Finally, it returns to our stage, stating that it was successful.



When the application was opened on the phone, the **GSF Logo** was placed on the computer screen and tested. As a result, the building was displayed in front of the screen as an augmented reality application.



This application can be added to the **displacement** and **rotation** operations called **Lean and Touch**.

It is also possible to place a **Plane** instead of placing a 3D model on the logo that comes to the stage as a plane and turn that **Plane** into a video player by adding a **Video Player**.

Another application alternative provided by Vuforia is the application called **Ground Plane**, which allows the desired 3D object and video player to be positioned on the **ground**, independent of any Image Target, because of ground scanning.

Again, after scanning a 3D space, augmented reality objects can be added to the desired objects or points.

It is planned to prepare additional notes for this and similar applications. However, independent of the notes, there are many supporting materials in digital media, primarily their resources.

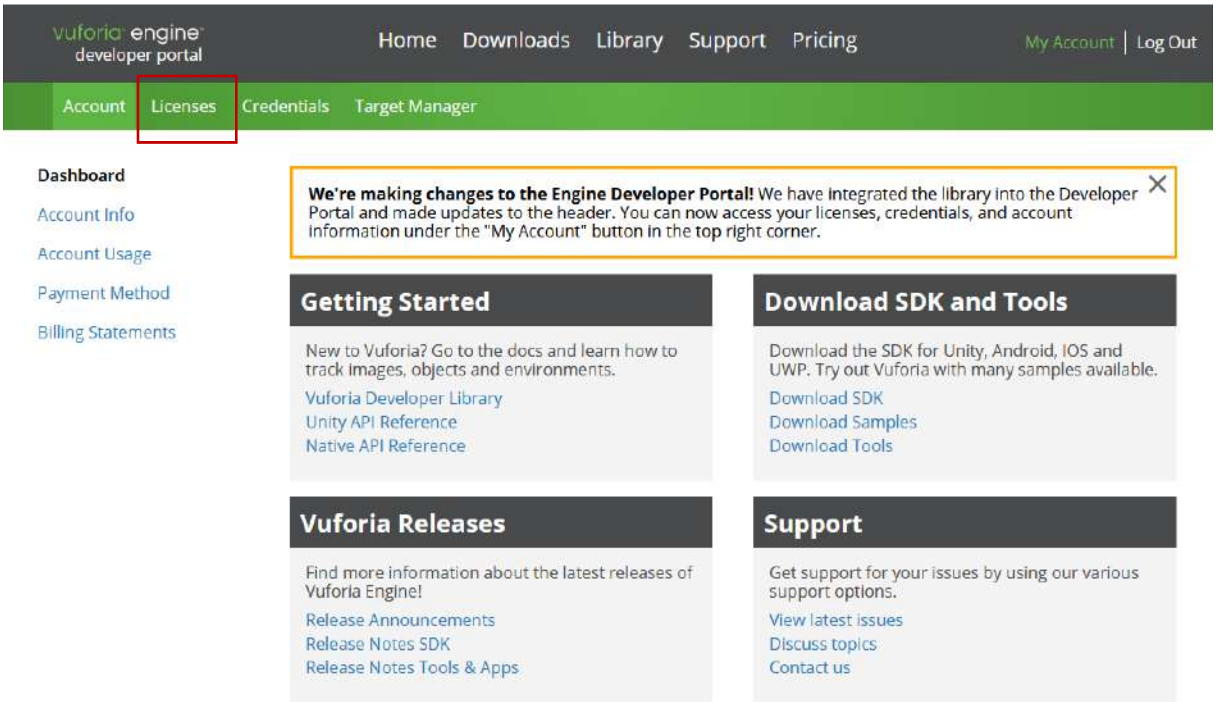
11.5.AR-Vuforia and Multi-Image Target

In addition to the single image-target application under the title of Augmented Reality, a similar study can be done on **multiple** images. Essentially, this means placing many images in the Vuforia Database and ensuring their recognition within an application of Unity. Therefore, a 3D model, animation or video can be assigned to each shape (picture). In this way, AR studies can also be done collectively on the pictures in the book (catalogue), called **AR-Book** (**Augmented Reality Book**).

Vuforia AR application is based on the principle of converting image file(s) uploaded to a database created here into a Unity package, downloading them and entering them into the Unity project. Let's repeat the upload and Unity package downloading process previously done for the **GSF logo** for the Engineering Faculty and Dumlupınar University logos.

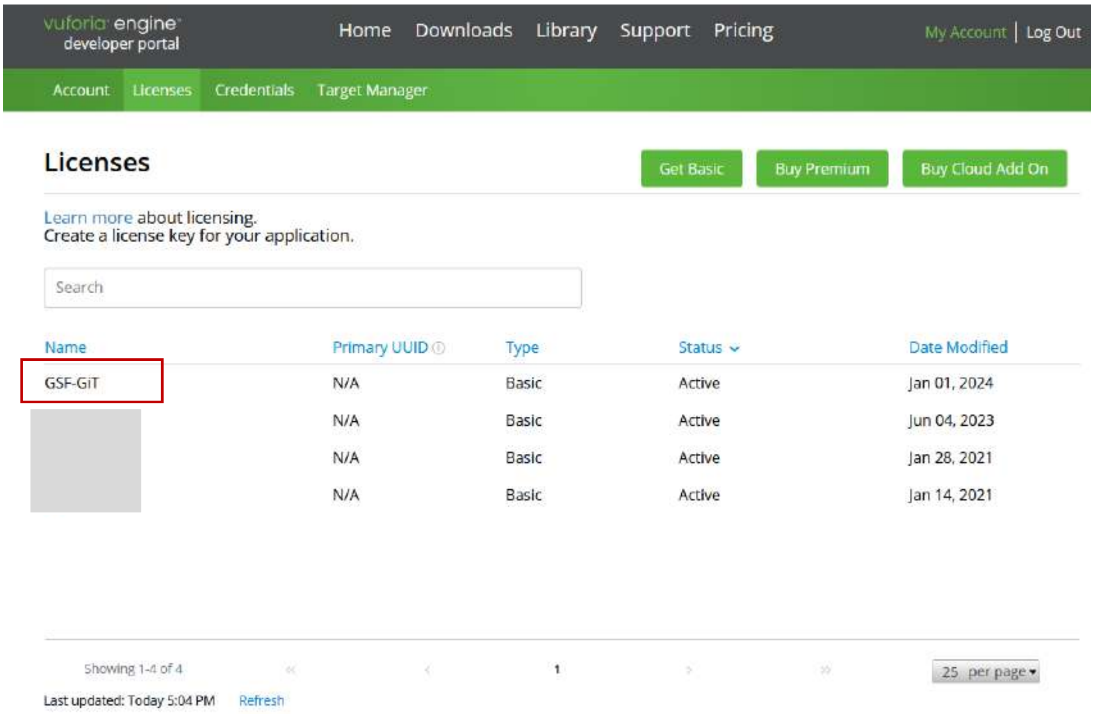
It is thought that explaining and repeating some parts of the first application from the beginning would be beneficial, and there will be partial similarities with the previous explanation.

Let's log in to our **Vuforia** account.

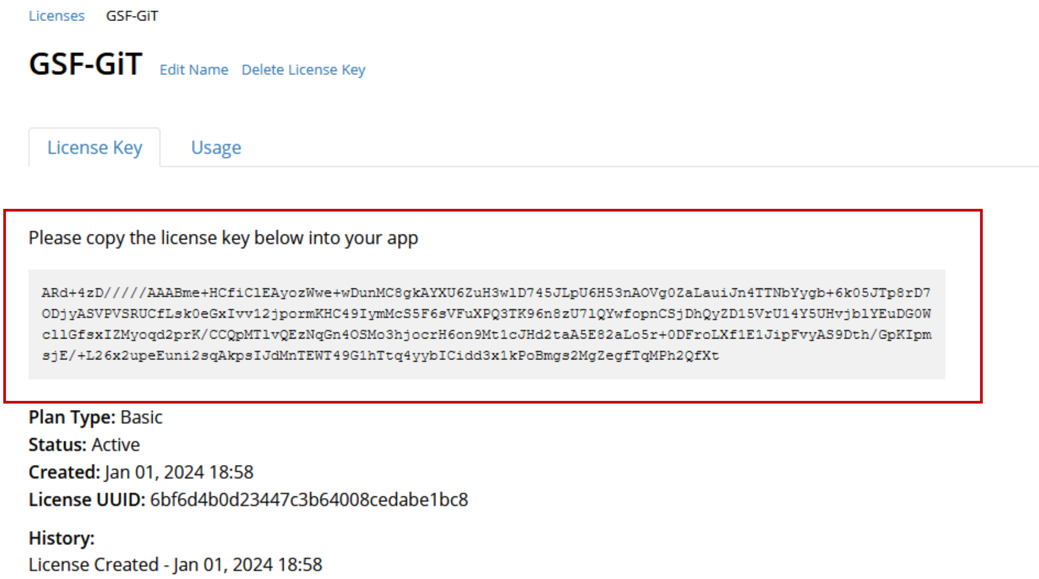


Here we go to the **Licenses** tab.

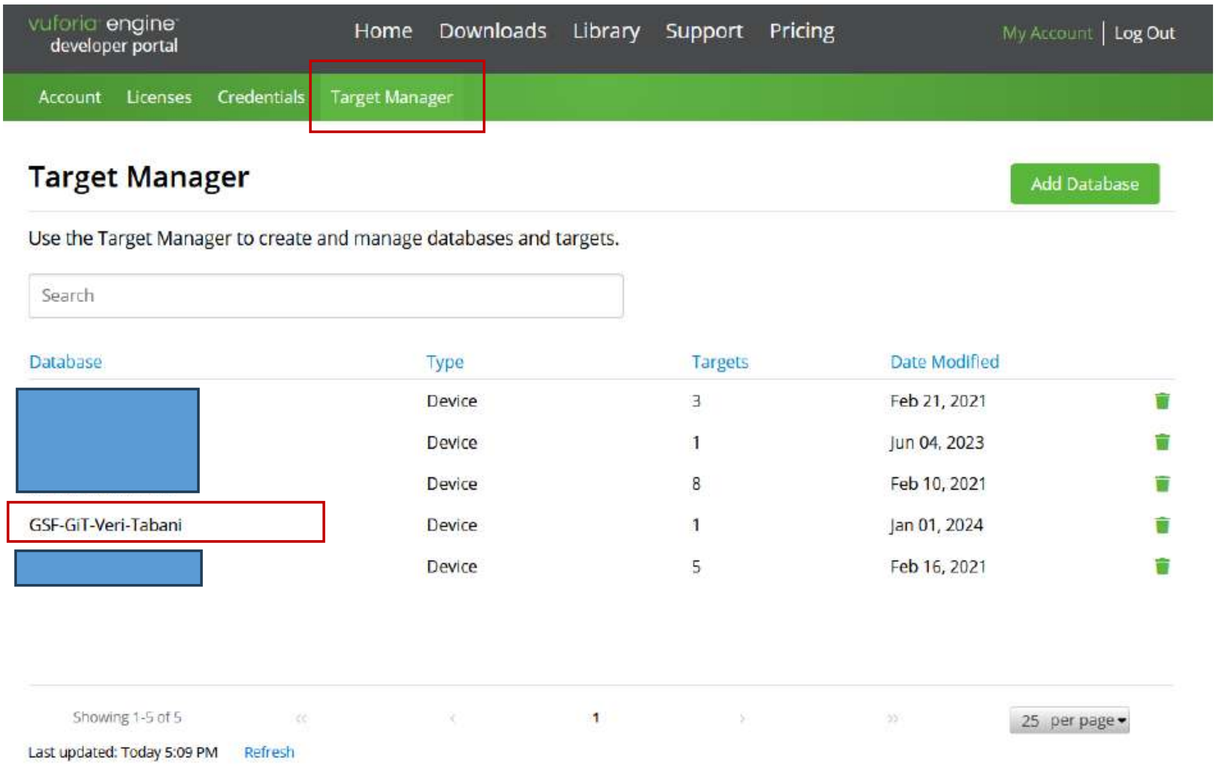
All licenses we created before will be opened.



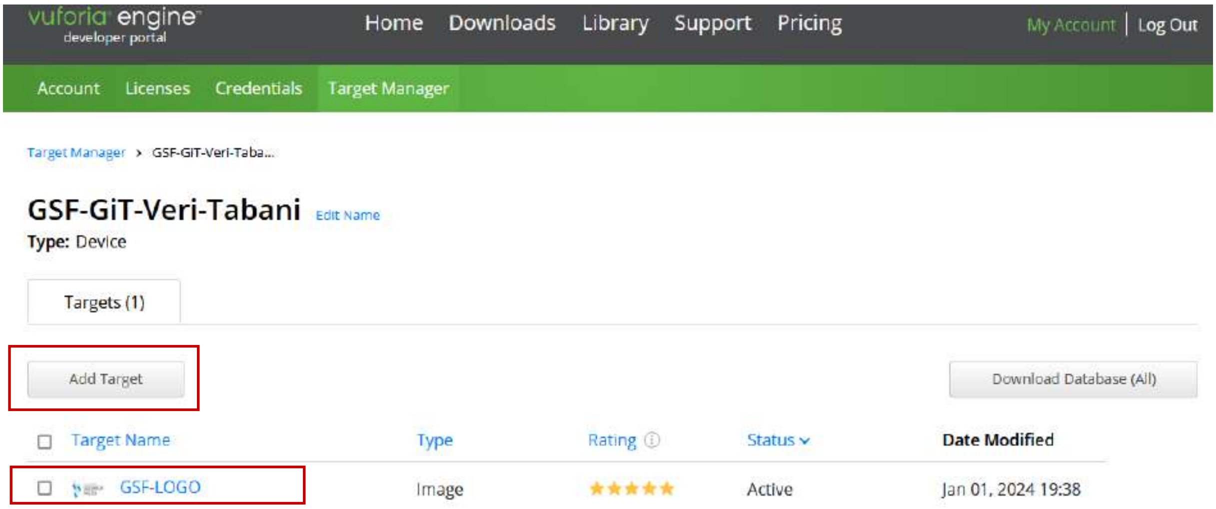
Click **GSF-GIT** and copy its license by clicking with our mouse. This license information will be needed in our Unity project.



Now, go to Target Manager and select the title we previously named **GSF-GIT Database**.

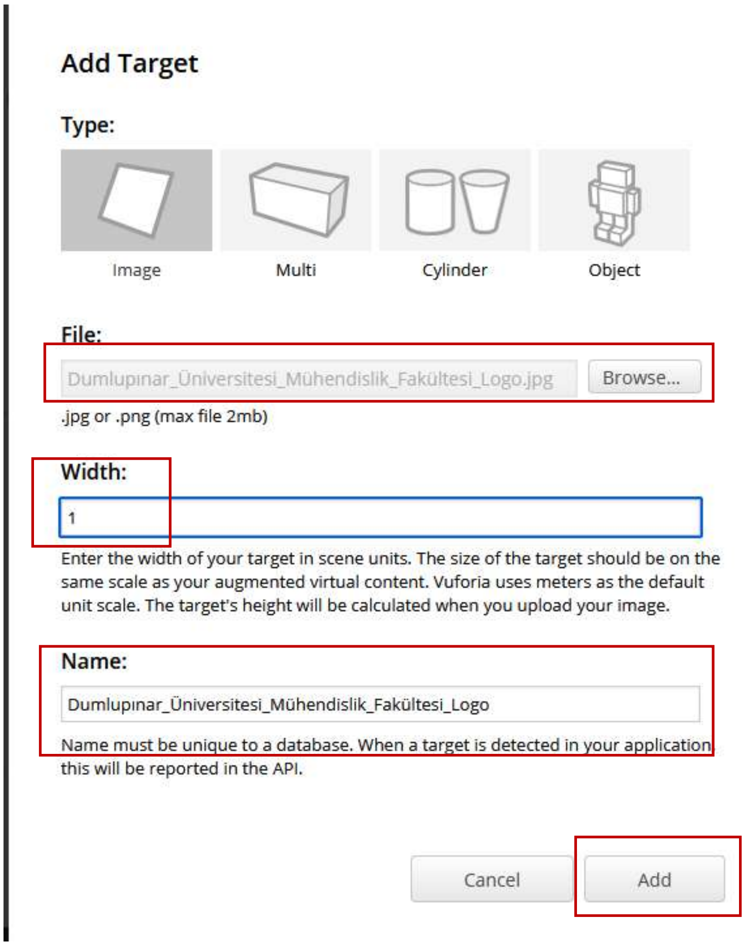


Here we will see that we have only uploaded an image file named **GSF-Logo**. Let's click **Add Target** to add two logo files.

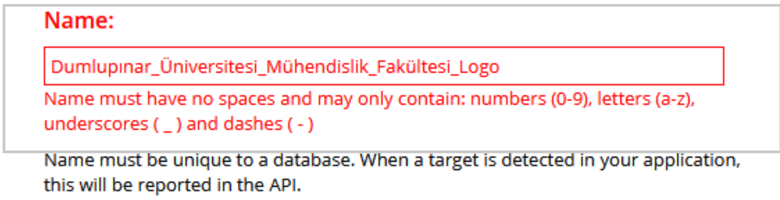


First, we select the Faculty of Engineering logo file and add it (**Add**).

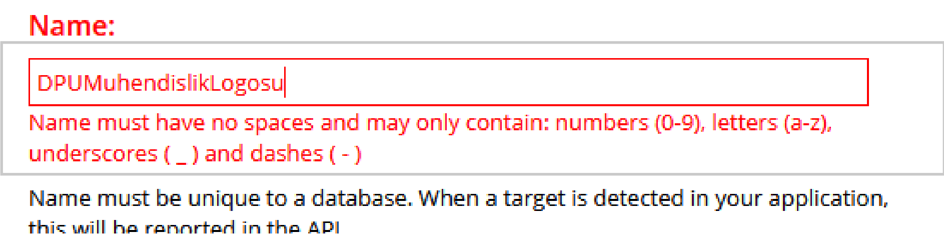
(Note: Possible errors are also shown)



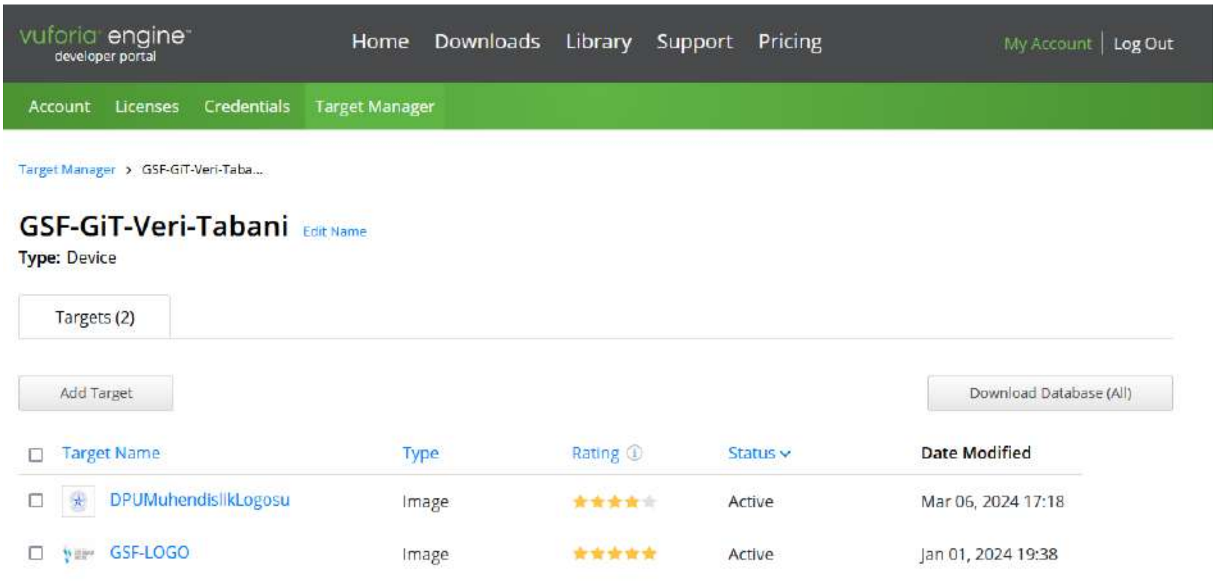
Here, the letter **Ü** caused problems because it does not exist in English.



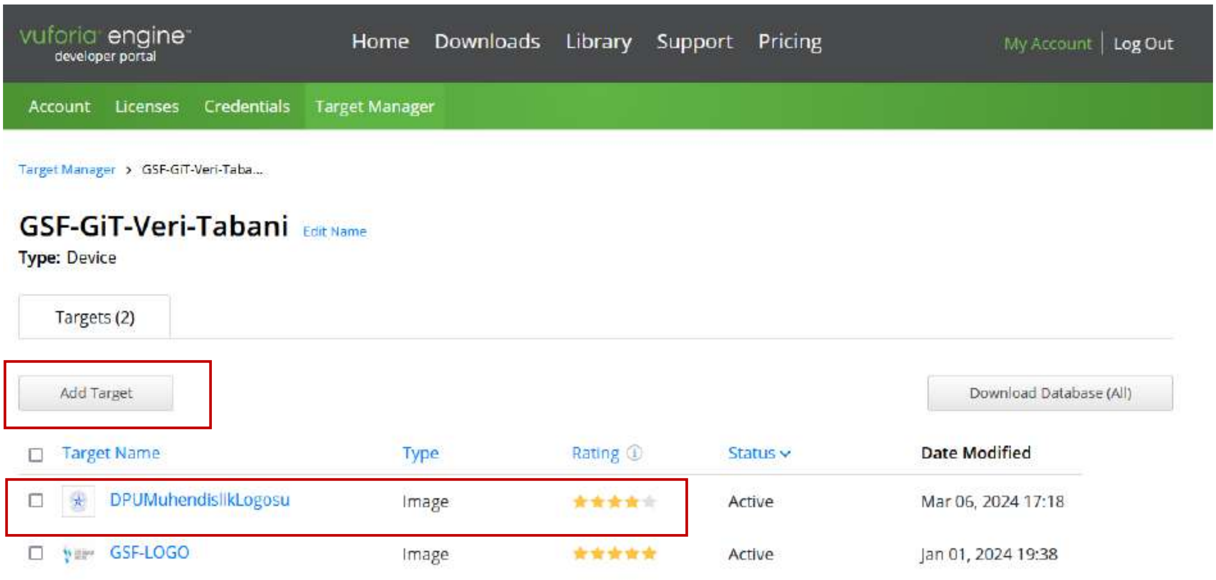
Let's **rename** and add.



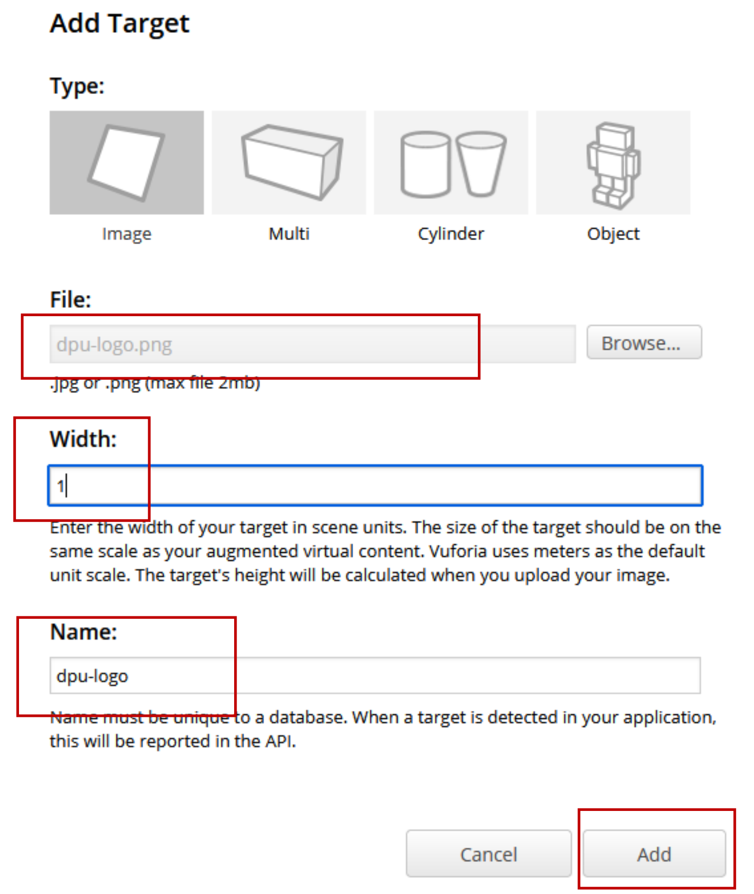
Now the upload will start. If we wait a bit and refresh the page, the quality of our shape will be rated with stars. 4 stars is a quality level that is considered good.



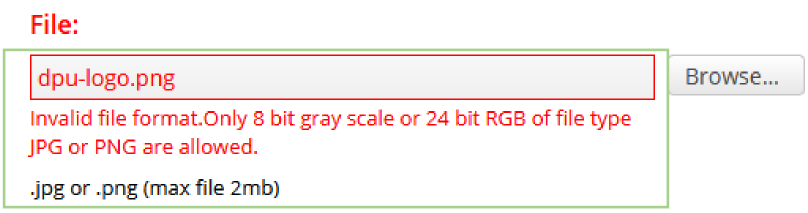
Now, upload the university logo with **Add Target**.



In the window that opens, select the DPU logo file and add it (**Add**).

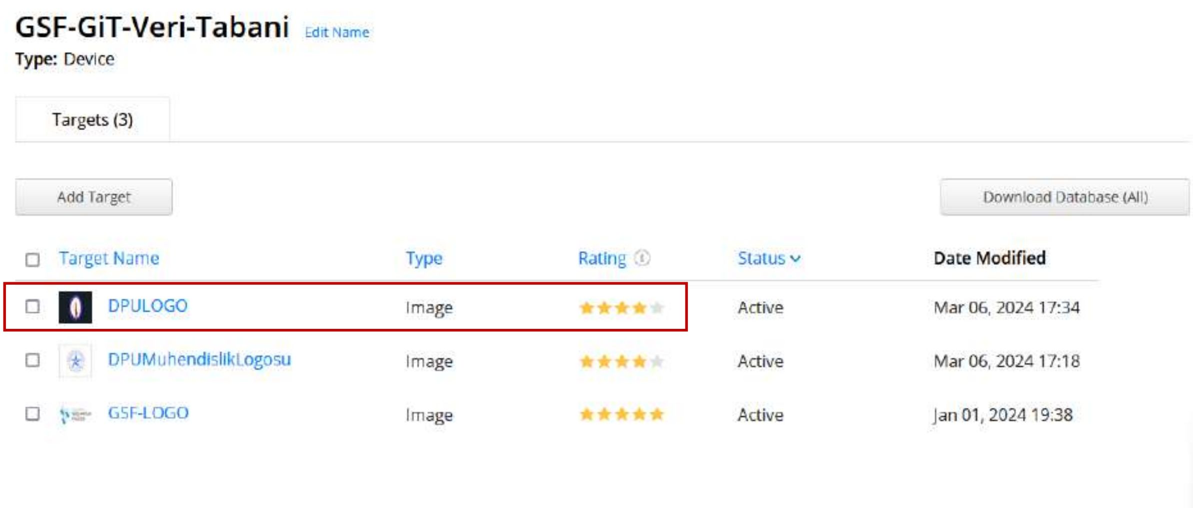


Here we encountered another error. Vuforia is a program that distinguishes images. So, try again by selecting a new file.

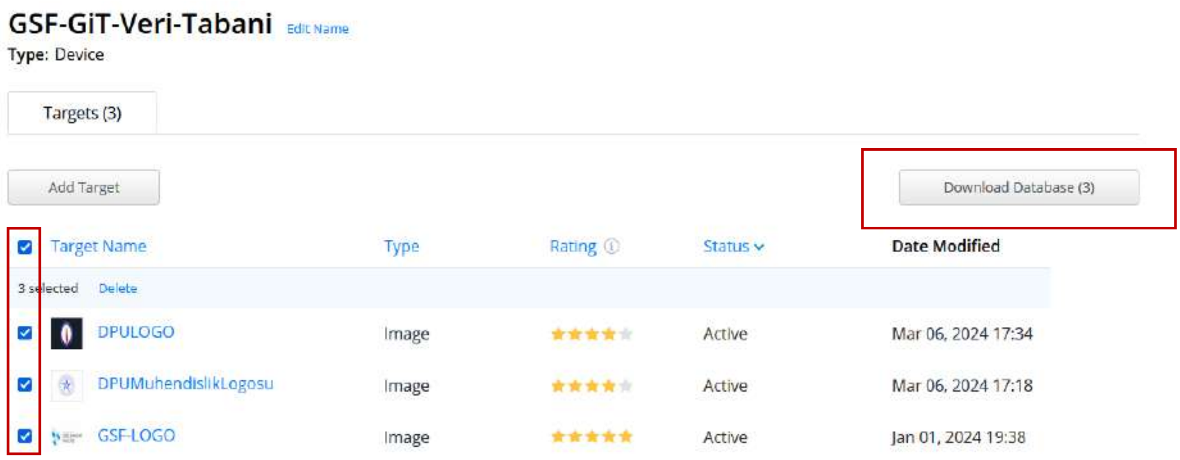


Try again after making changes within the warnings or taking the shape from another source.

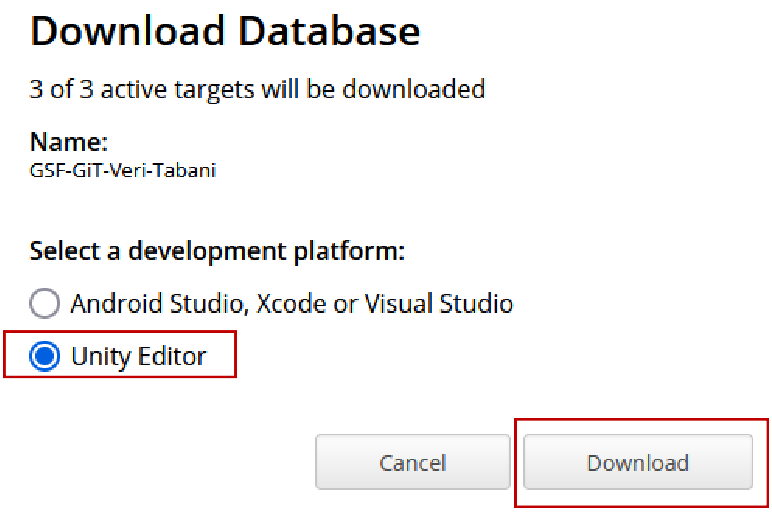
This image file is also rated 4 stars and is available.



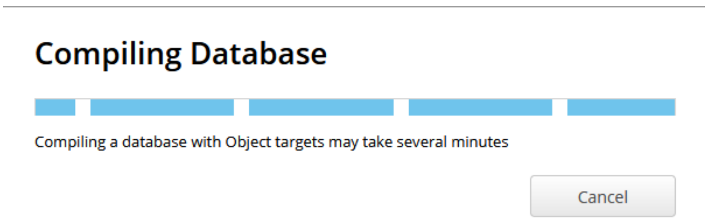
Then, select all these files and click on **Download Database (all)**, where there are 3 images.



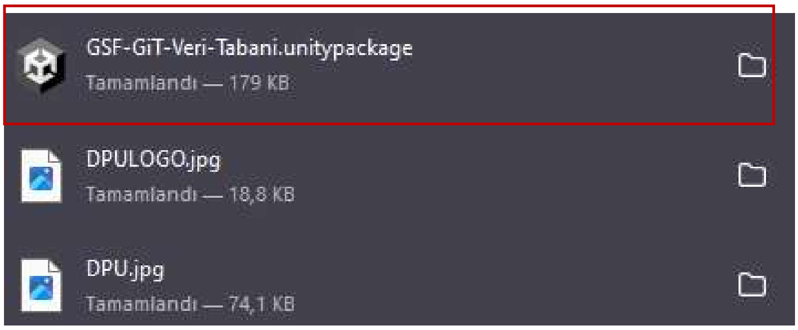
In the window that opens, select the Unity Editor and download it (**Download**).



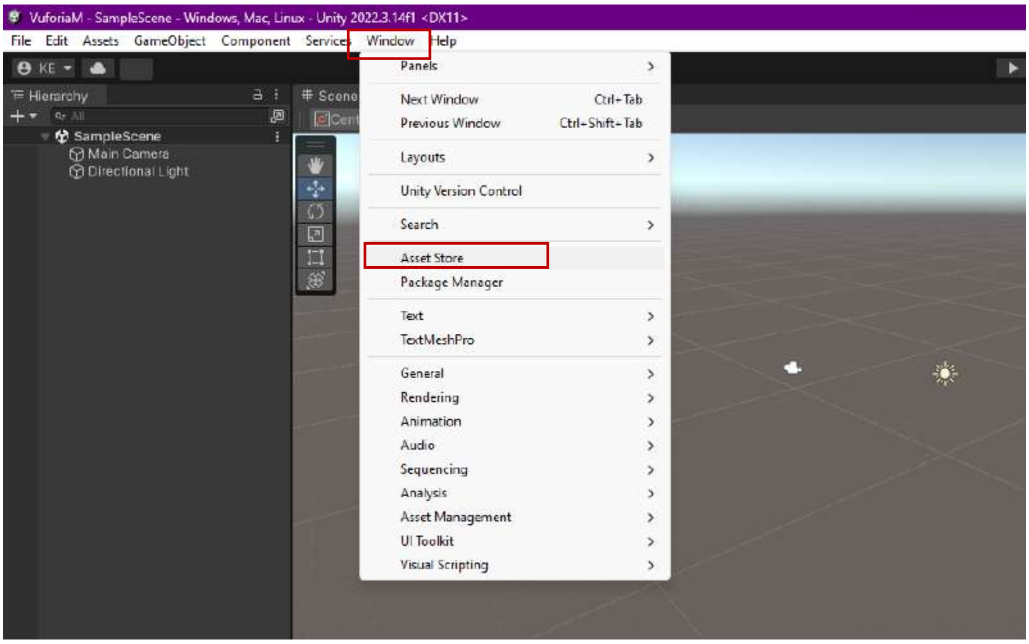
Vuforia starts compiling the database for Unity.



In our web browser, we see that the file named **GSF-GİTVeri_Tabani.unitypackage** has been downloaded to the hard disk.

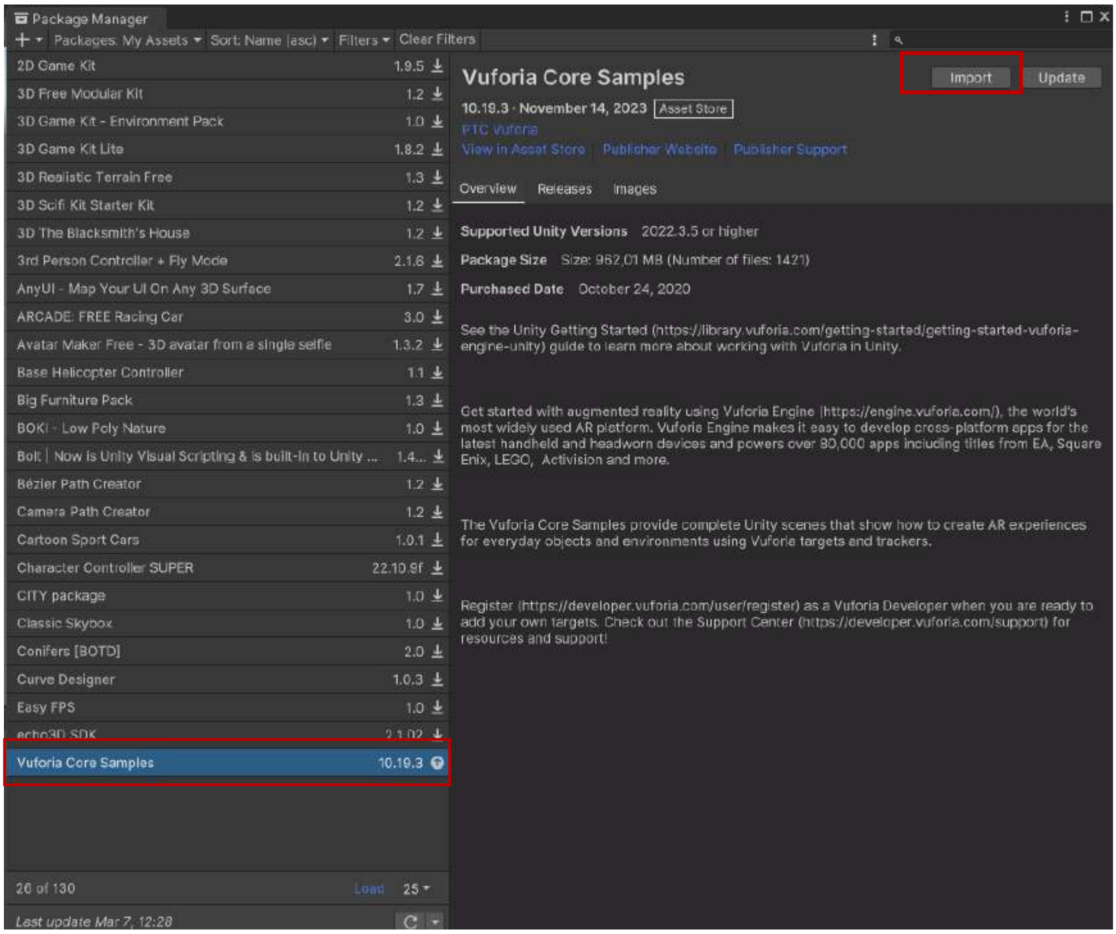


Create a project in Unity. Click **Window>Asset Store>Search Online**.

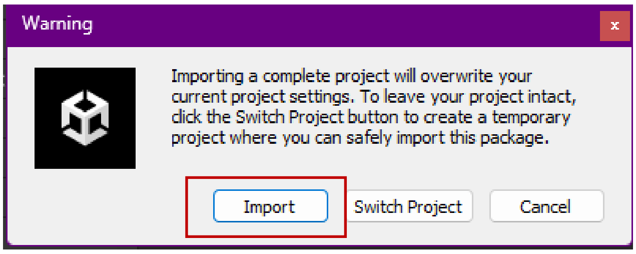


Here, search for Vuforia Core Samples and open it in Unity by selecting it. It will be displayed directly under the **Package Manager**. Since we have downloaded it before, we can continue by **Importing**. If we have not downloaded it before, we can select **Download** or **Update** if there is an update.

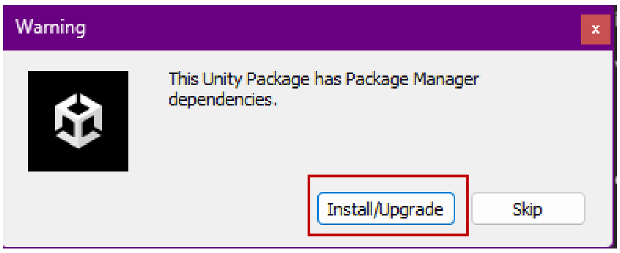
Vuforia recommends a minimum Unity version of 2022.3.16.



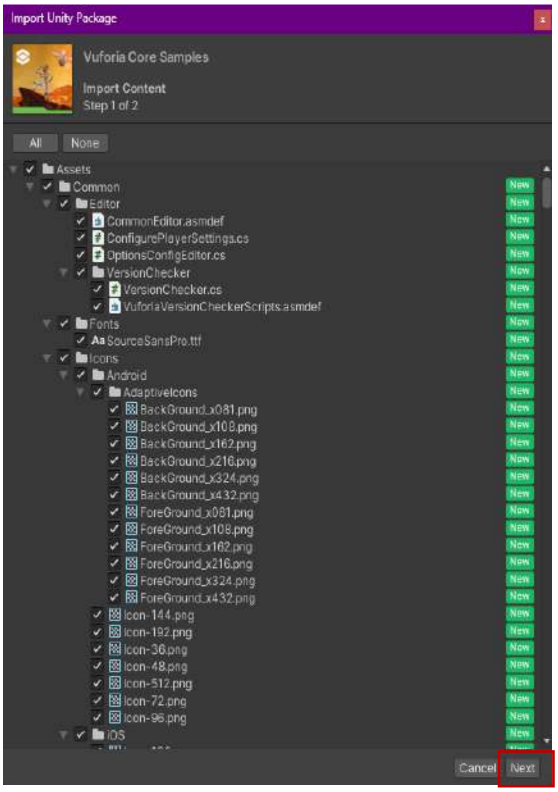
After clicking **Import**, if a warning comes up that the project will be processed, continue by clicking **Import**.

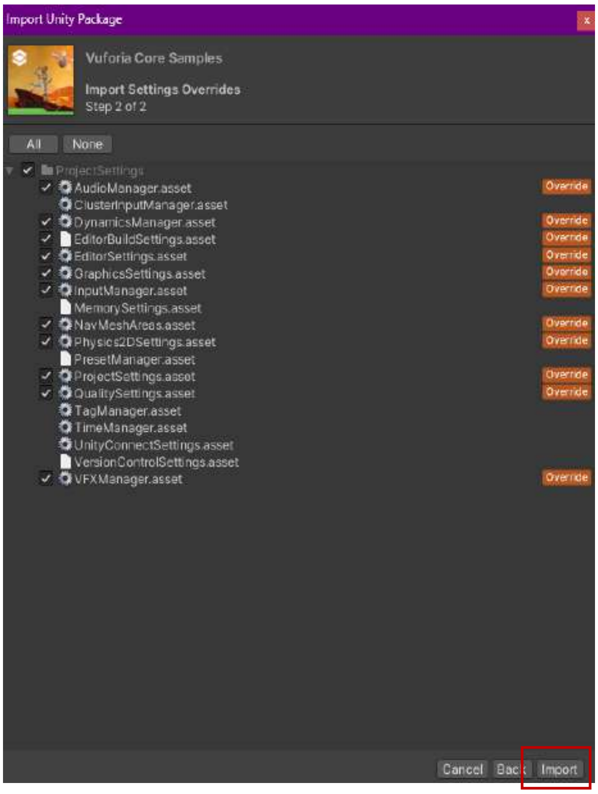


Unity may inform you that there are some updates and ask for confirmation. Press **Install/Upgrade**.

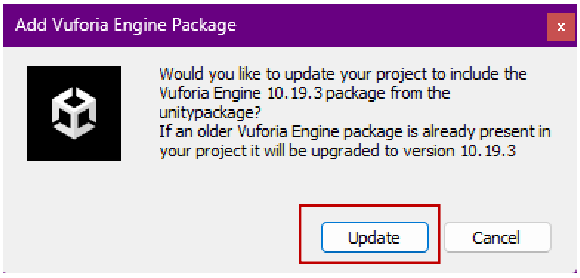


In the window with options for import operations, click **Next** and then click **Import** in the window that opens, which contains many settings.

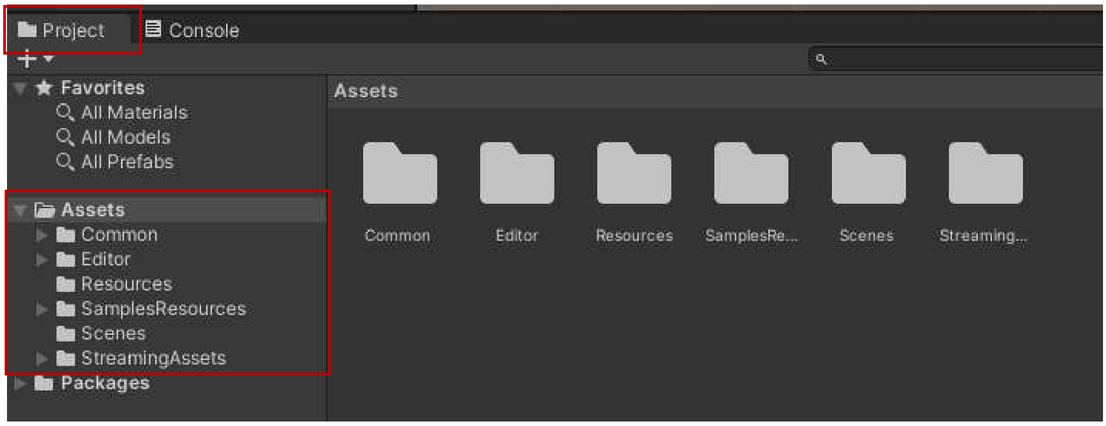




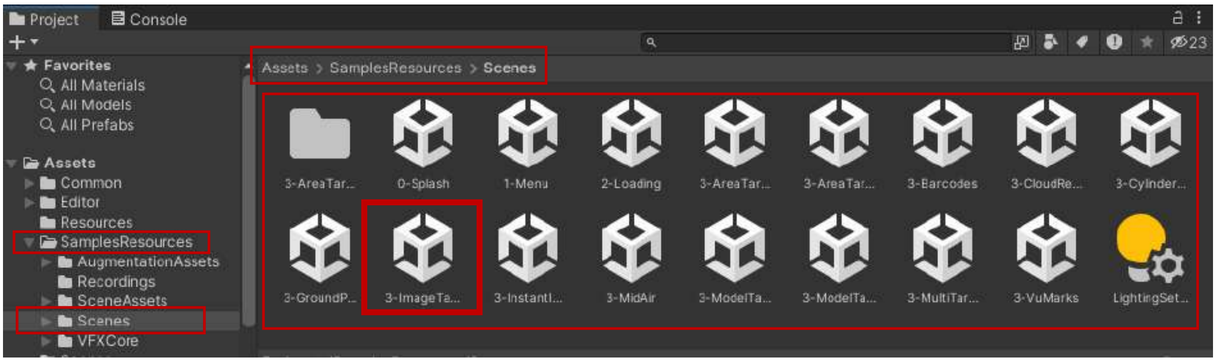
Finally, if Vuforia has an up-to-date package, it will be updated with **Update**. Continue by saying **Update**.



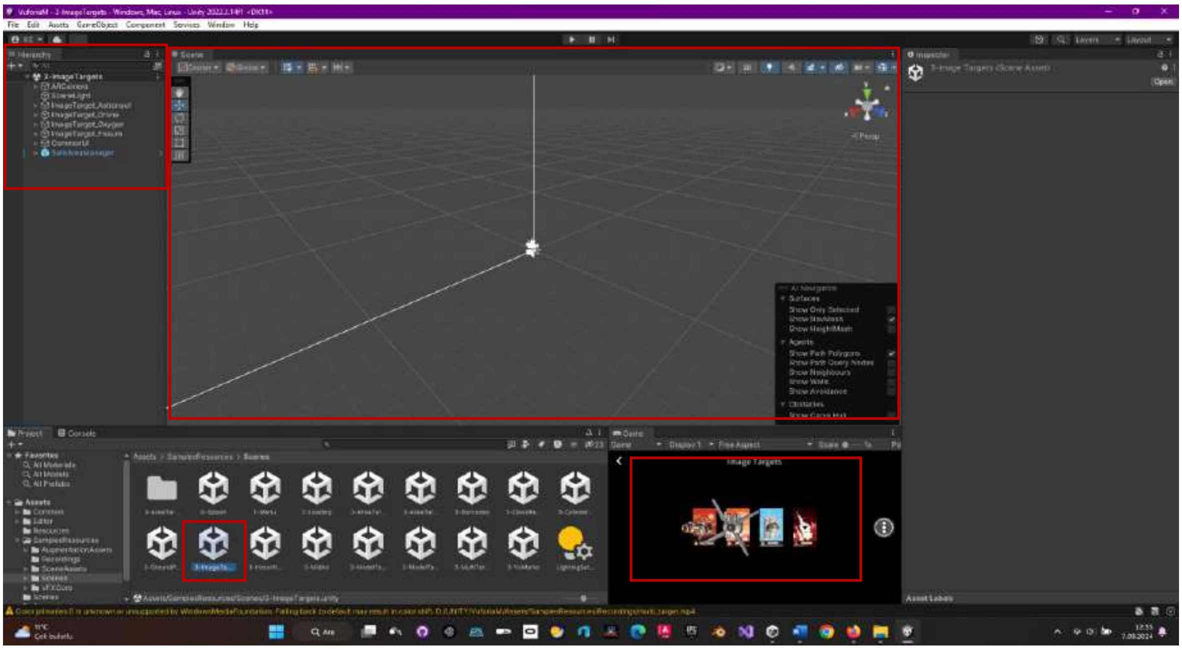
At the end of this process, we can see that new folders have been created under **Project>Assets**.



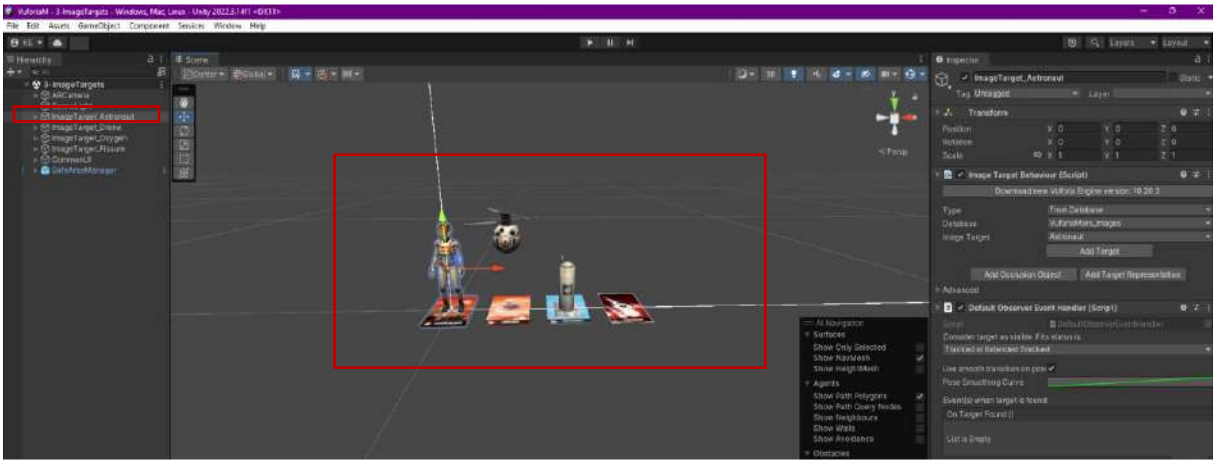
Go to the **Scenes** section under the **SampleResources** folder. Access Vuforia's rich augmented reality scenes. In the scenes, select the **ImageTarget** scene and open this scene with a double click.



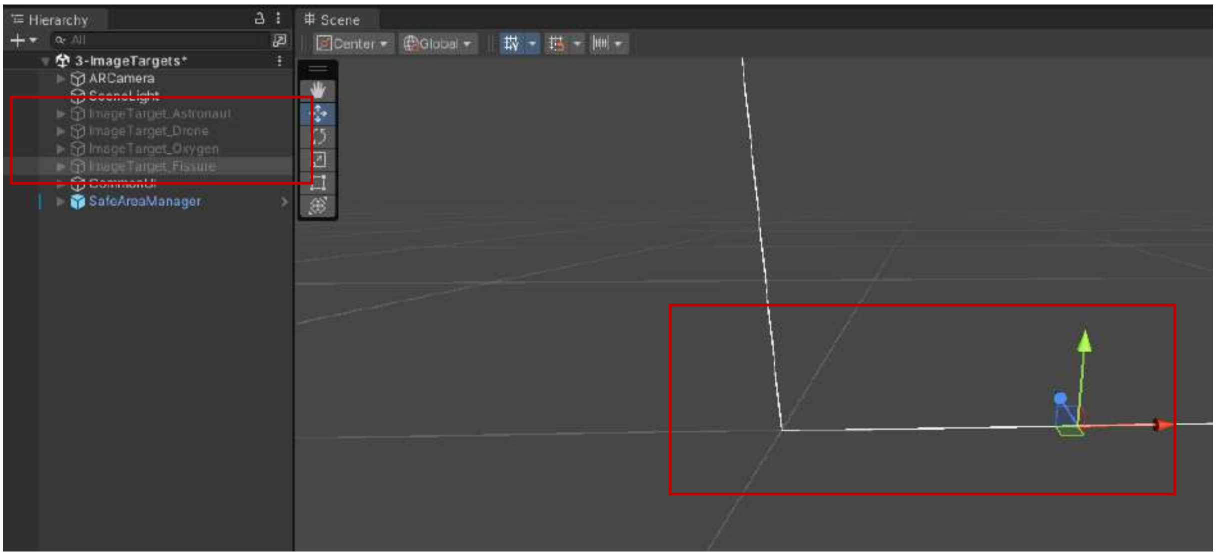
Scene will open.



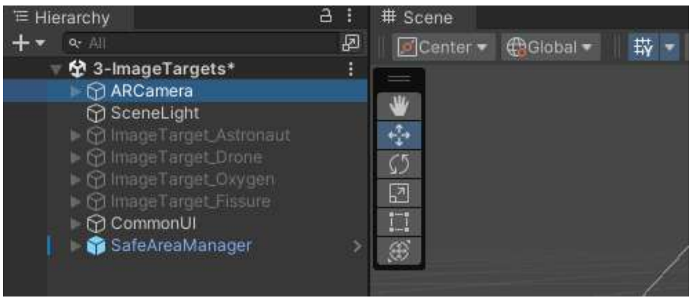
Models can be viewed by double-clicking any **Image Target** in the scene that comes with the **UI (User Interface) Canvas** structure.



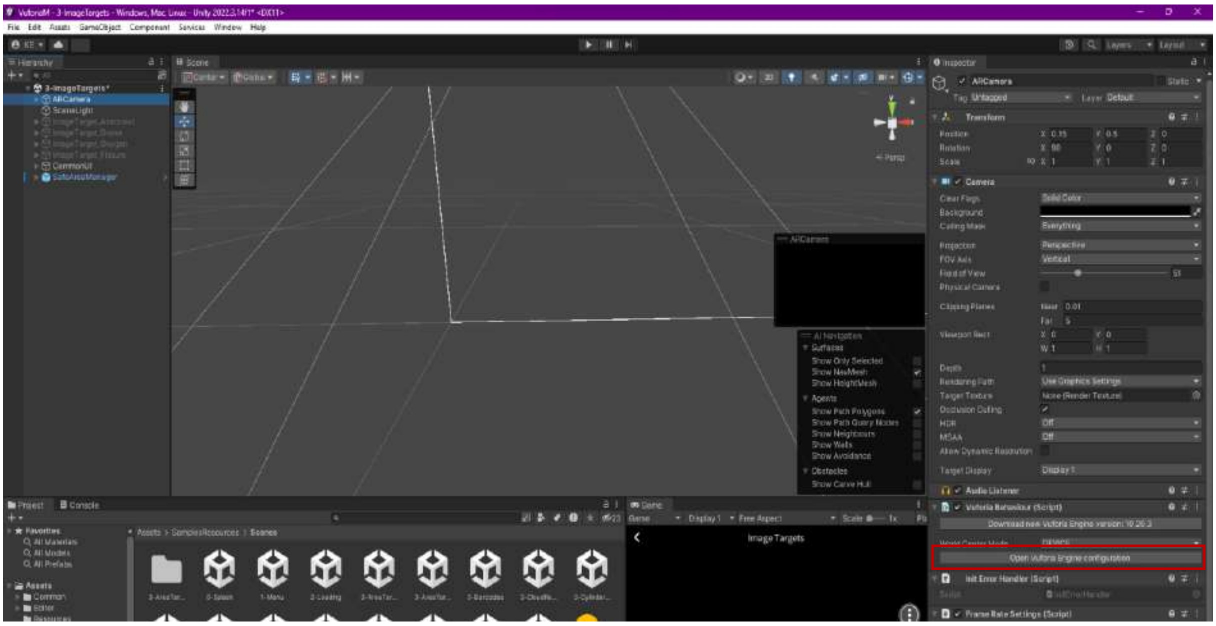
Since we will be creating our own **database**, let's disable all **Image Target** objects and free up the space of objects in the scene.



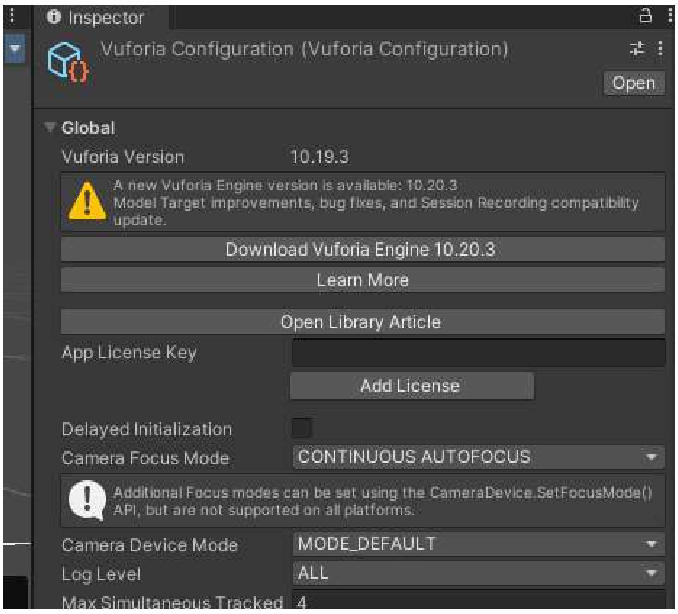
The **ArCamera** is generated already.



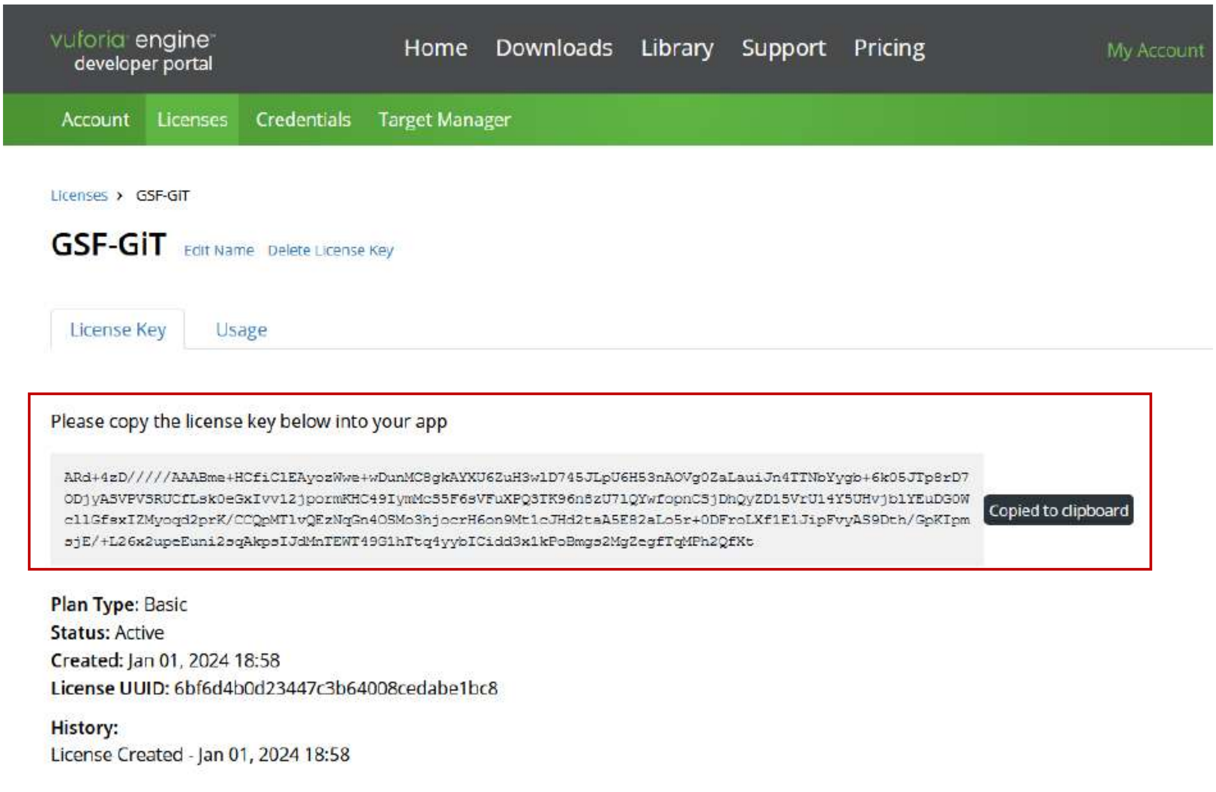
Choose **Open Vuforia Configuration** in the **Inspector**.



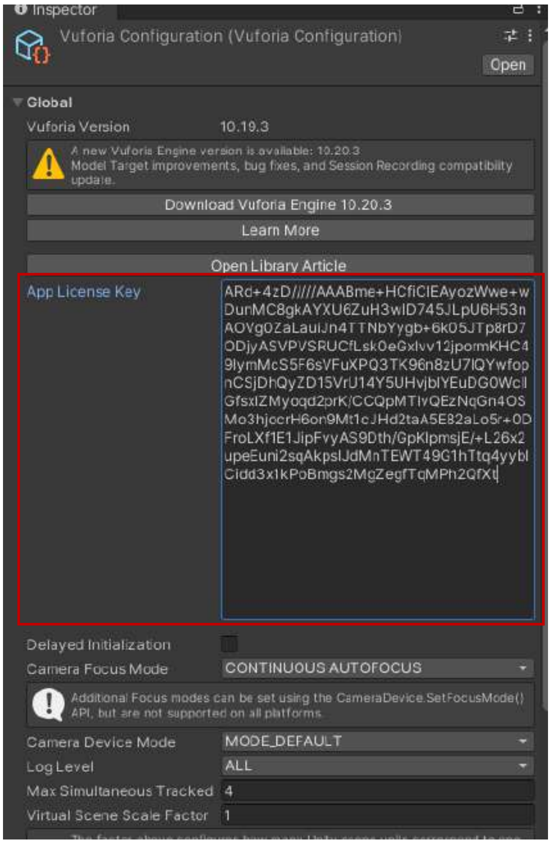
In the incoming menu, go to the window that says **App License Key**.



Remember the **license key** we copied in Vuforia. We memorized it, but if there is other information written on it, copy it again.

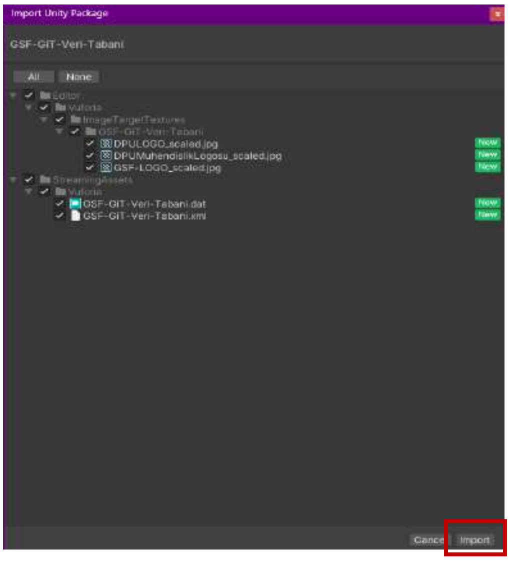


Paste the key into the **App License Key** section in Unity.

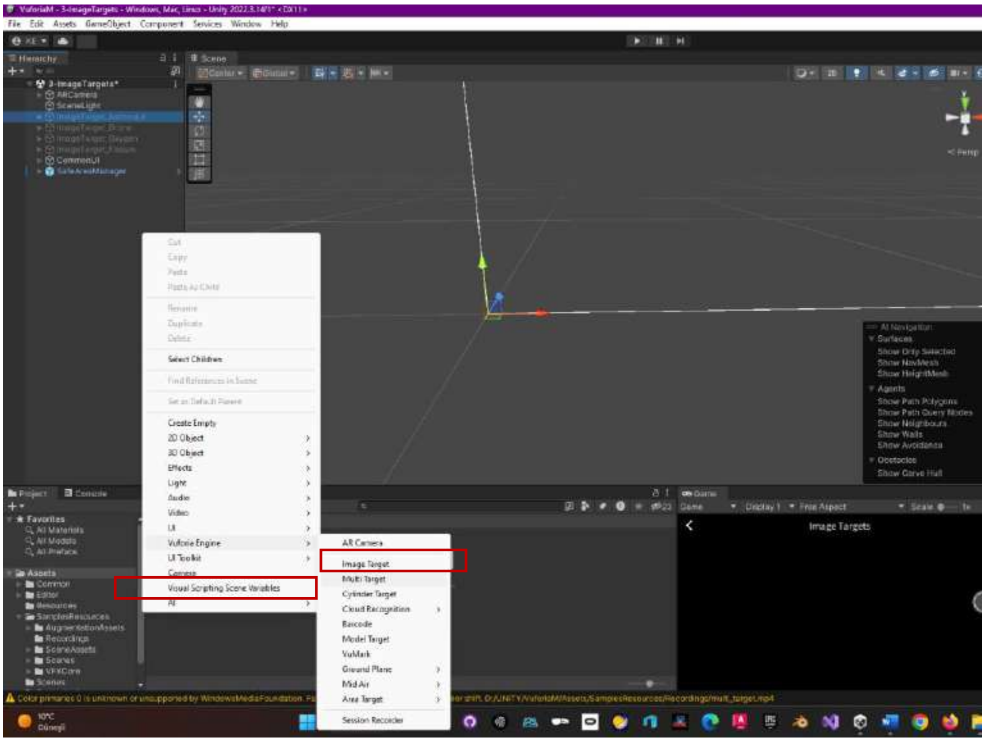


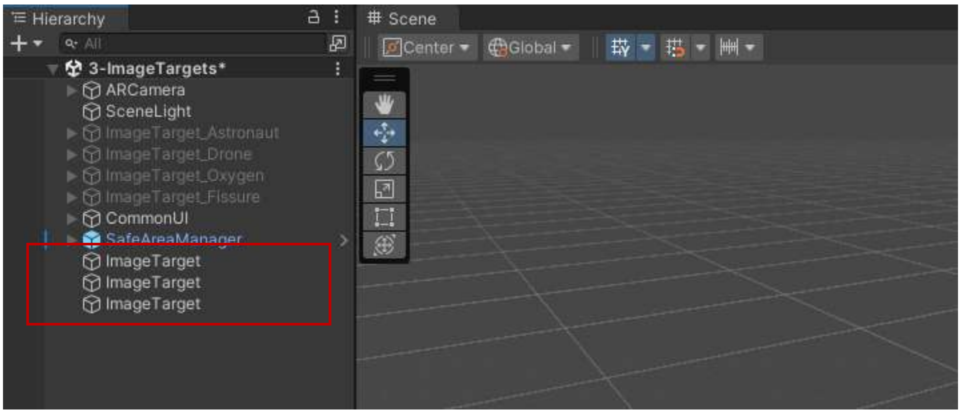
Now, drag the **GSF-GiT-Data-Base.unitypackage** file that we downloaded from **Vuforia** to the Assets section of our project.

A window will open where we can see the package content and our **JPG** files. Click **Import**.

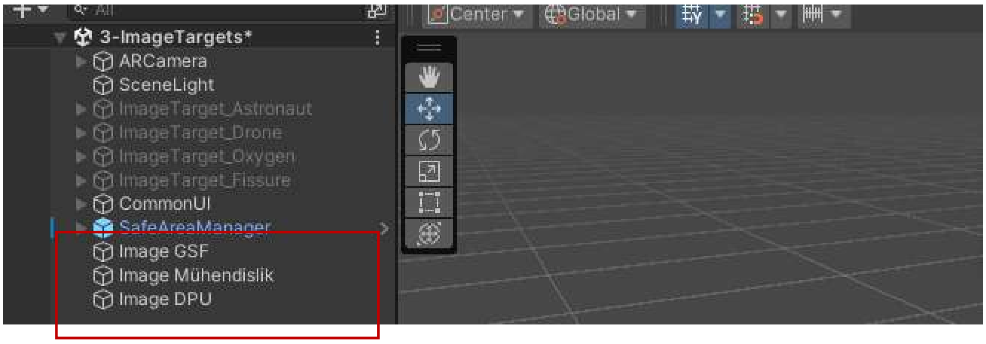


At this stage, we need to create new **Image Target** objects instead of the ones we made passive. Open a window with the right mouse button in the **Hierarchy** section and click the **Vuforia Engine** option. An object named **Image Target** will be added to our object list. Since we will be working with three targets (multiple), repeat this process twice more and increase the number of **Image Targets** to three.

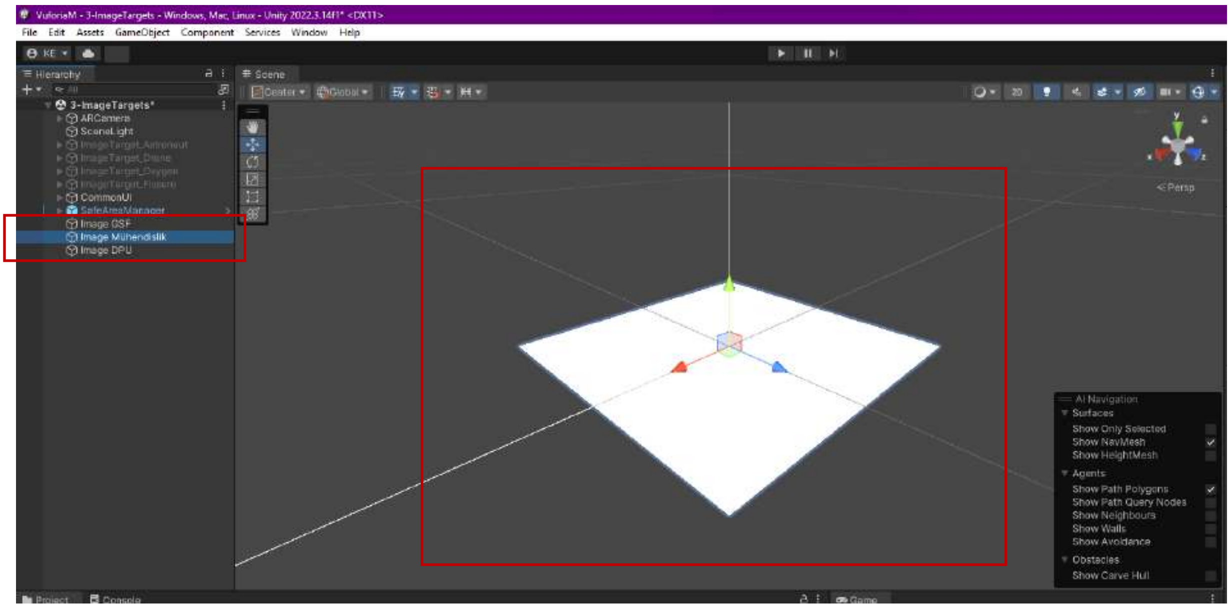




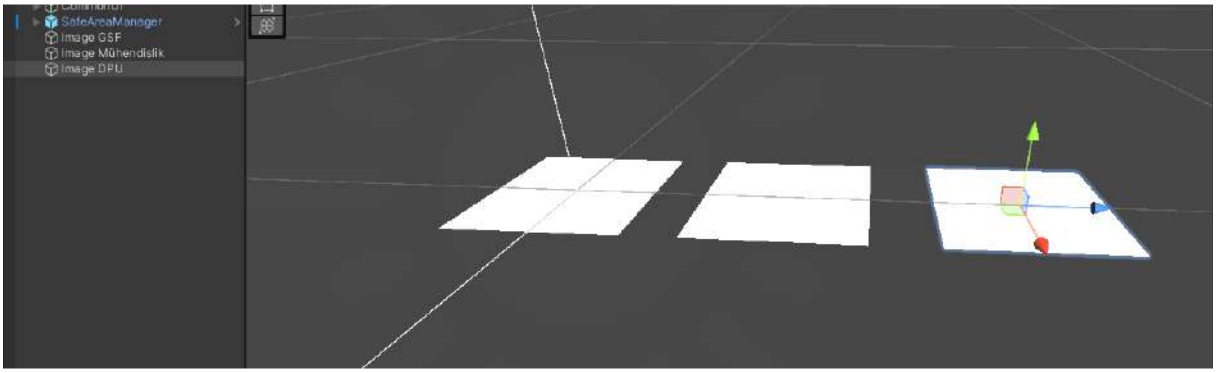
We can name each one to avoid confusion.



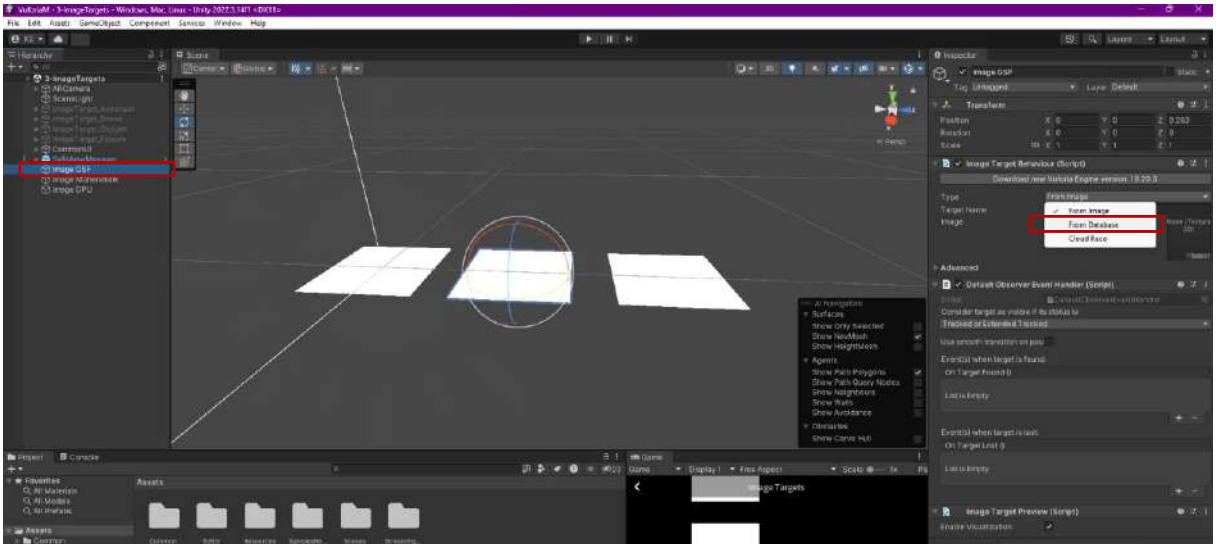
Double-clicking on one of these target objects in the **Hierarchy** will bring the scene into focus. Clicking on the other two will reveal that all three are on top of each other.



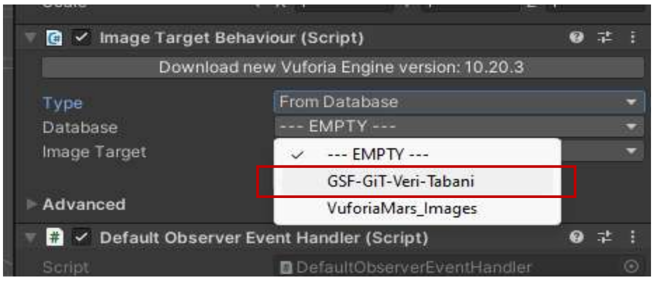
So, reposition the plane image objects so that they are next to each other in the scene.



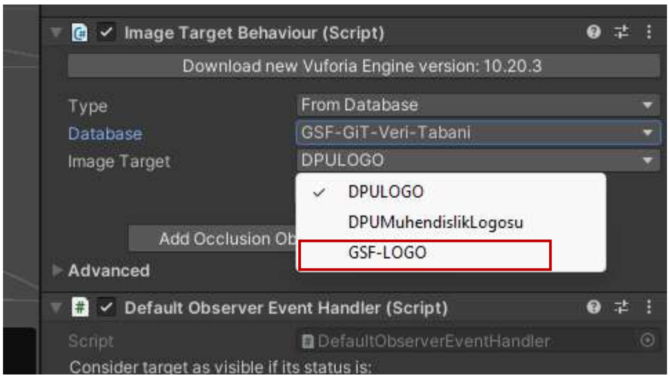
Now, it's time to match these white cards with the images in our database. To do this, first select the **Image GSF** object. In the **Inspector** section, select **From Database** under the **Type** option.



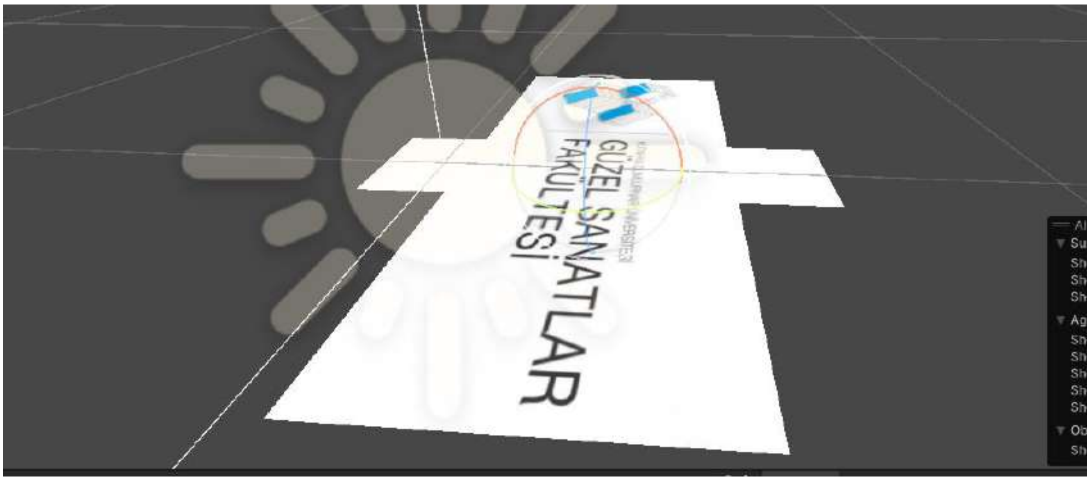
Here, select the database we created in the **Database** section.



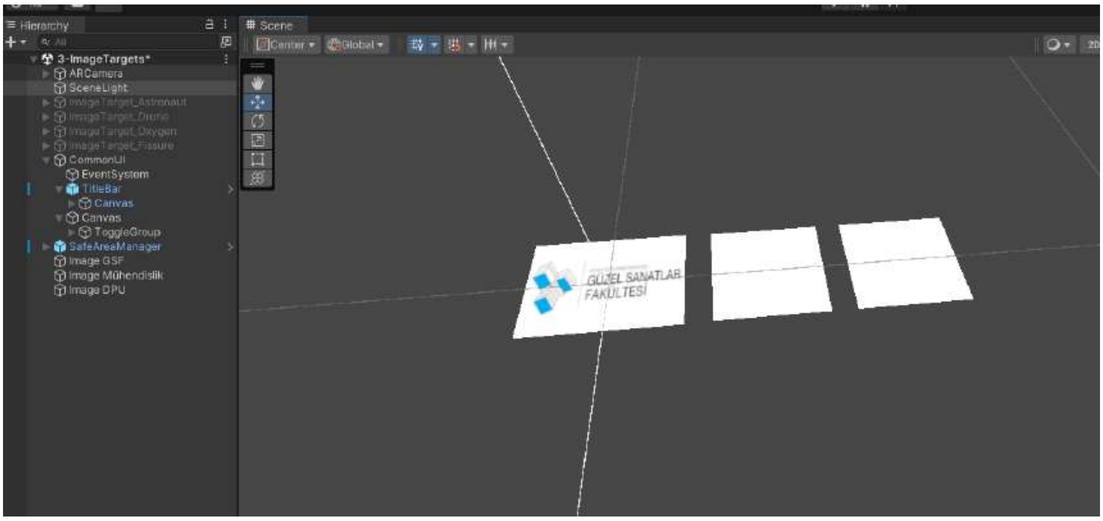
Then, select the **GSF-LOGO** file as the **Image Target** in this database. After the selection, the match is made automatically without any further action.



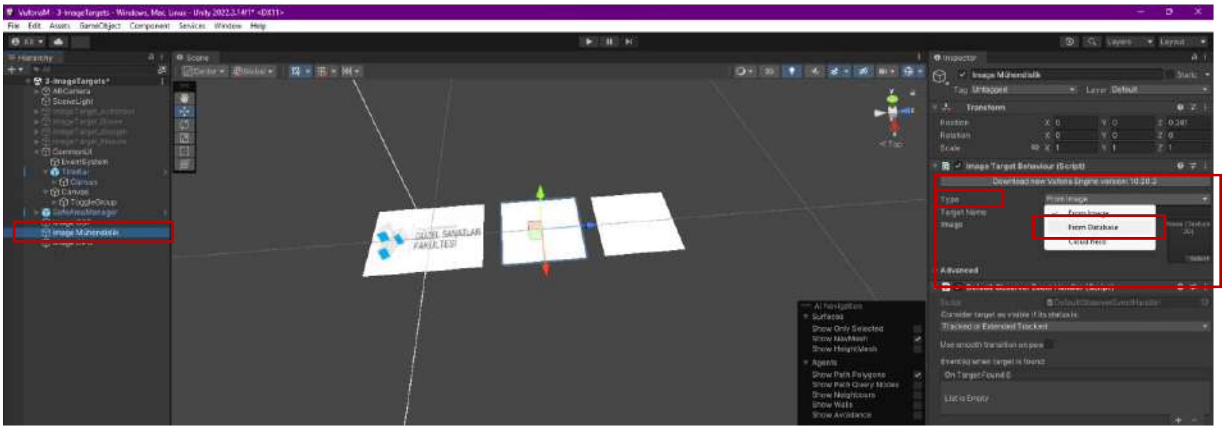
The **GSF-LOGO** shape is matched with the **Image Target** object in the scene.

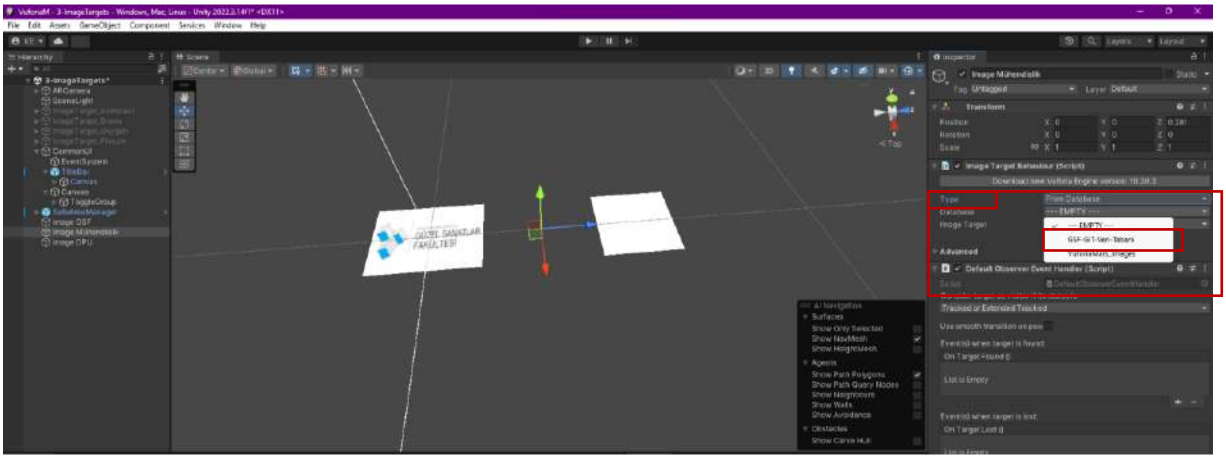


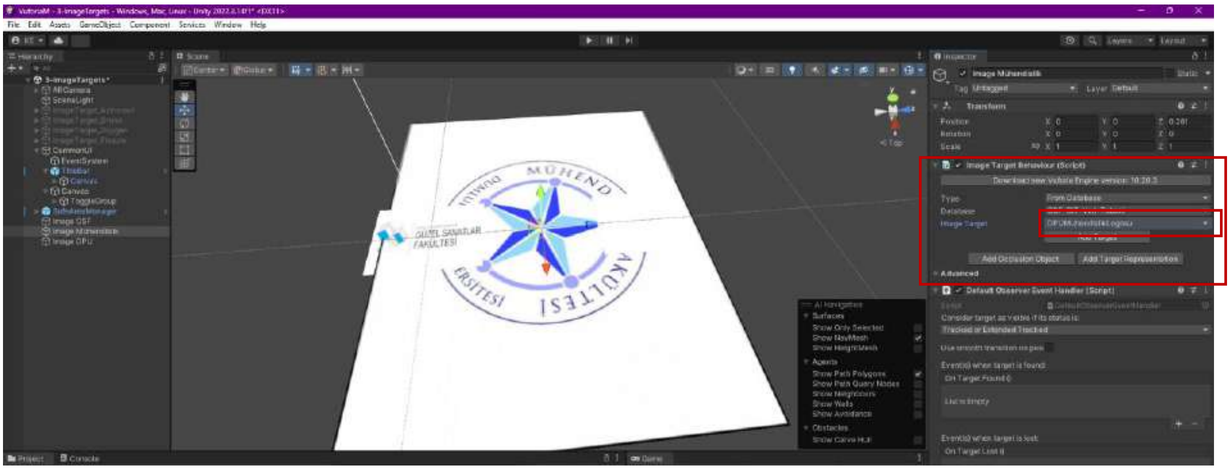
However, for the shape to be compatible with the target, the image must be selected and resized.



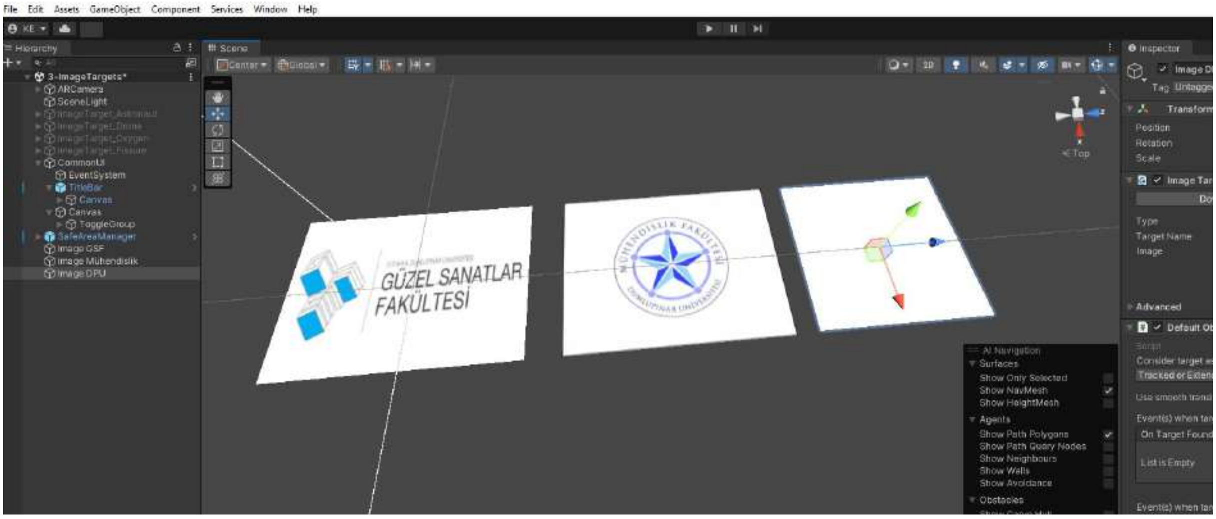
Now, select the Image Target object and repeat the same process for it.



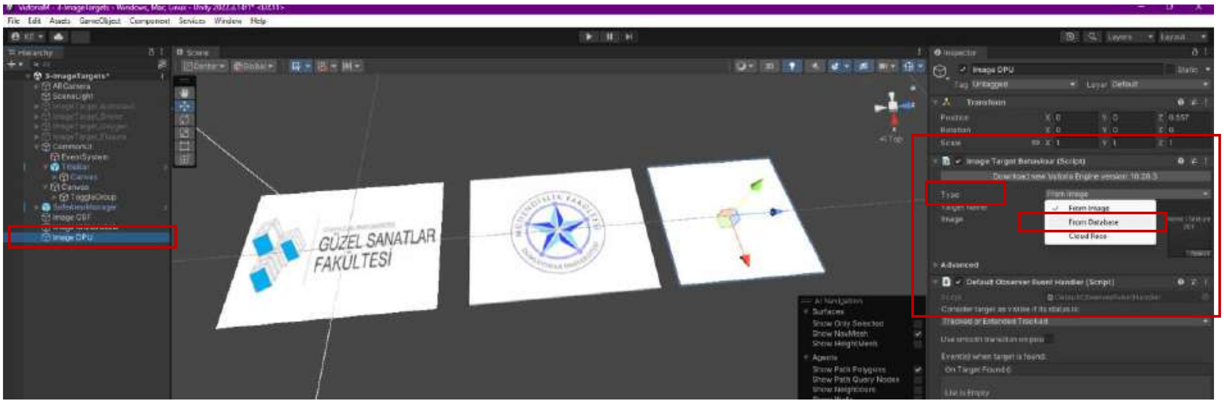


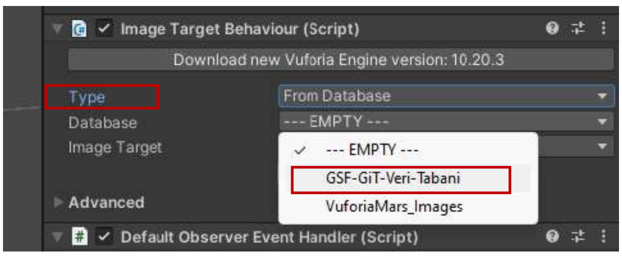


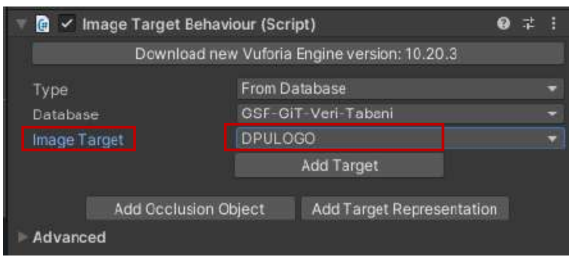
Adjust the place and position of the image.



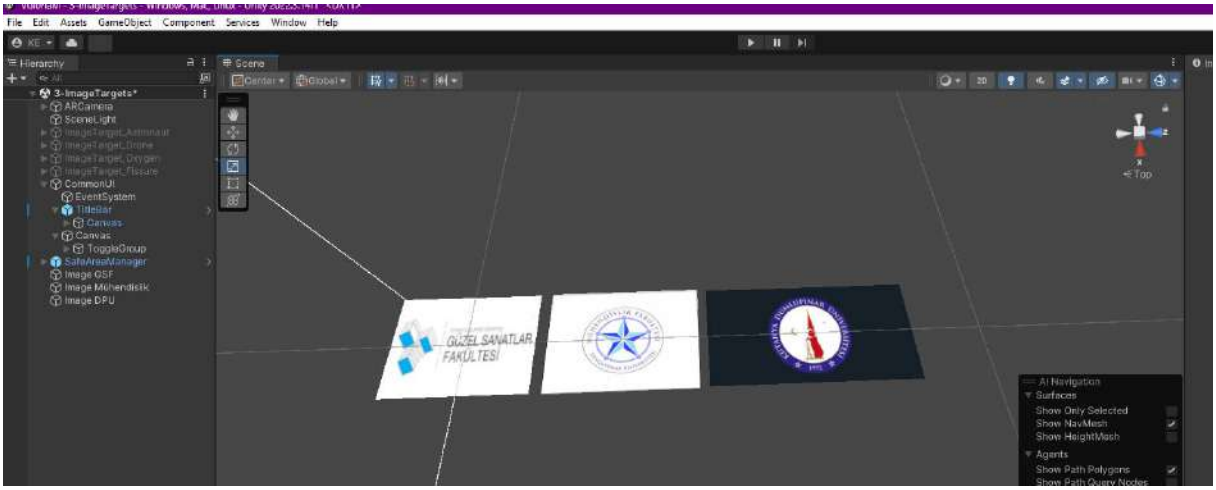
Finally, match the third image to the Image DPU target.





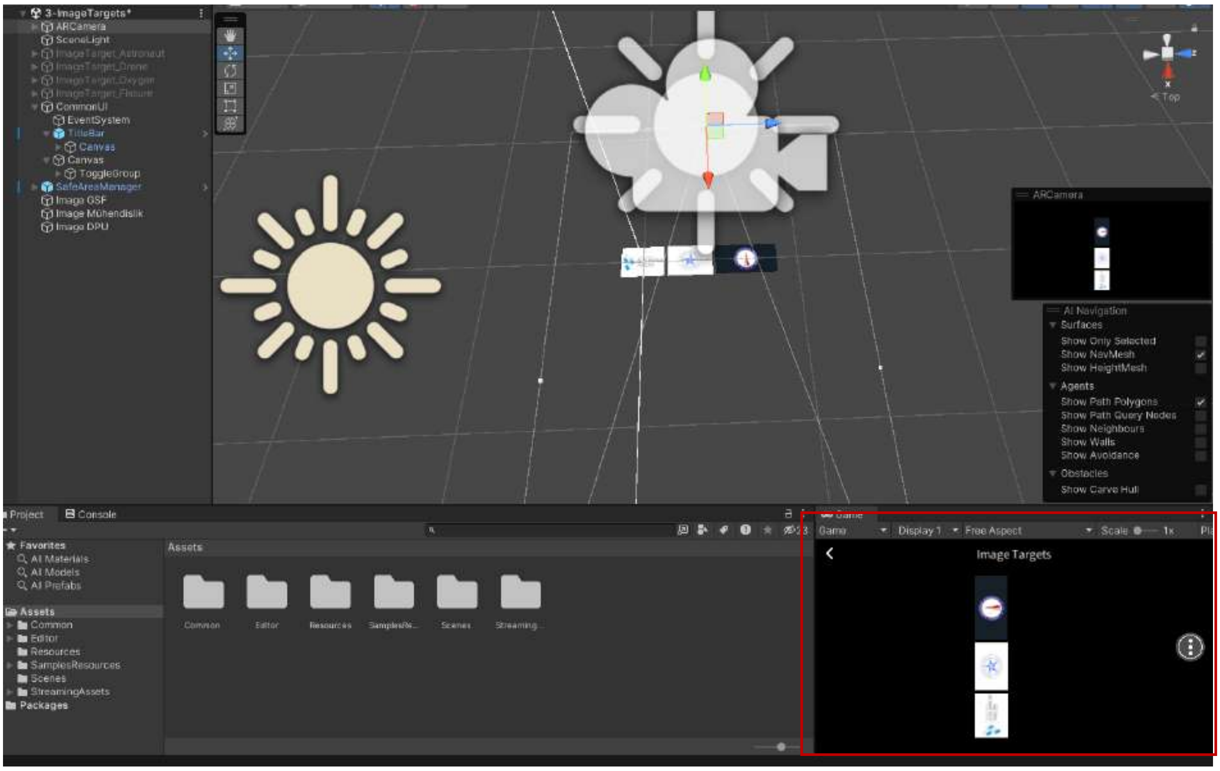


Adjust the place and position of the image.

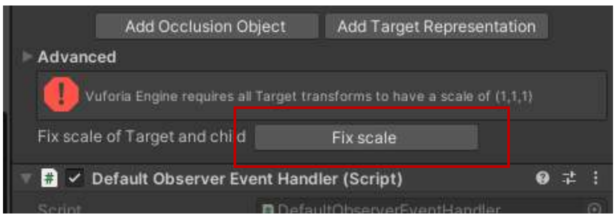


A dark background photo was used here for the DPU logo, but a white background image would be more functional.

Three images can be placed in the frame by changing the **ARCamera** position.



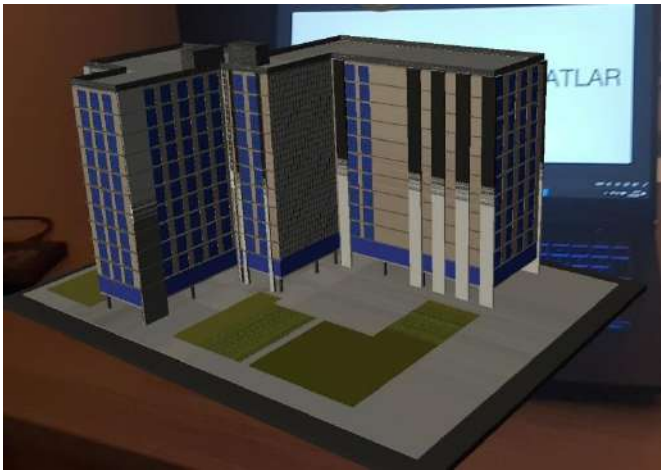
When **Image Target** objects are selected, if a scale warning appears in the **Advanced** section, click **Fix Scale** and then confirm with **Ok**.



At this stage, the kind of operation that will be done on the image/photo displayed with the augmented reality camera should be determined.

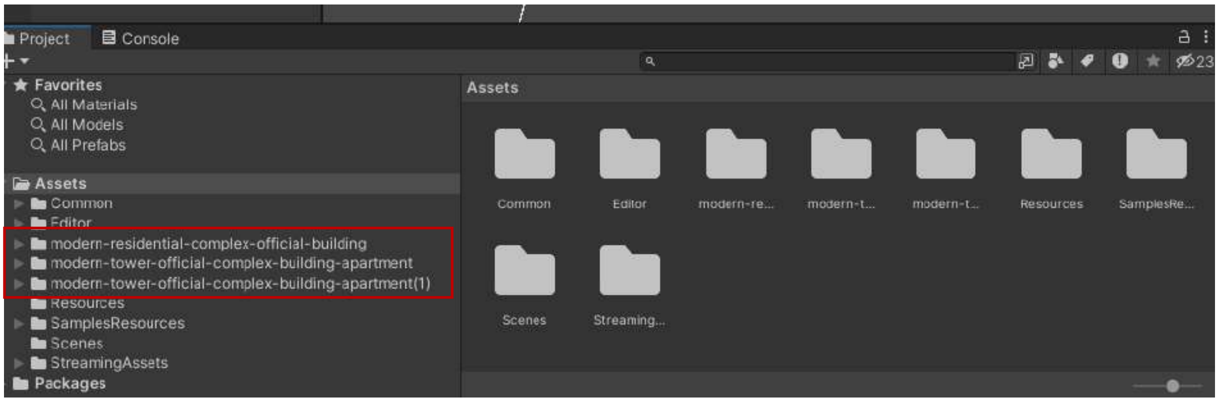
As you may recall, in our single **Image Target** application, we used the Modern Tower Official Complex Building Apartment model from the **Sketchfab** site.

We matched it with GSF-LOGO, and when we brought the phone/tablet to this picture, the building opened in 3D.

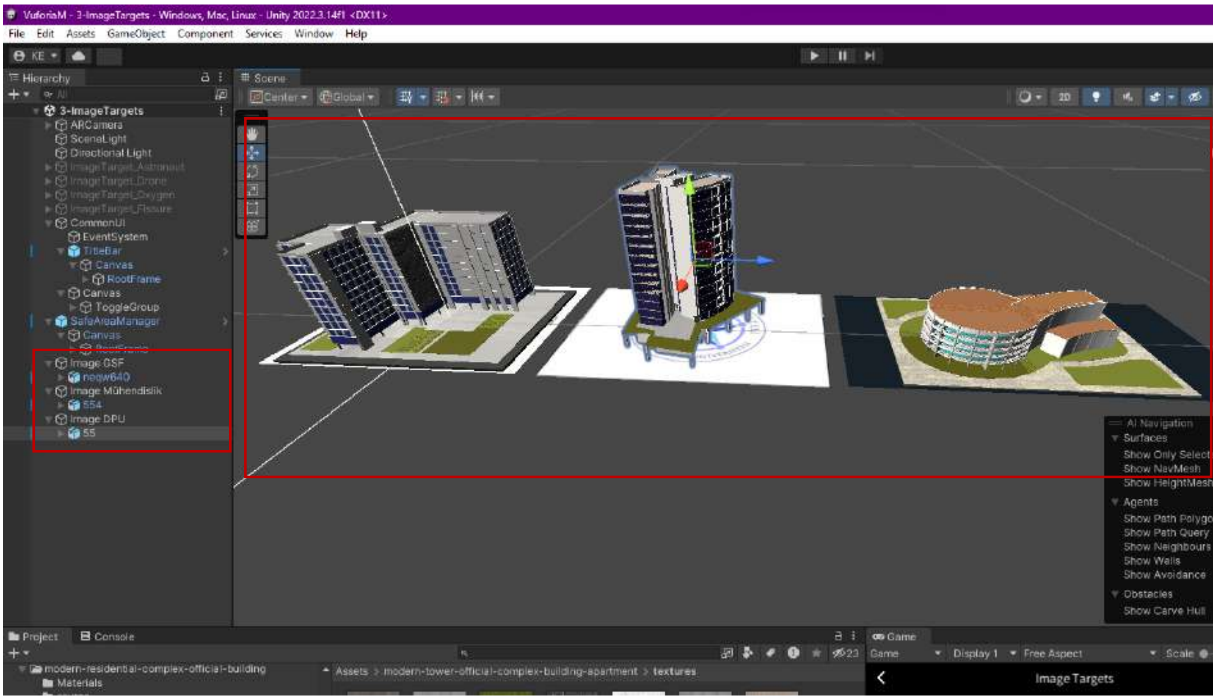


Now, assign the **Image Target** images of the three in a similar way. For this purpose, we can use **Unity Asset Store** or **Sketchfab**.

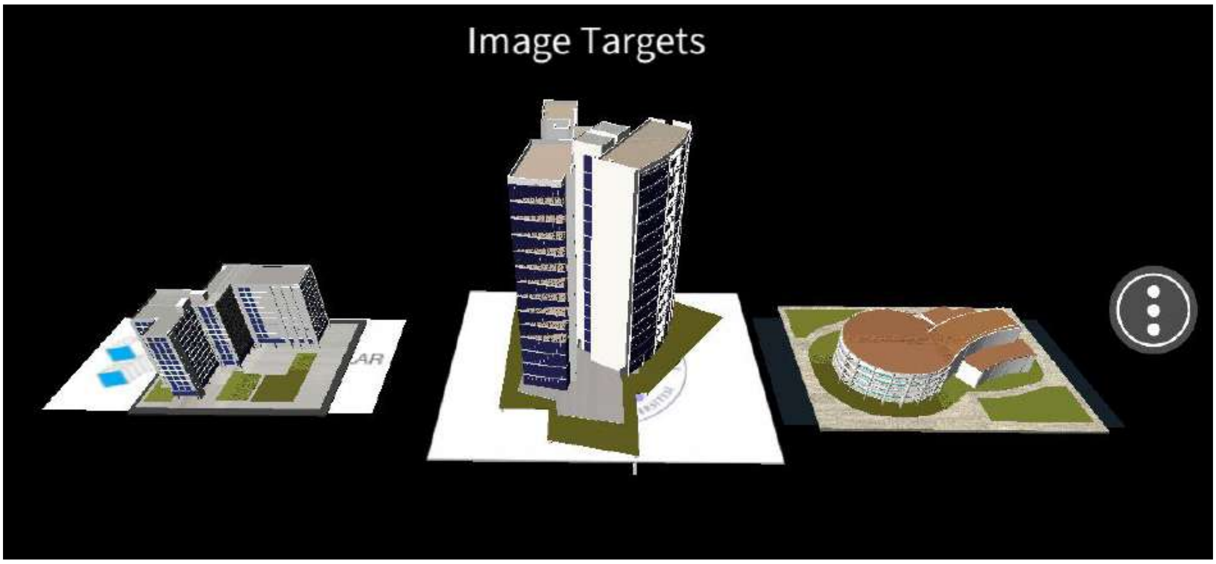
In the multi-target shape application, three building models were downloaded from **Sketchfab**, whose names are shown in the figure.



The buildings were brought into the scene one by one, scaled, rotated and moved to position them on the cards. They were also dragged into and connected to the **Image Targets** in the **Archive**.

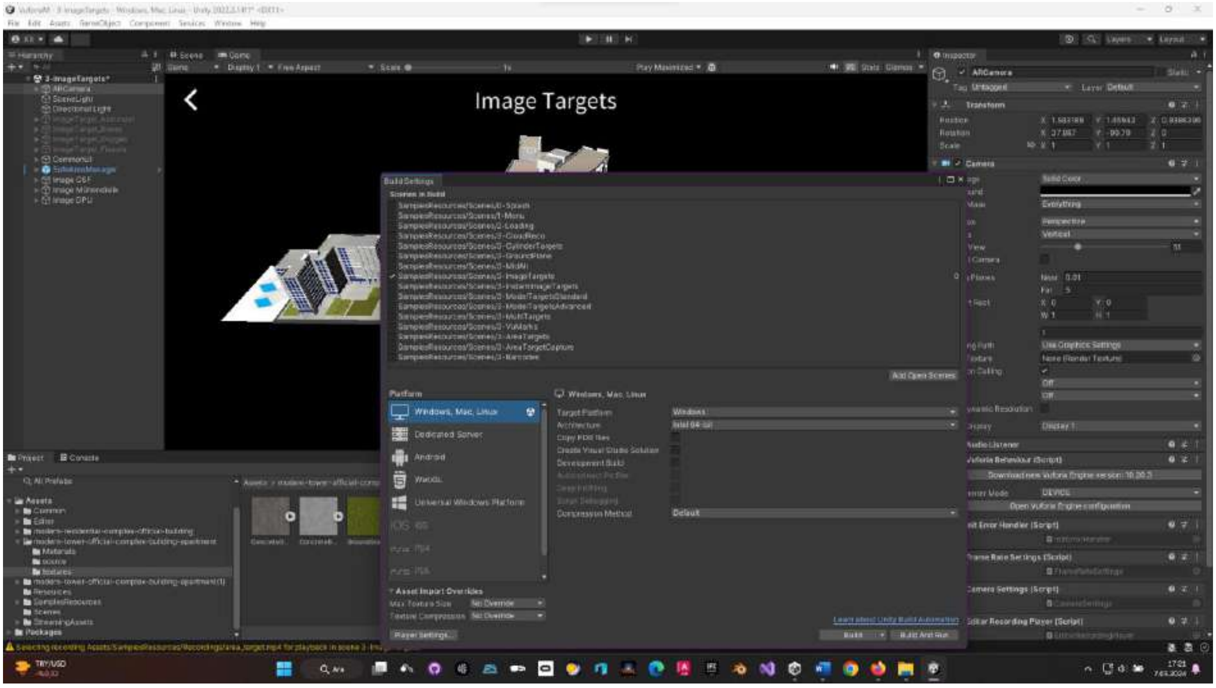


The image on the game window is given below.



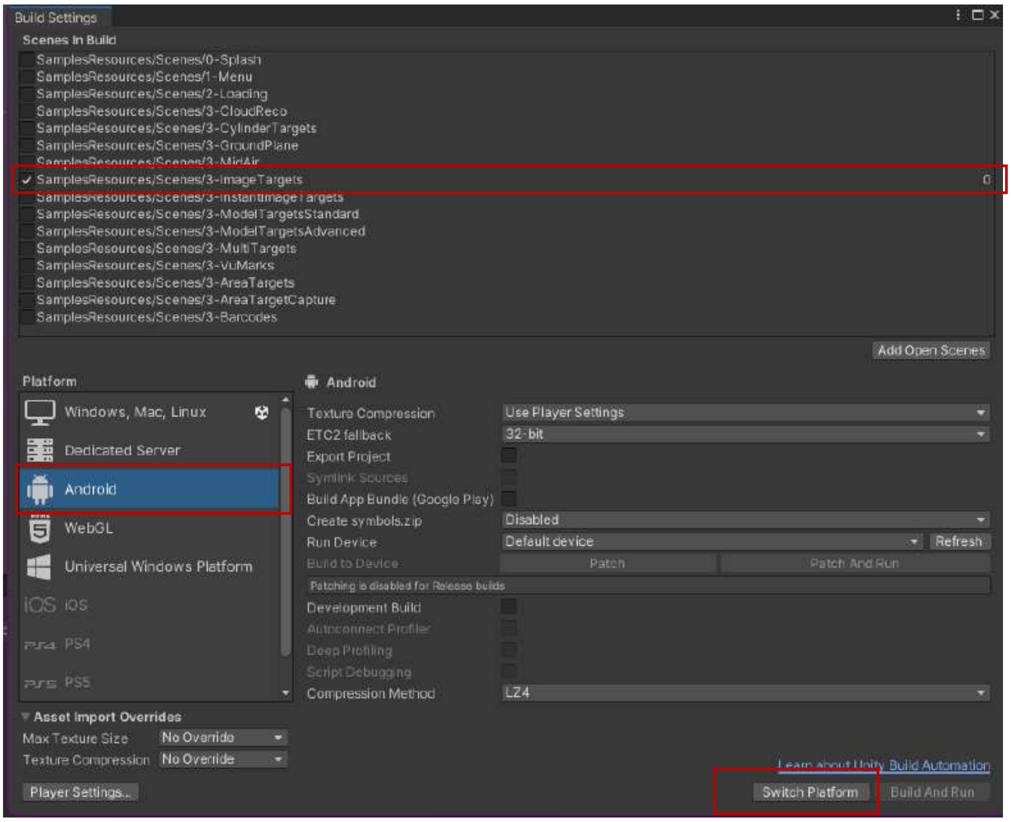
11.6.Deployment into Mobile Devices

Let's repeat the steps in the **Single Image Target** application here. This will open the **Build Settings** window. There are many scenes listed in the **Scenes in Build** section.

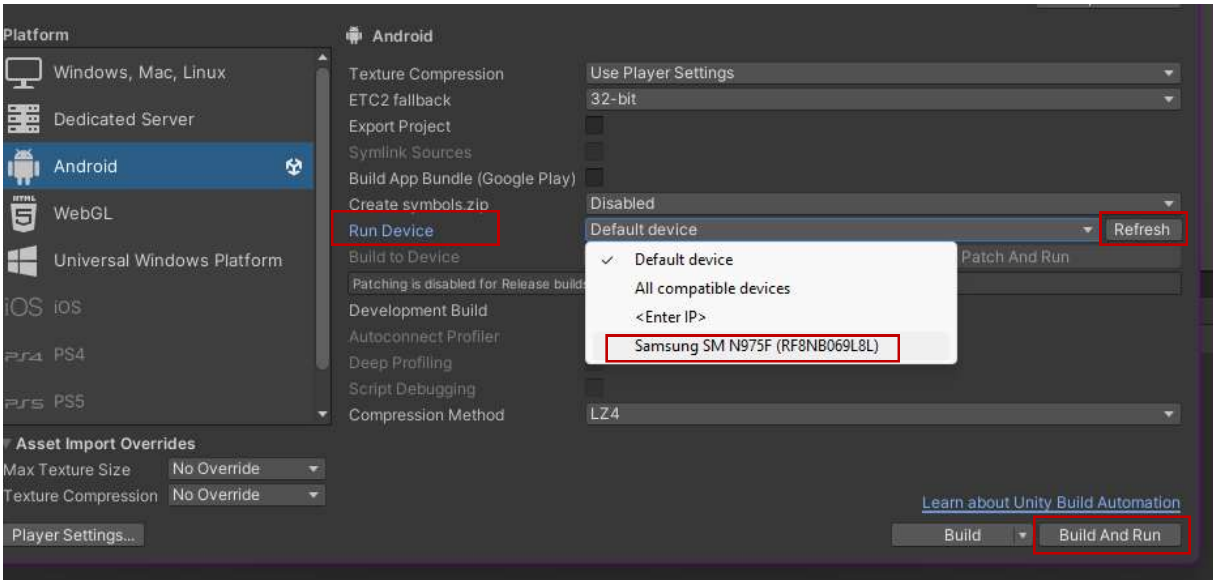


However, since our project is about **Image Target**, the boxes of the ones other than **Image Targets** are checked and excluded from the process.

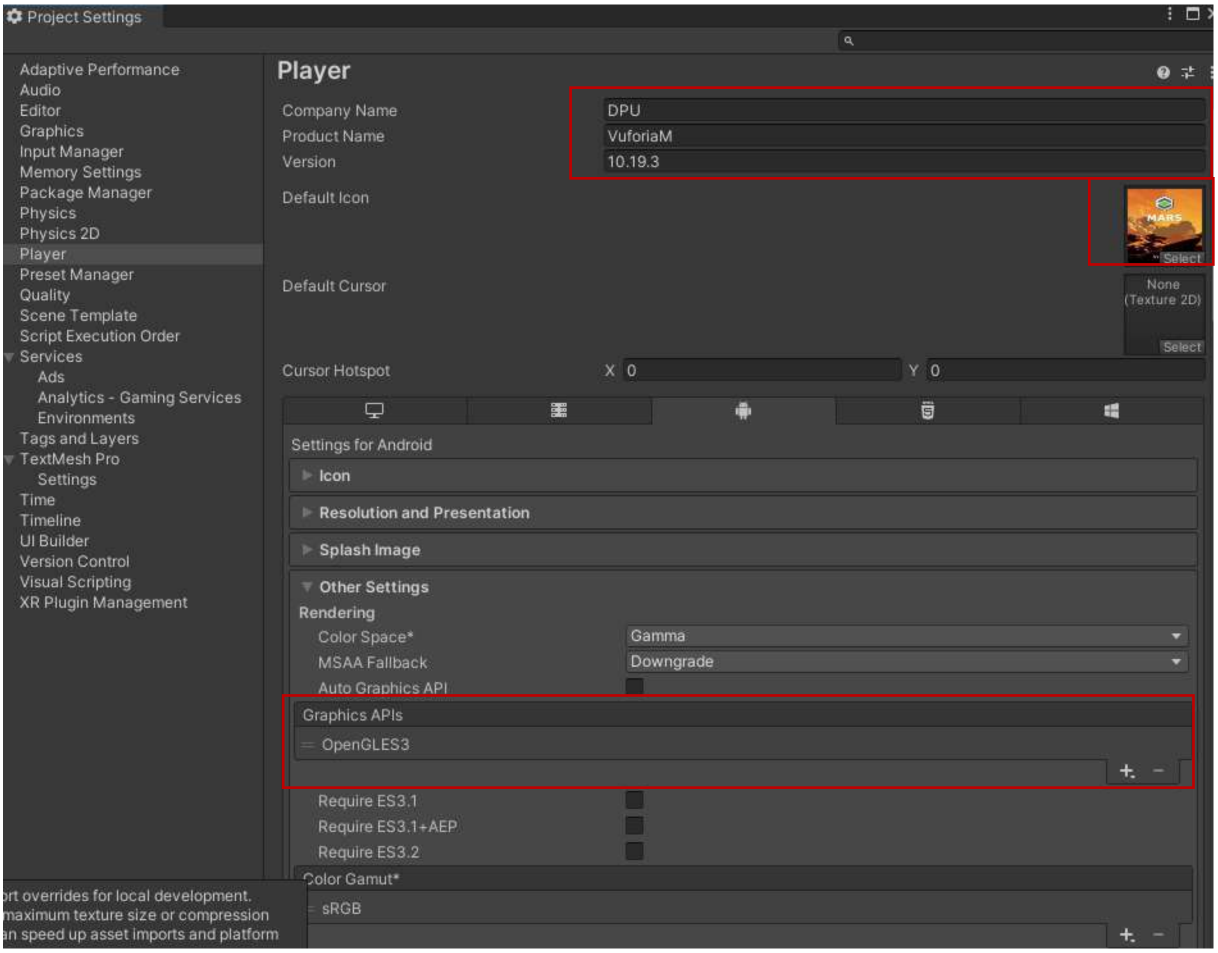
Also, since this will be a mobile application, the **Switch Platform** process is done by selecting the **Android** platform.



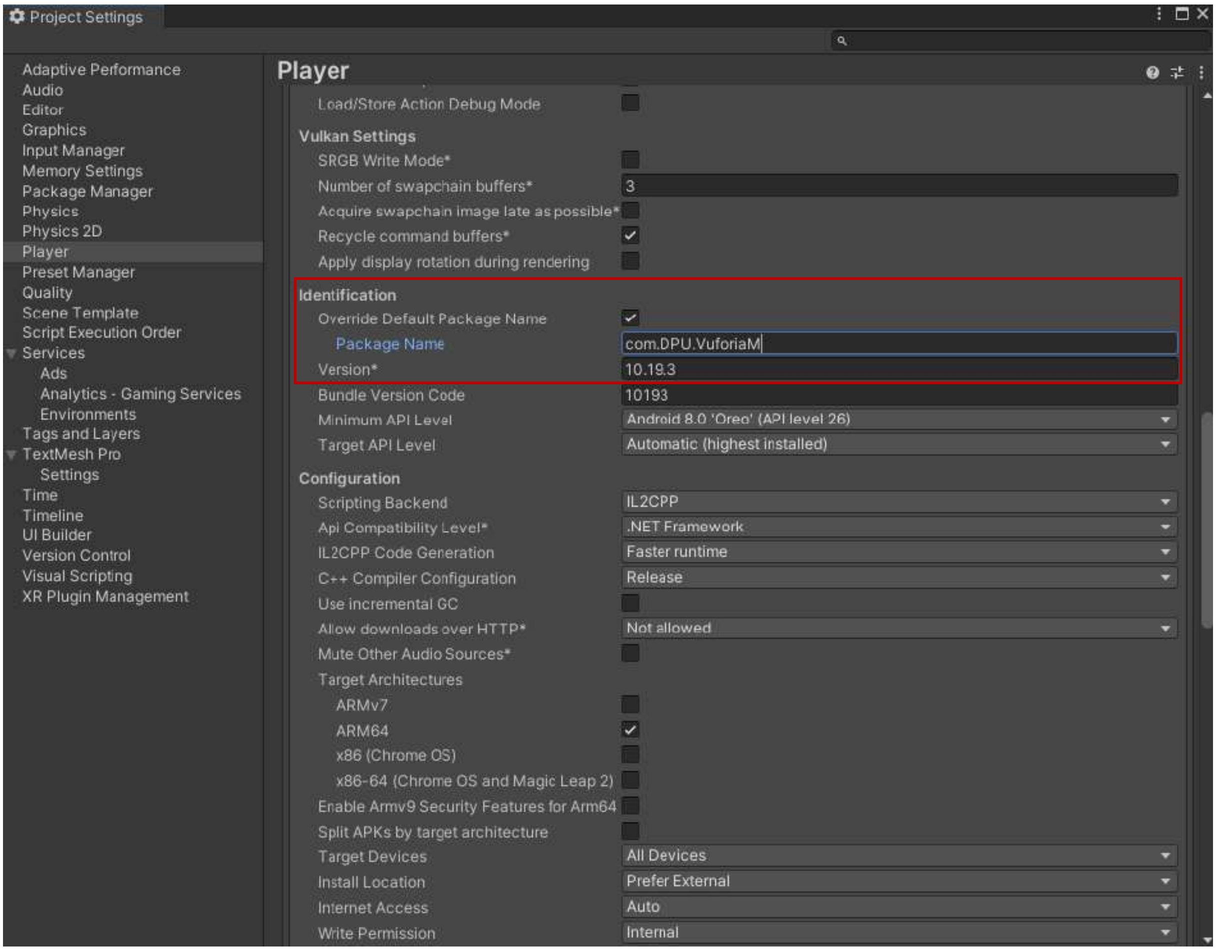
Let's connect our phone with developer features to the computer with a cable and **Refresh** the **Run Device** section.



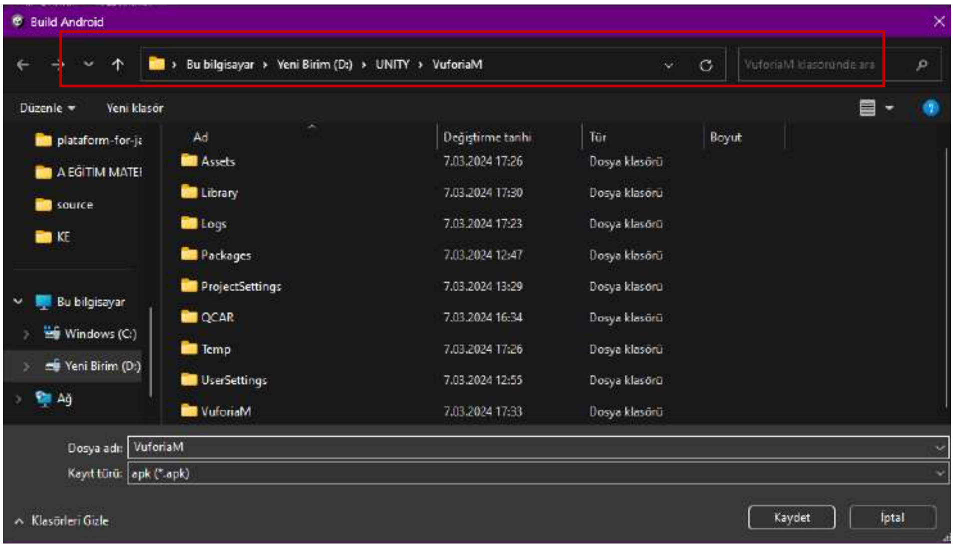
Go to the **Player Settings** section and specify the **Company Name** and application logo. In the **Graphics API** section, <u>delete</u> the **Vulkan** and **OpenGLE2** options other than **OpenGLES3** with the **"-"** key.

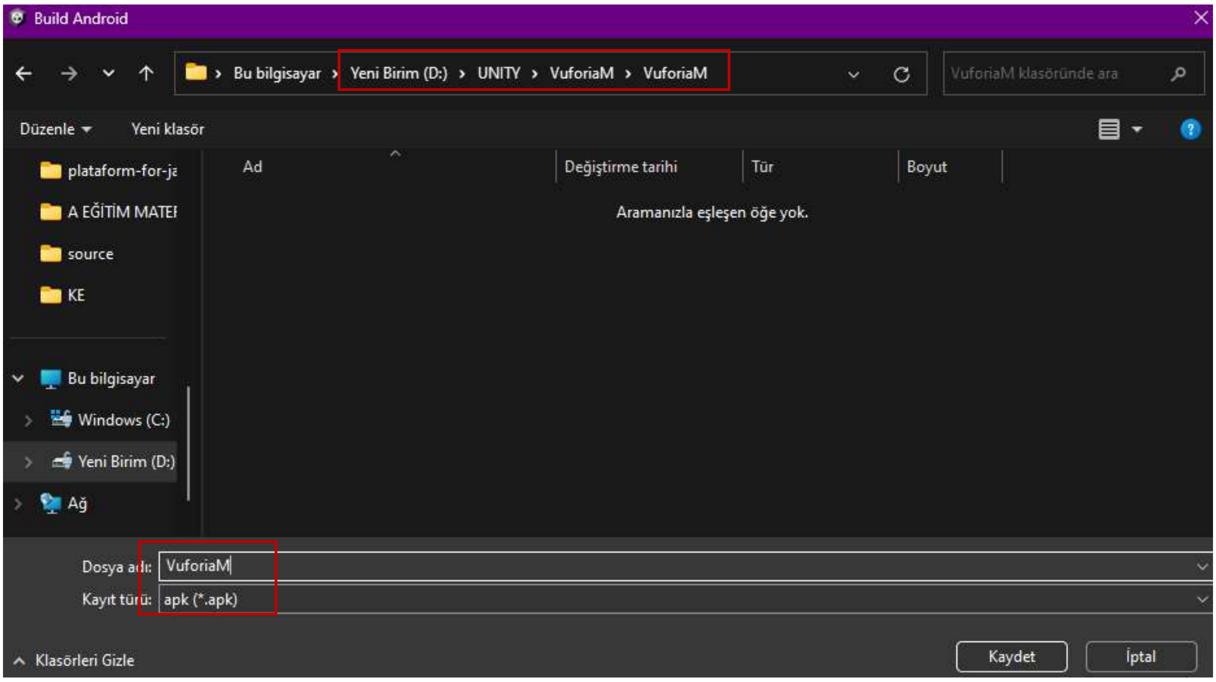


In the **Identification>Package Name** section, change the package name to match the definition above; **com.DPU.VuforiaM**

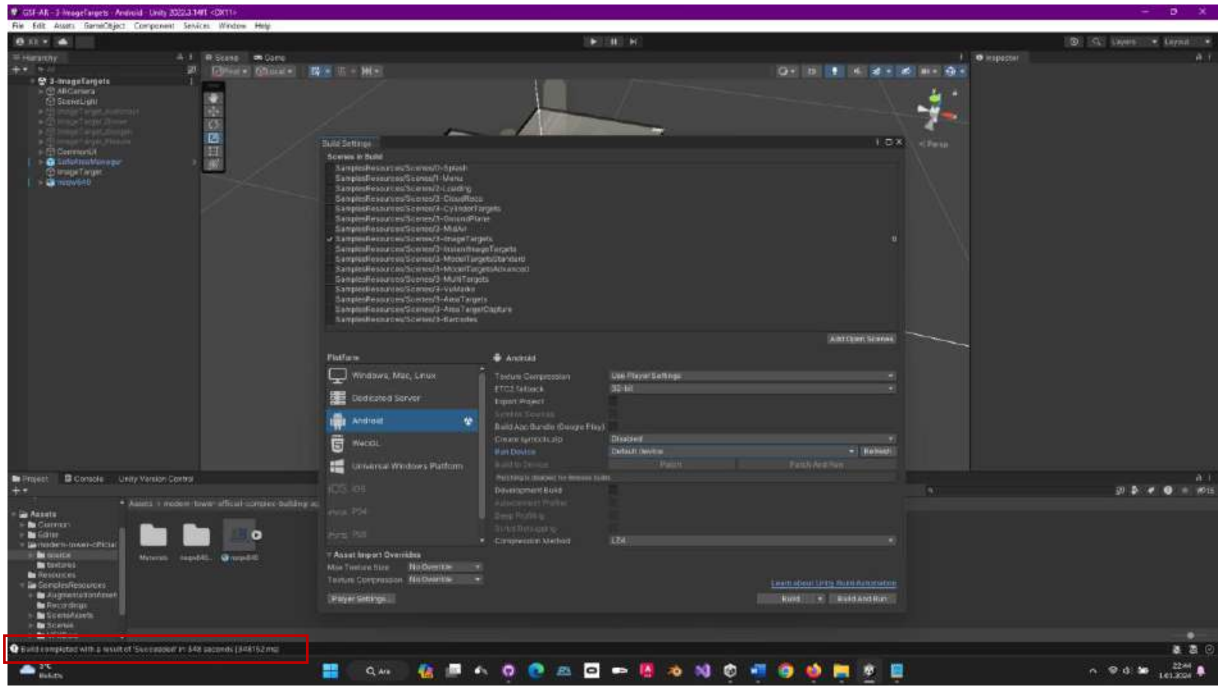


When we select the **Build and Run** section, in the window that opens, we create a new folder named **VuforiaM** with the right click of our mouse to register the application, and we specify the name of our **APK** file as **VuforiaM.apk**.

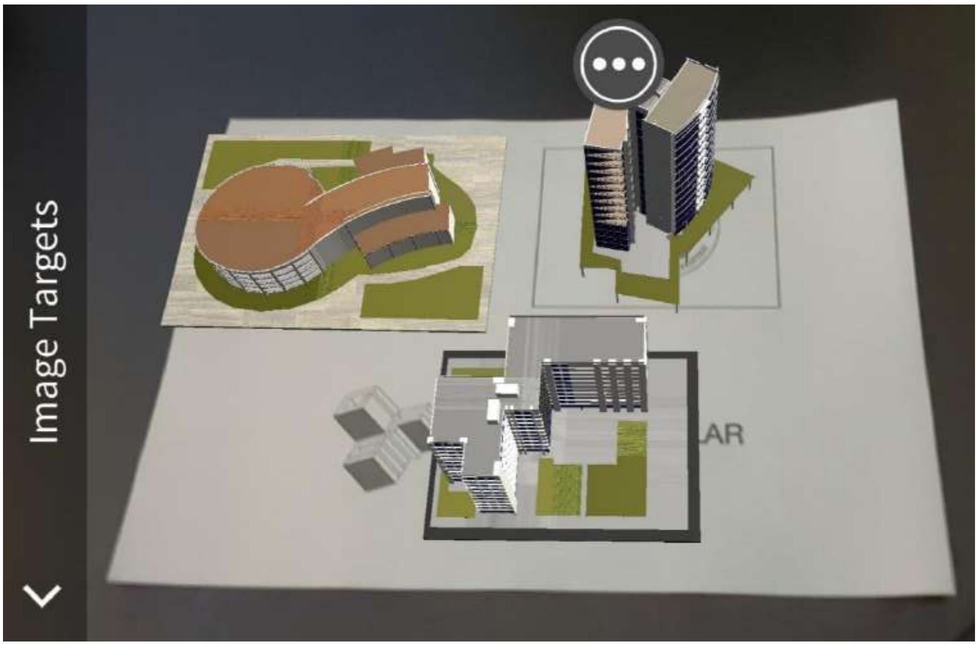




The deployment process starts with the **save** button. It will take a few minutes to complete. Finally, it returns to our stage, stating that it was successful.



When the application was opened on the mobile device, it was tested on an A4 paper with three logos. As a result, the buildings matched with the three images were displayed as an augmented reality application.



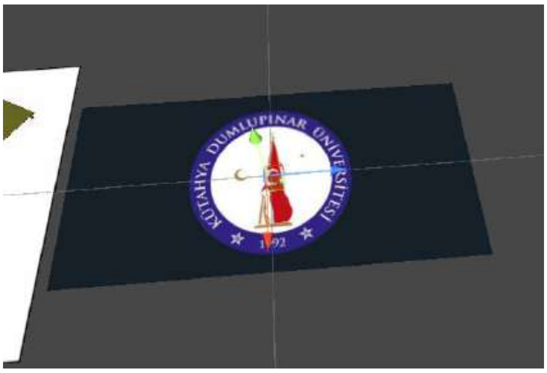
Wherever the app sees three logos, whether individually or in groups, it will display the building model matched on top of the logos in augmented reality format.

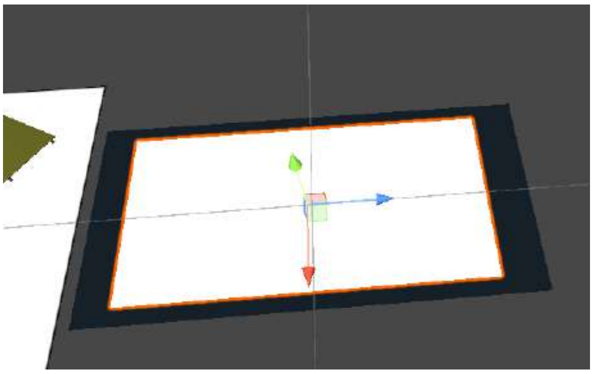
Logos will be displayed individually or in groups on wall posters, computer screens, business cards or documents.

The size of the matched models and their angles relative to the camera can be adjusted by the developer.

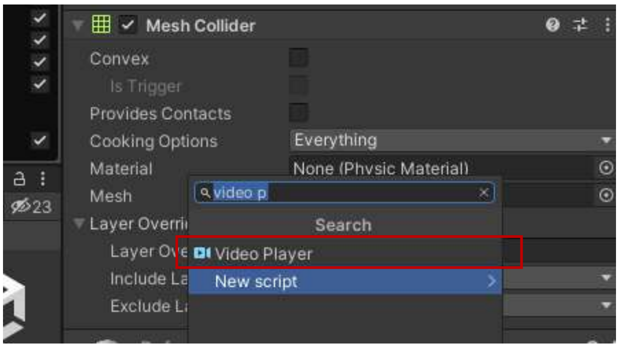
11.7.Augmented Reality with Video Player

In this application, let's place a **Plane** instead of a 3D model on one of the cards. Here, the **DPU logo** is used. Adjust the size and position of the **Plane** so that it is on top of the logo.

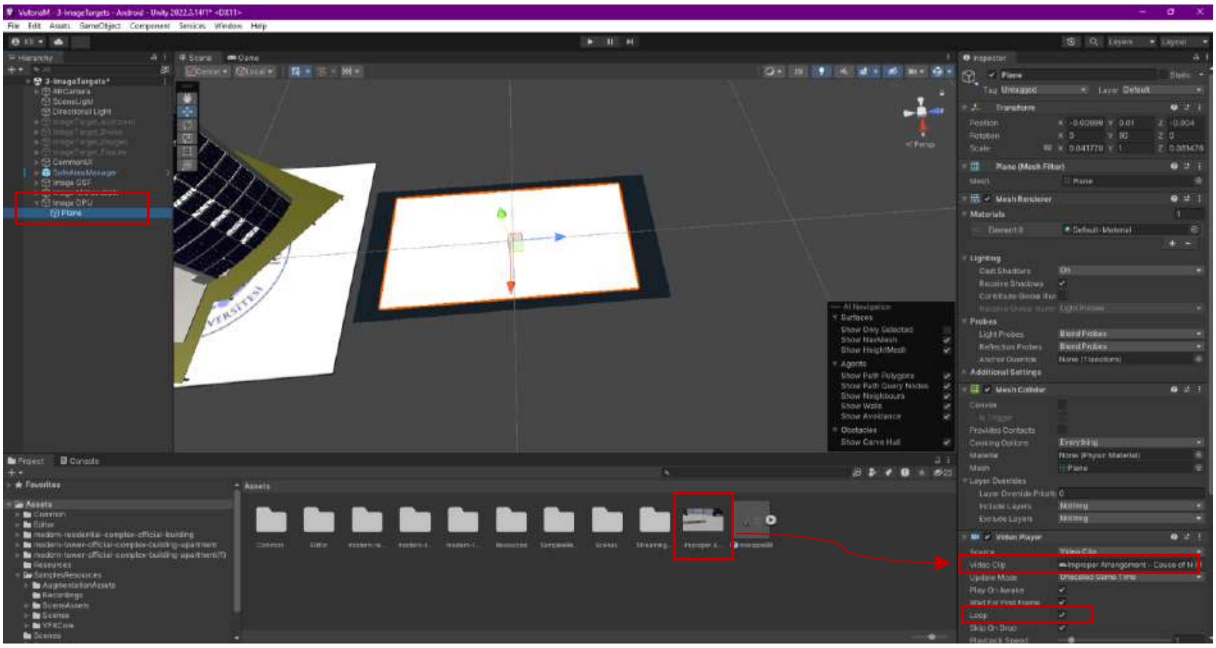




Now, add a **Video Player** with **Add Component** in the **Inspector** section of the **Plane**.



Drag and connect a video file that we prepared before to the **Video Player>Video Clip** section that we added to the **Plane** and activate the **Loop** box. Connect the **Plane** under **Image DPU**.



When we start the application on our phone, it will see the DPU logo, and the matching video will play on it. If desired, the sound of the video can be turned off by clicking the **Mute** check box.

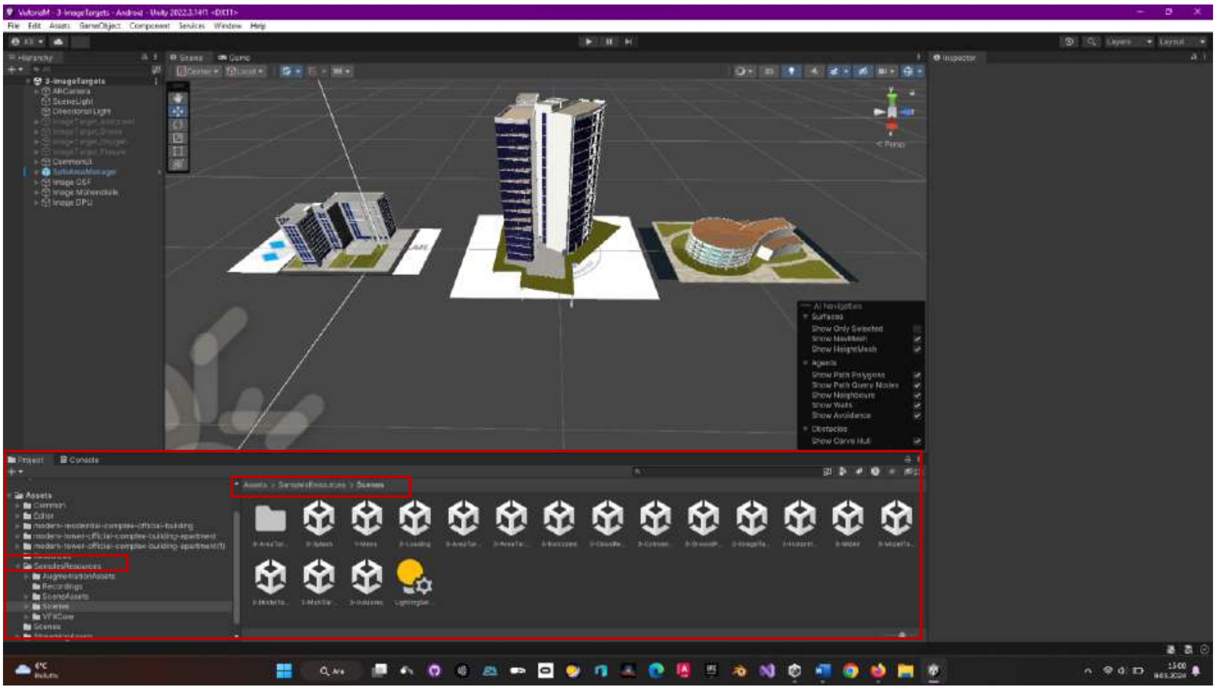
After getting information about single and **multi Image Target** applications, we can continue with **Ground Plane**, which is the application type where the model is displayed by scanning on the ground/surface without the need for any image…

11.8.Vuforia AR – Ground Plane

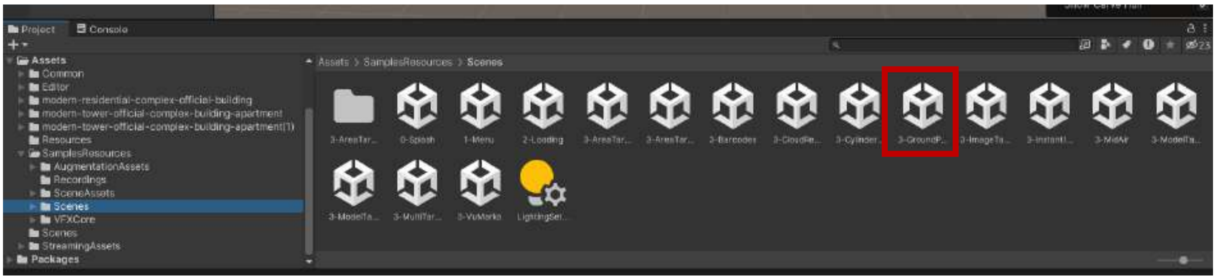
One type of augmented reality application is to open models directly on the ground plane without a target. In this study called **Ground Plane**, we will continue from the point reached in single and multiple target (**Image Target**) applications.

As you may recall, we downloaded the **Vuforia Core Samples** asset from the **Asset Store** and applied single and multiple targets (**Image Target**) to our project. Let's continue with the same project.

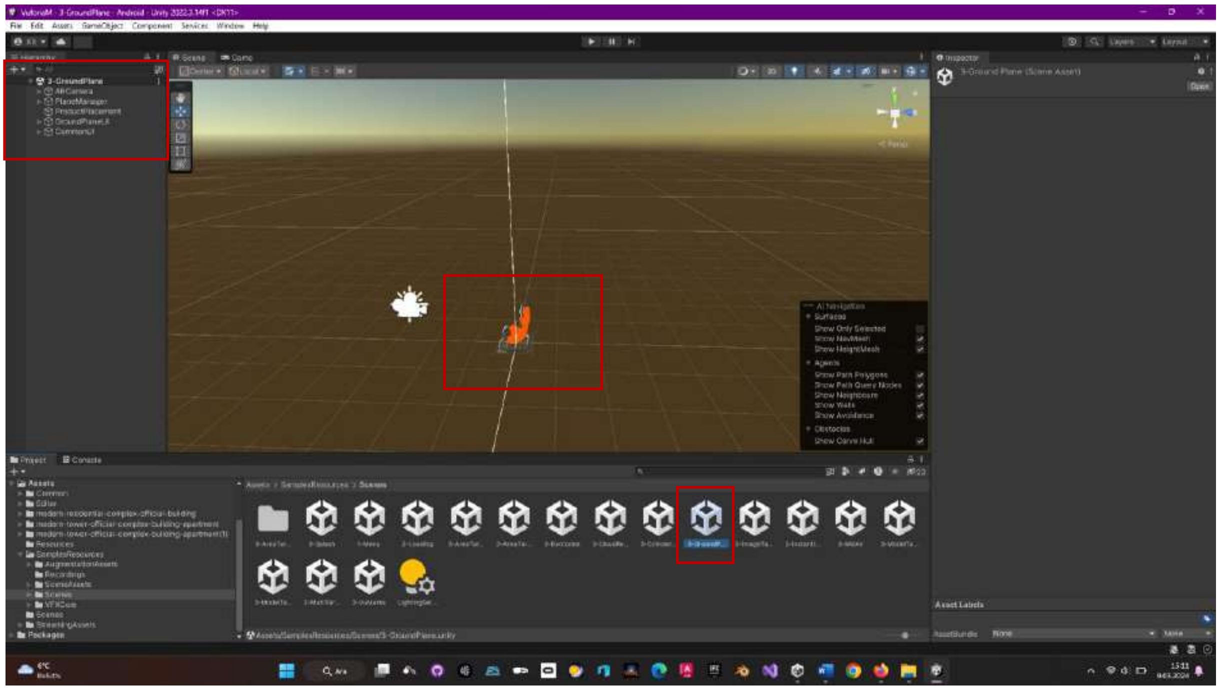
There are many AR application scenes available under **SamplesResources>Scenes** in the **Assets** section.

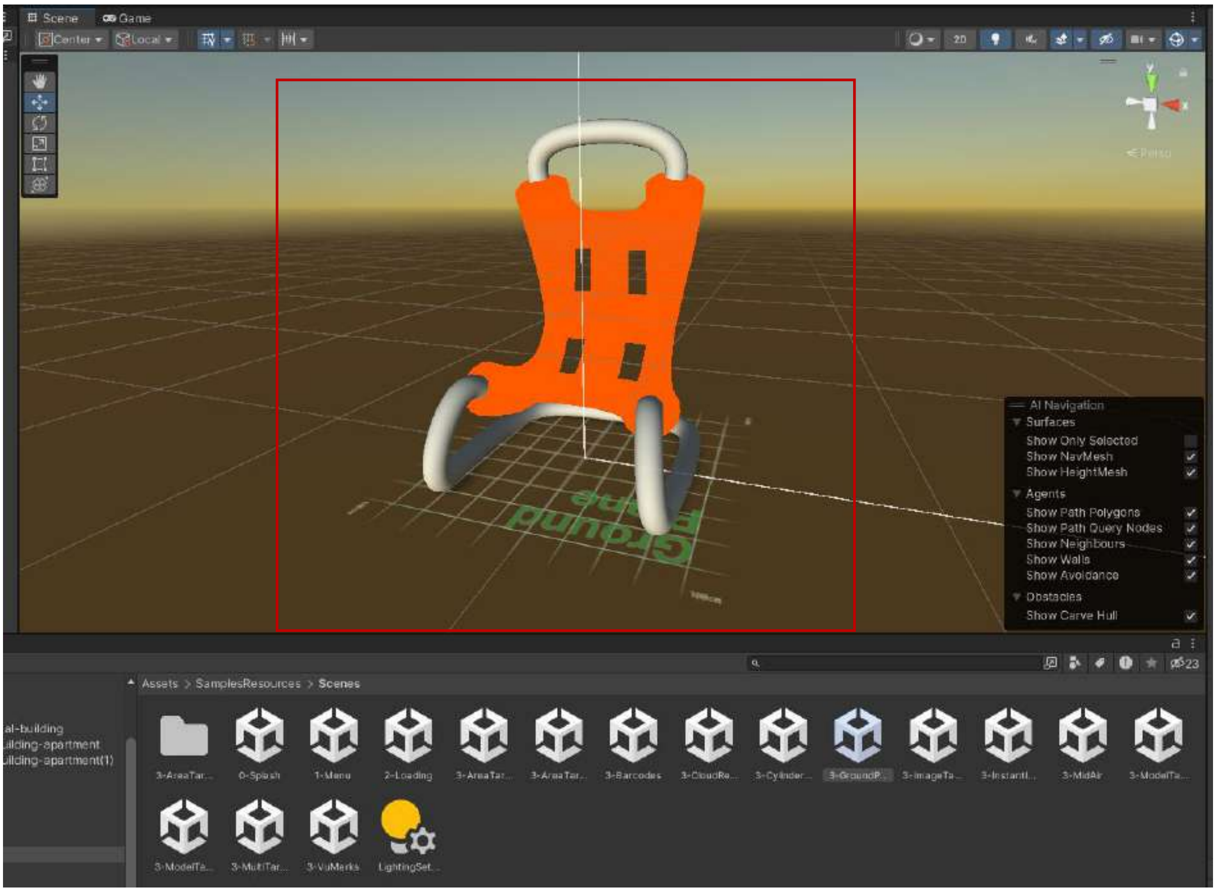


Now, open the **Ground Plane** scene from these **AR** models by double-clicking on it.

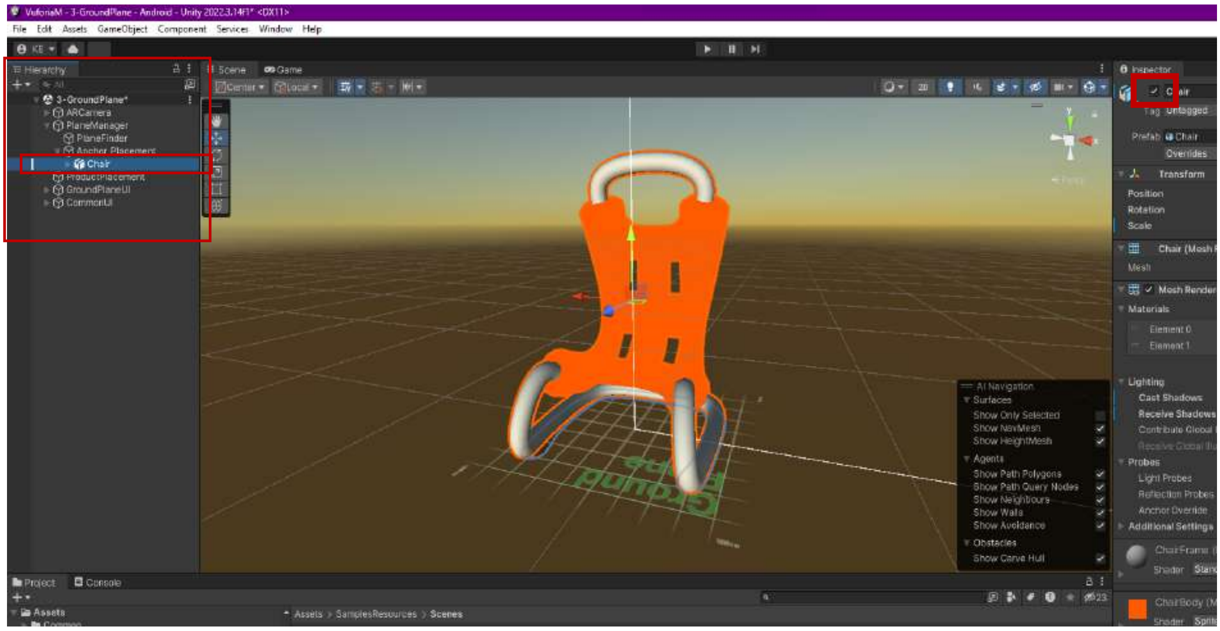


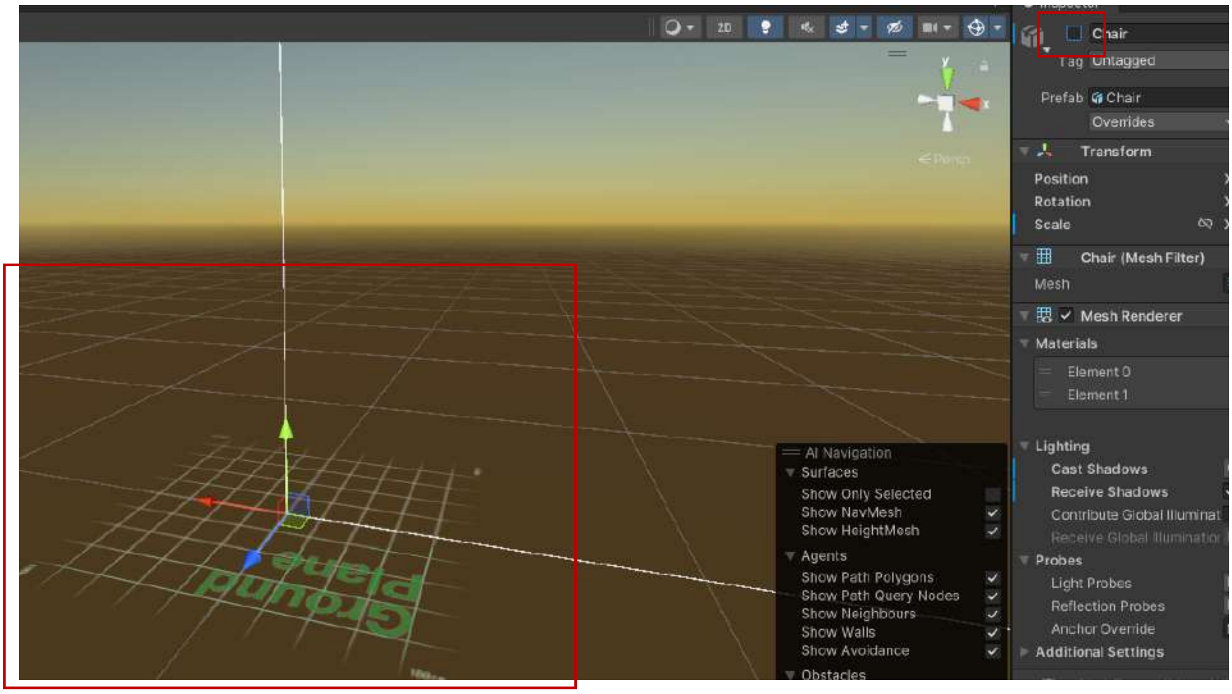
The **Ground Plane** scene is a template with a rocking chair model inside a UI (user interface) framework.



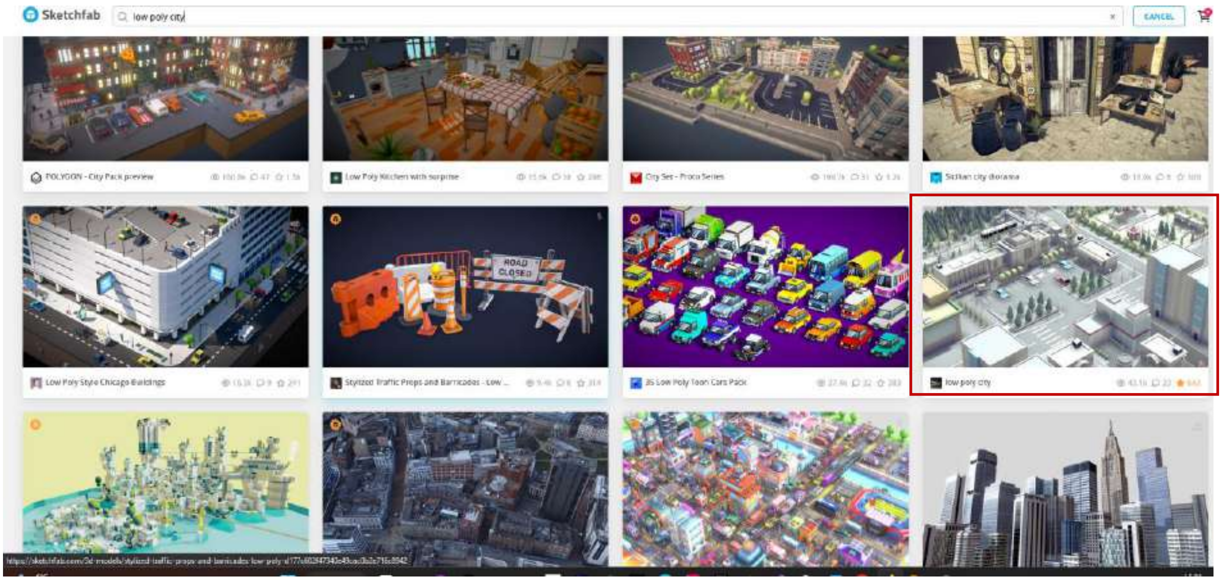


Since we want to open our model on the ground, make this object named **Chair** passive. For this purpose, access the chair with **Hierarchy>PlaneManager>Anchor_Placement>Chair** and **uncheck** its box to make it invisible.

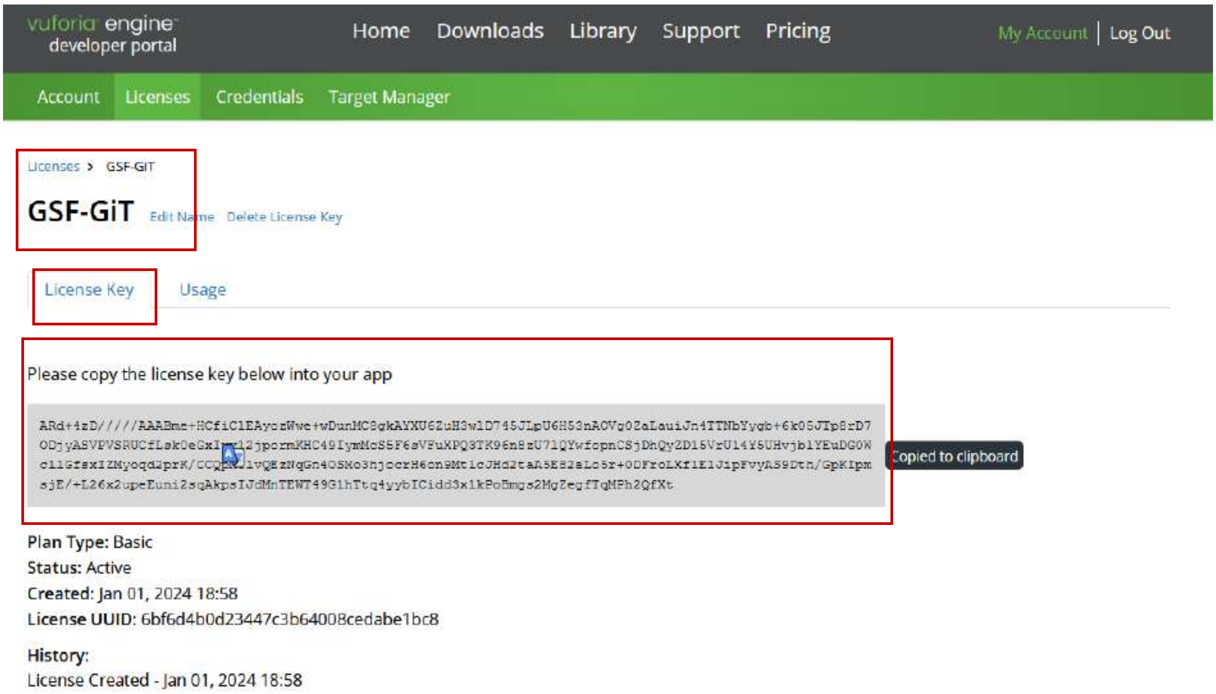




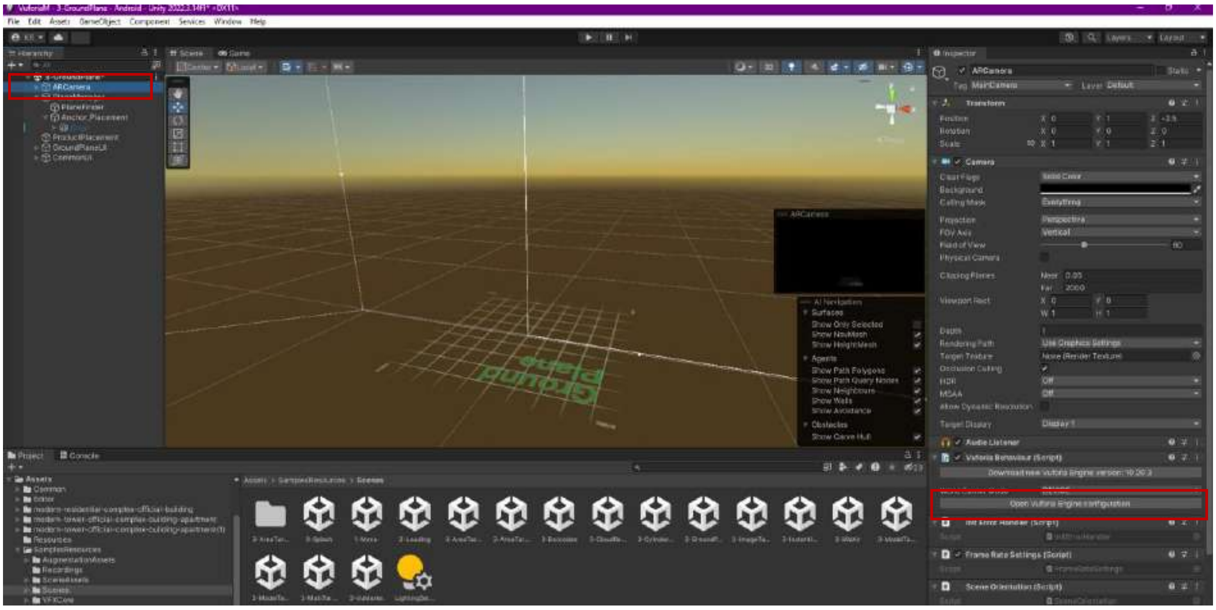
Use the low poly city model found on **Sketchfab**, which we use from time to time in Unity tutorials.



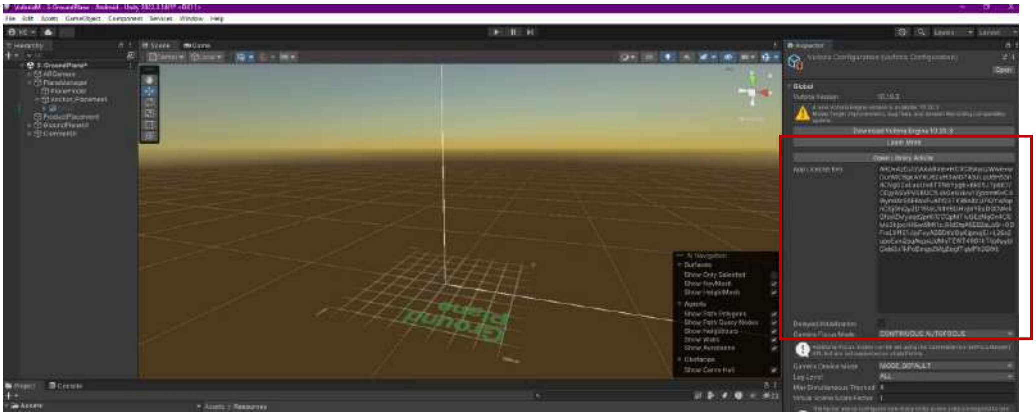
During the preparation phase of the scene, we will need the **license key** we created earlier on the **Vuforia** site. For this purpose, **GSF-Git > License Key**, located in the **Vuforia Licenses** section, will be used.



After copying the **license key**, click **Inspector>Open Vuforia Engine** configuration in **ARCamera**.

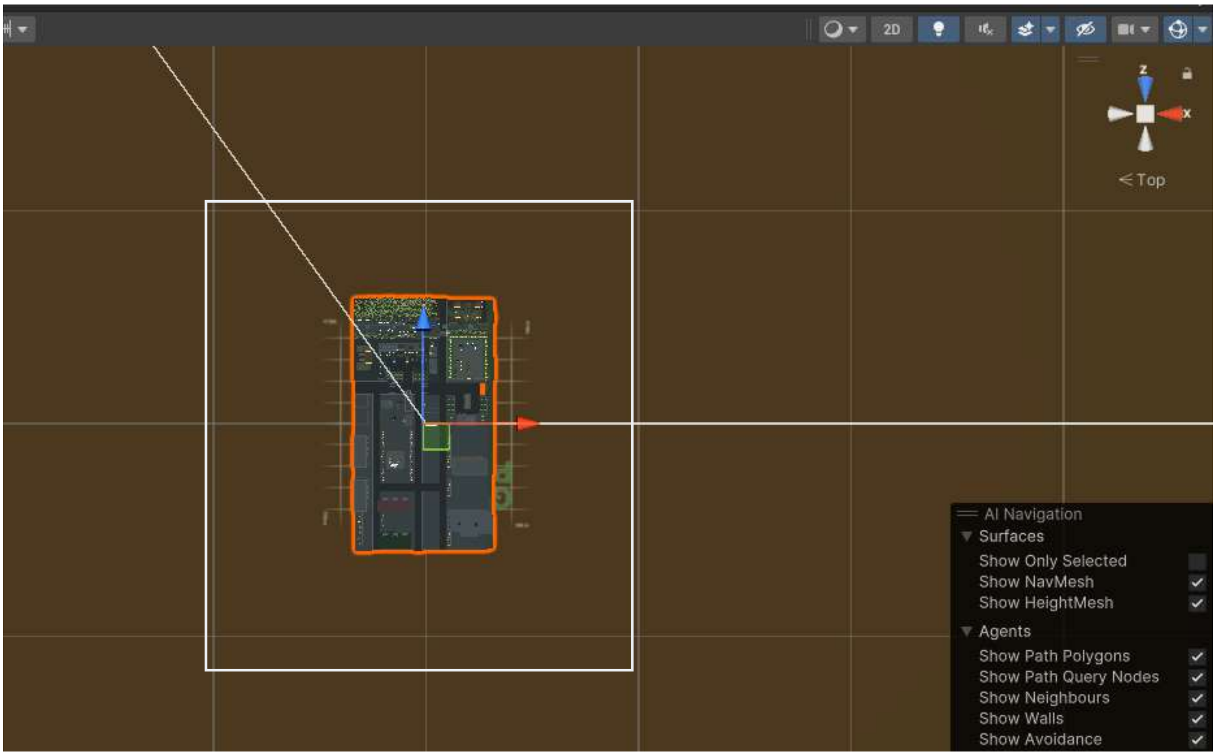


Since we are on the same project, we see that this **key** is already in the **App License Key** section. If we are in a new project, we will need to paste it here.

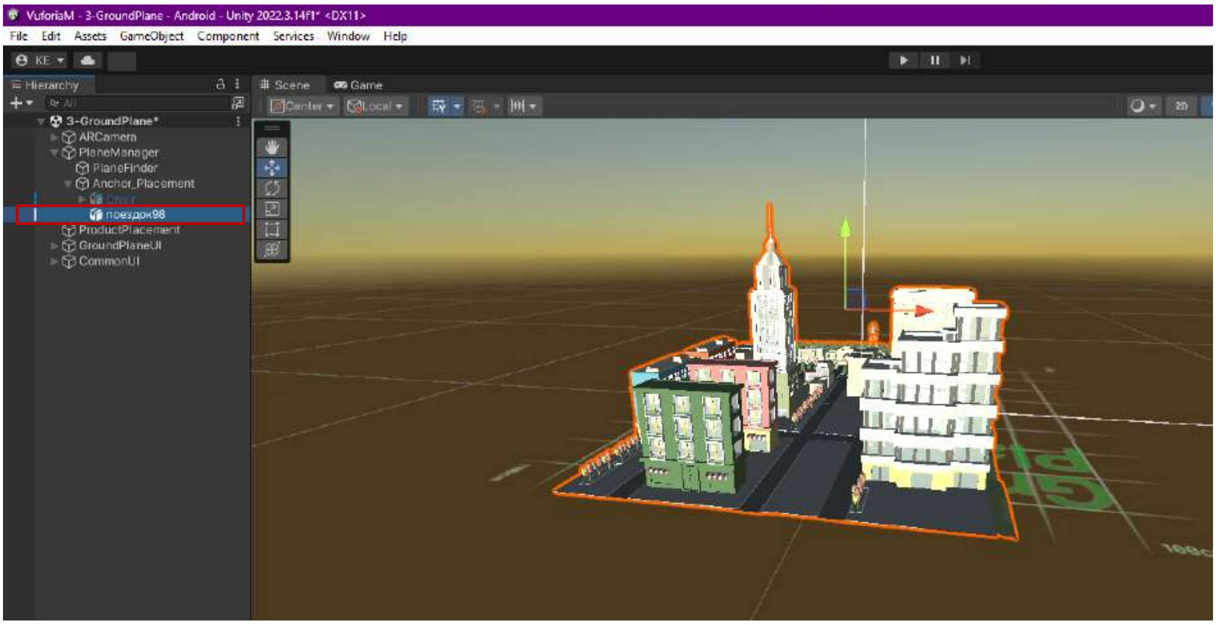


Drag the Sketchfab, low poly city, **fbx** file into **Assets**.

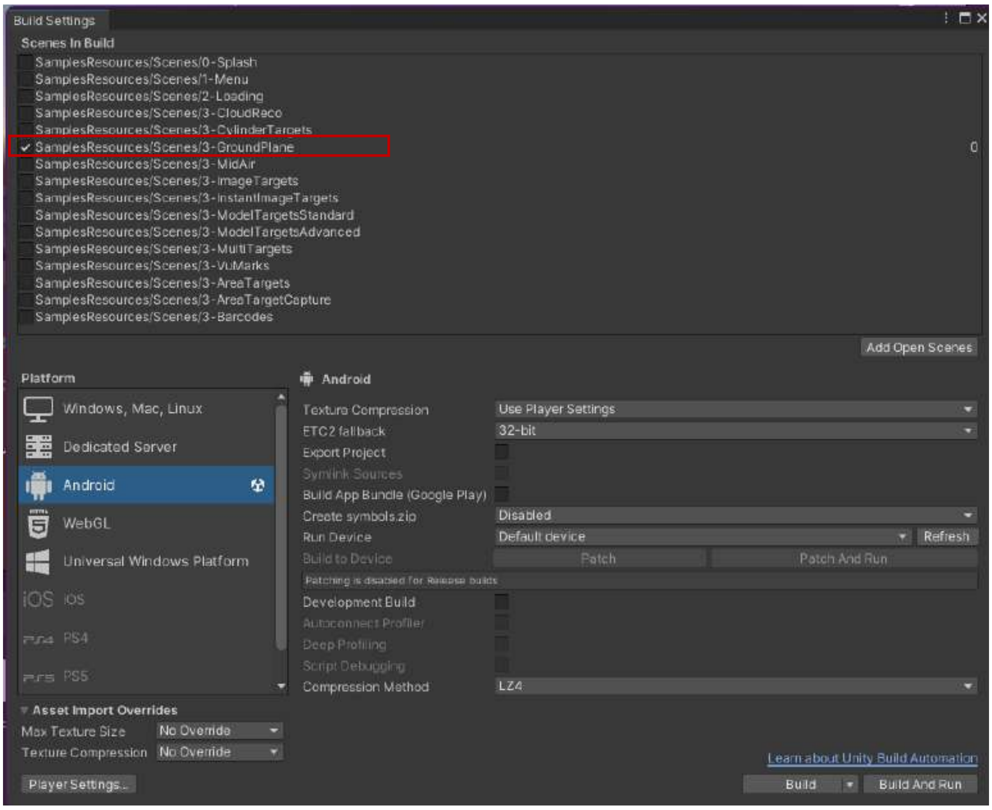
Then, position it in the **Ground Plane** area on the stage by minimizing it and using the gizmo to control it from the top, right and left.



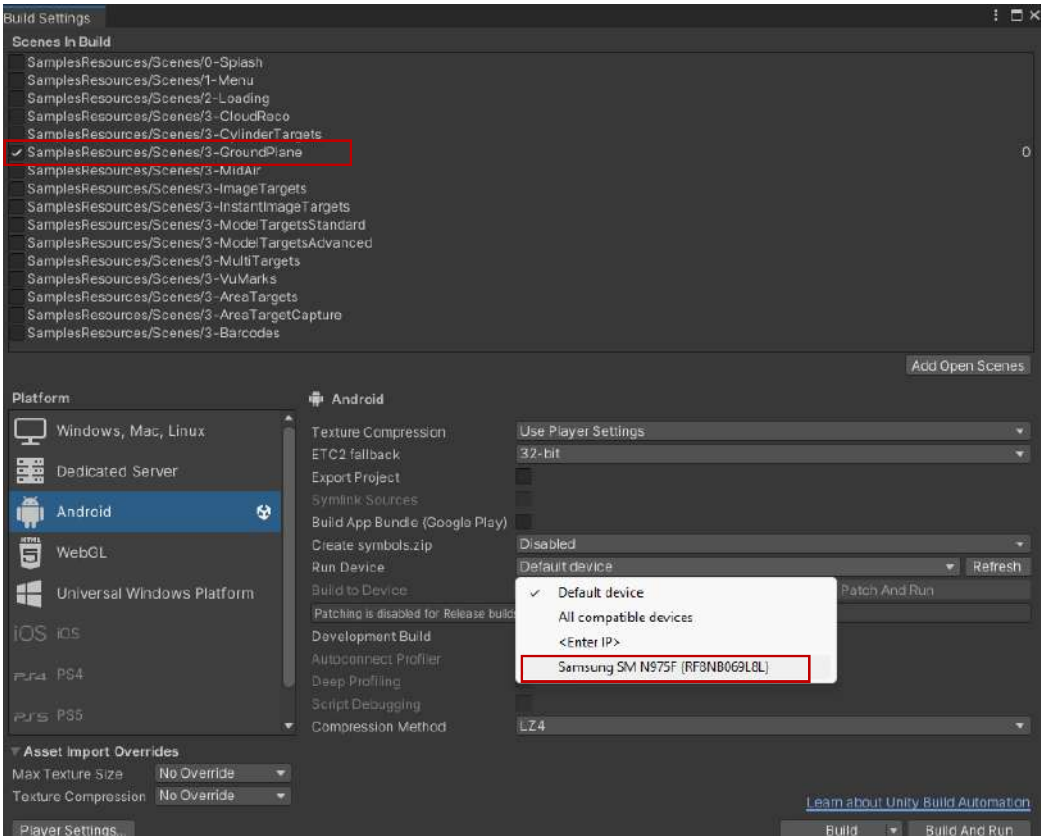
Drag and attach the model file under **PlaneManager>Anchor_Placement**. Check that the **Chair** object is inactive.



After the scene design, open the **File>Build Settings** window. Here, the **Android** (IOS for iPhone) platform should be switched to, and only **Ground Plane** should be selected from the scenes.

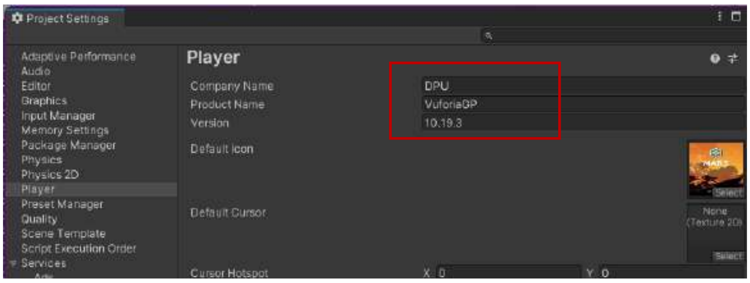


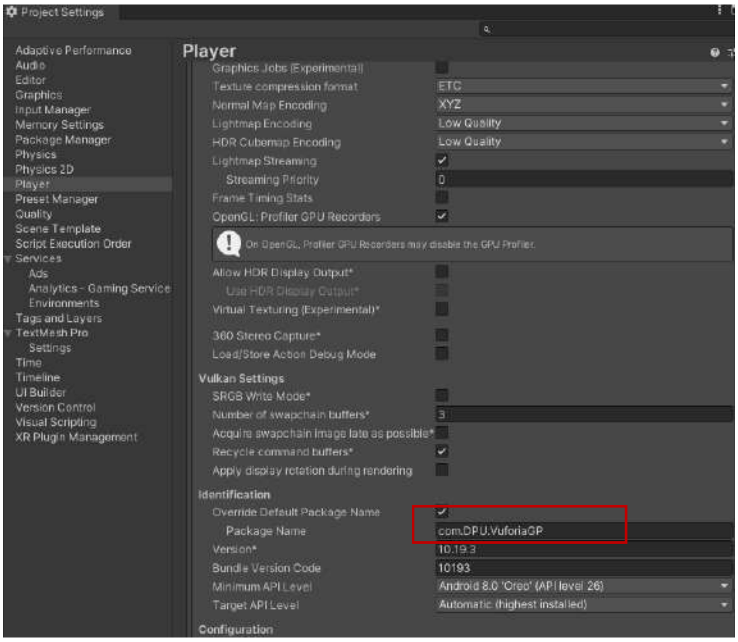
Connect the **mobile** device and check.



Make a few changes to the **Player Settings** to prevent applications from being overwritten.

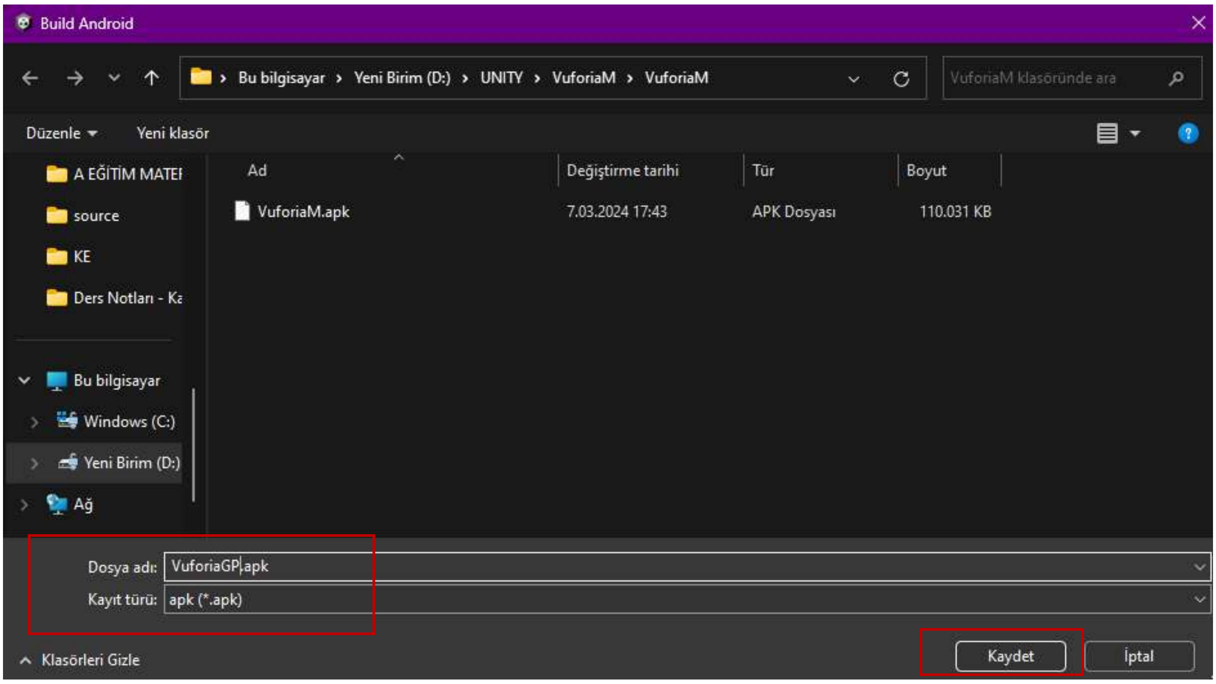
We named this application **VuforiaGP** as the **Production Name**. We ensured that the same name was used in the **Package Name** section. The abbreviation **GP** is considered a **Ground Plane**. Of course, every designer can create a naming system.





When we click on the **Build and Run** section, we will be asked in which folder the application will be created and its name. Enter our preference for this project; **VuforiaGP.apk**

The application will be created on both the disk and the phone by clicking the **save** button.



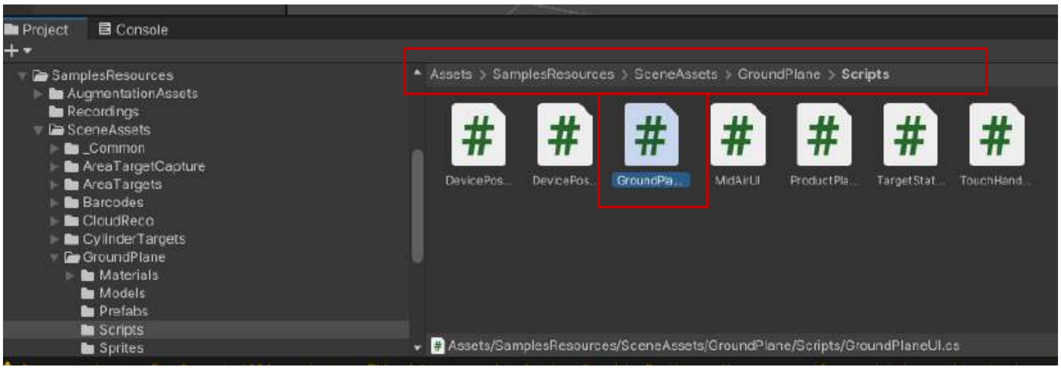
After opening the application on the **mobile** device, the ground/surface scan is performed. When the **honeycomb** texture is seen, the model can be loaded onto the surface/ground.

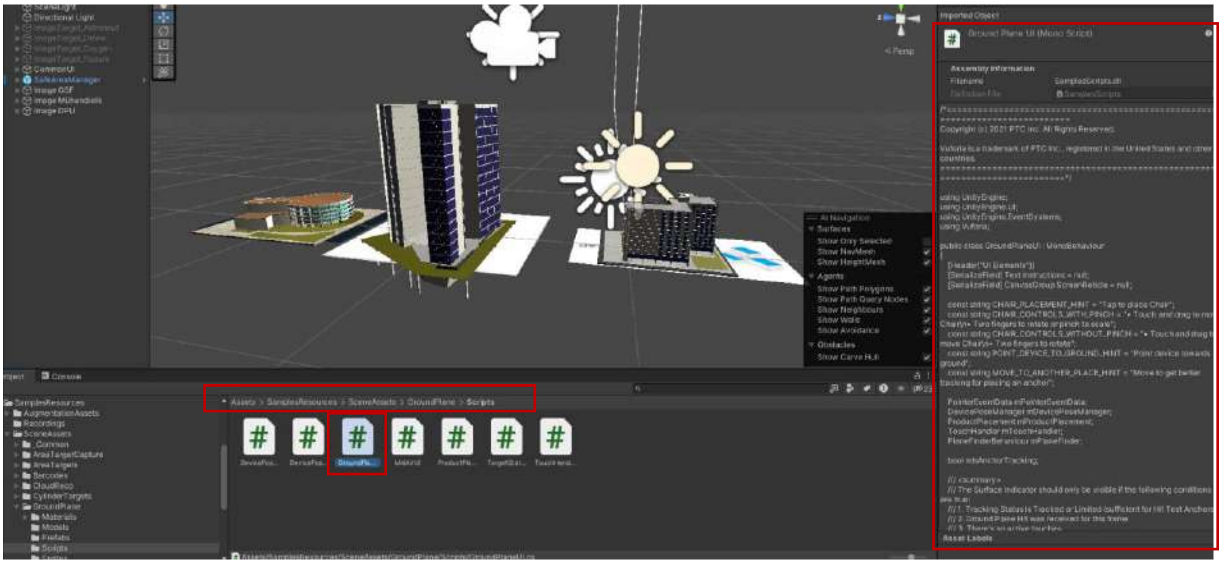
By touching the honeycomb, the model will appear here.

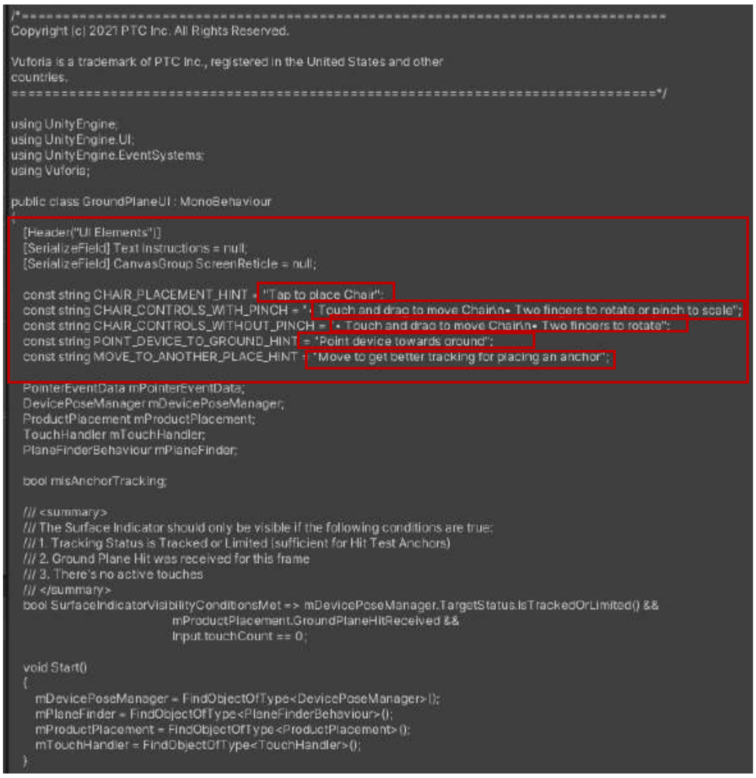
Turning the mobile device allows the honeycomb to be viewed in different locations and sizes, and the model reappears at that point.

The same model is displayed in front of the computer below.

Another issue is the **UI (interface)** texts seen on the screen. **Texts** such as **Product Placement, Tap to place Chair, Touch and Drag to move Chair, Two fingers to rotate** are templates and are for the **disabled** chair and can be **changed**. For a change, the relevant lines can be found in the **script** (**C# code**) files and changed with the desired expressions.





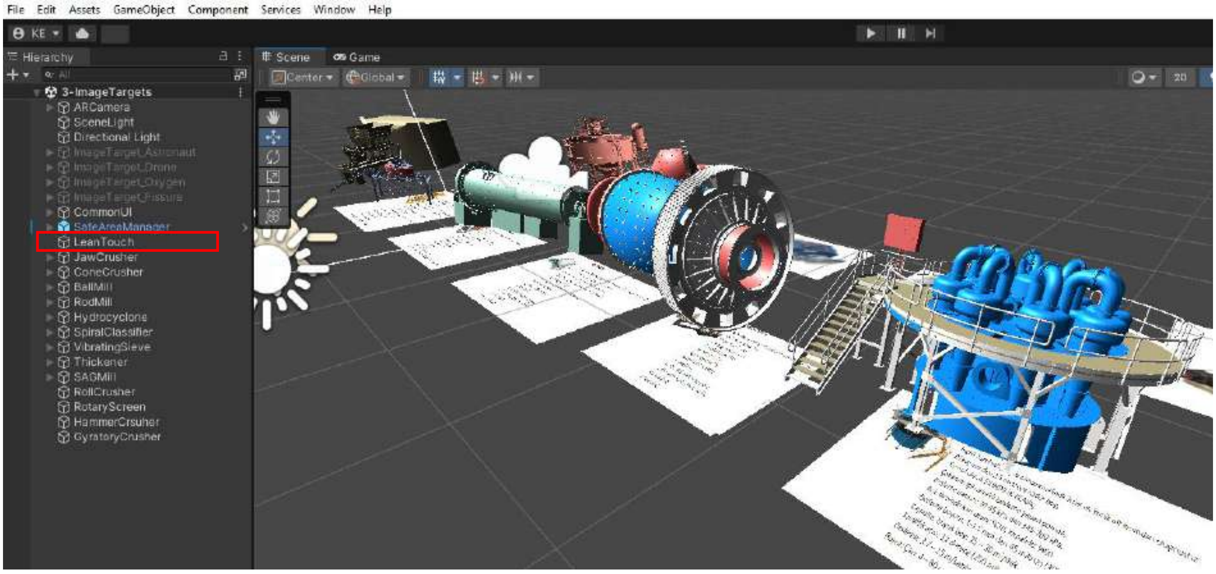


```
/*==============================================================================
Copyright (c) 2021 PTC Inc. All Rights Reserved.

Vuforia is a trademark of PTC Inc., registered in the United States and other
countries.
==============================================================================*/

using UnityEngine;
using UnityEngine.UI;
using UnityEngine.EventSystems;
using Vuforia;

public class GroundPlaneUI : MonoBehaviour
{
    [Header("UI Elements")]
    [SerializeField] Text instructions = null;
    [SerializeField] CanvasGroup ScreenReticle = null;

    const string CHAIR_PLACEMENT_HINT = "Tap to place Chair";
    const string CHAIR_CONTROLS_WITH_PINCH = "• Touch and drag to move Chair\n• Two fingers to rotate or pinch to scale";
    const string CHAIR_CONTROLS_WITHOUT_PINCH = "• Touch and drag to move Chair\n• Two fingers to rotate";
    const string POINT_DEVICE_TO_GROUND_HINT = "Point device towards ground";
    const string MOVE_TO_ANOTHER_PLACE_HINT = "Move to get better tracking for placing an anchor";

    PointerEventData mPointerEventData;
    DevicePoseManager mDevicePoseManager;
    ProductPlacement mProductPlacement;
    TouchHandler mTouchHandler;
    PlaneFinderBehaviour mPlaneFinder;

    bool misAnchorTracking;

    /// <summary>
    /// The Surface Indicator should only be visible if the following conditions are true:
    /// 1. Tracking Status is Tracked or Limited (sufficient for Hit Test Anchors)
    /// 2. Ground Plane Hit was received for this frame
    /// 3. There's no active touches
    /// </summary>
    bool SurfaceIndicatorVisibilityConditionsMet => mDevicePoseManager.TargetStatus.IsTrackedOrLimited() &&
                            mProductPlacement.GroundPlaneHitReceived &&
                            Input.touchCount == 0;

    void Start()
    {
        mDevicePoseManager = FindObjectOfType<DevicePoseManager>();
        mPlaneFinder = FindObjectOfType<PlaneFinderBehaviour>();
        mProductPlacement = FindObjectOfType<ProductPlacement>();
        mTouchHandler = FindObjectOfType<TouchHandler>();
    }
```

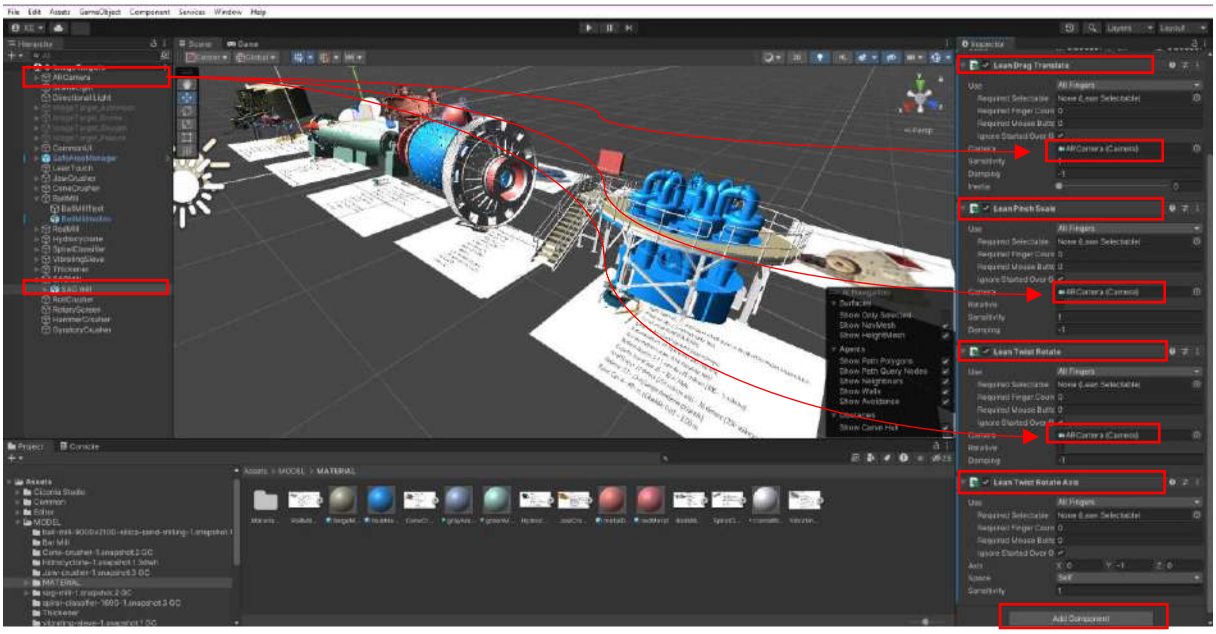
Even the **honeycomb - reticle_ground_surface (sight)** and other similar figures seen on the screen under **Sprites** can be changed.

Although the commands include **dragging** and **rotating** the model on the mobile device screen, if these features do not work, the asset named **Lean Touch** can be downloaded and used from the Unity Asset Store. Information about this is included in the asset introduction.

In the example, **mineral processing** machines are used for **Image Target** application of **Vuforia**. After **Import**ing process of **Lean Touch** asset, right-click on **Hierarchy** window and add **Lean Touch** object.



Now, we can **add** the related **Lean Touch** components to the objects located on the scene. For each asset, **Lean Drag Translate, Lean Pinch Scale, Lean Twist Rotate** and **Lean Twist Rotate Axis** components are added (e.g., **SAG mill**). Drag the **ARCamera** to the components in the **Inspector** having **Camera** use.

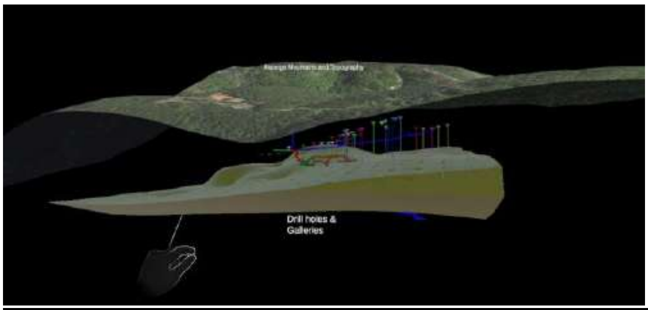


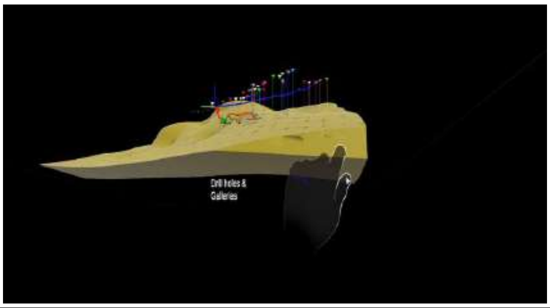
As a result, when the **APK** file is run, **SAG mill** object can be controlled by **fingers**. It can be **dragged**, **scaled** and **rotated**.

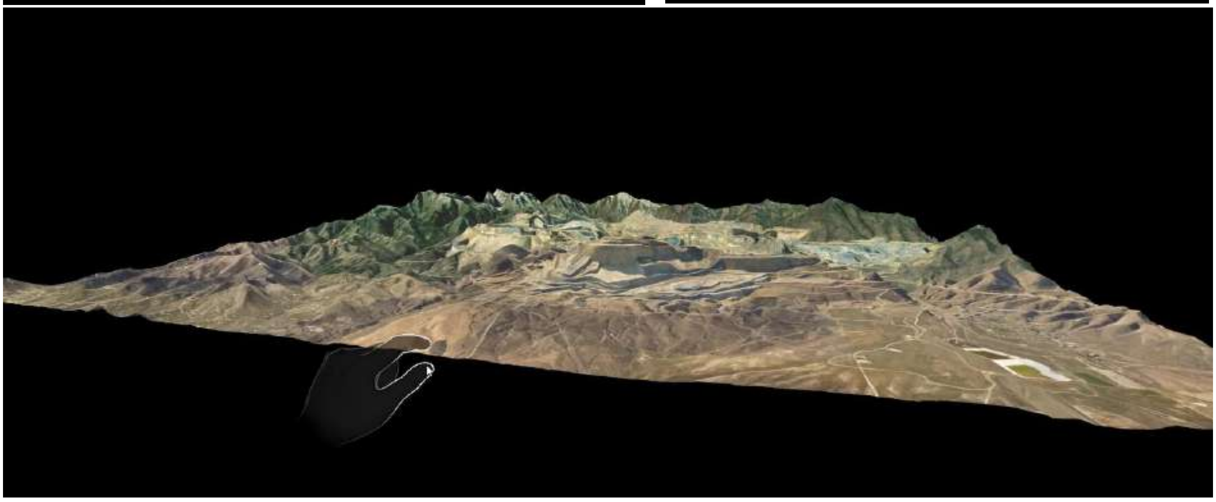
11.9.Similar Examples for Mining

The applications developed for an open pit mine and a quarry are given below. One is in front of the computer, and the other is on the floor of the university corridor.

Although, Hololens 2 applications are not included in the book, some of deployments are shown to give an idea.



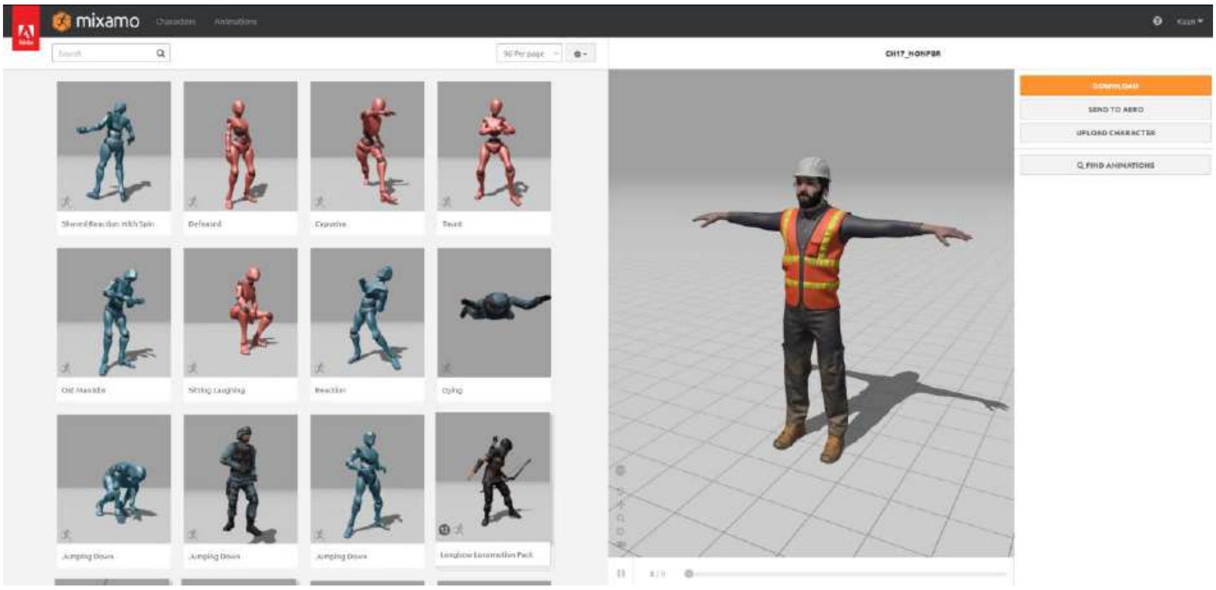




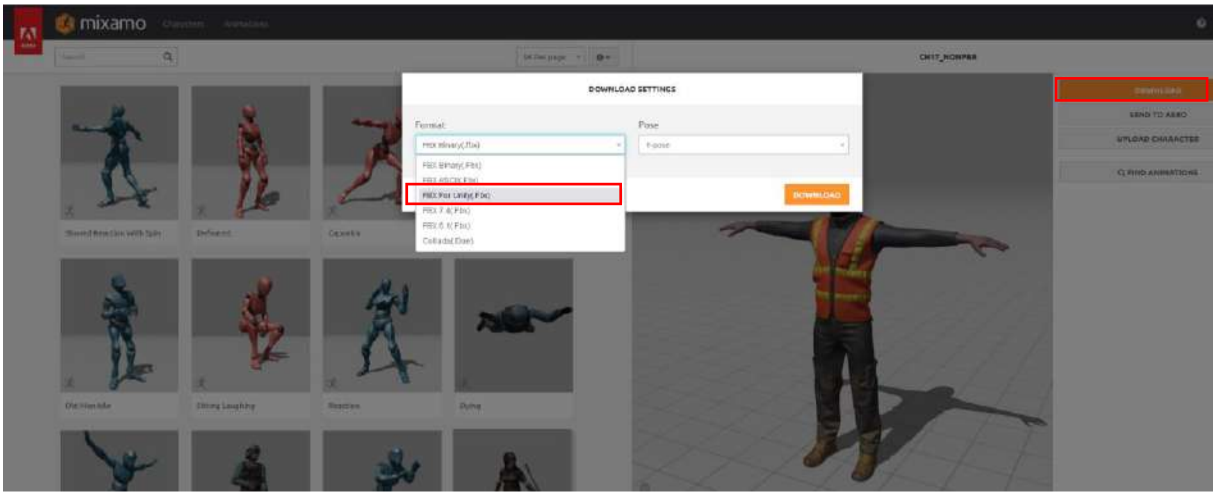
12. USE OF ANIMATION AND ADOBE MIXAMO CHARACTERS IN UNITY

Adobe Mixamo (www.mixamo.com) has a large library of characters and animations. The characters and animations found here can be combined and turned into interactive animations in the animator editor in Unity.

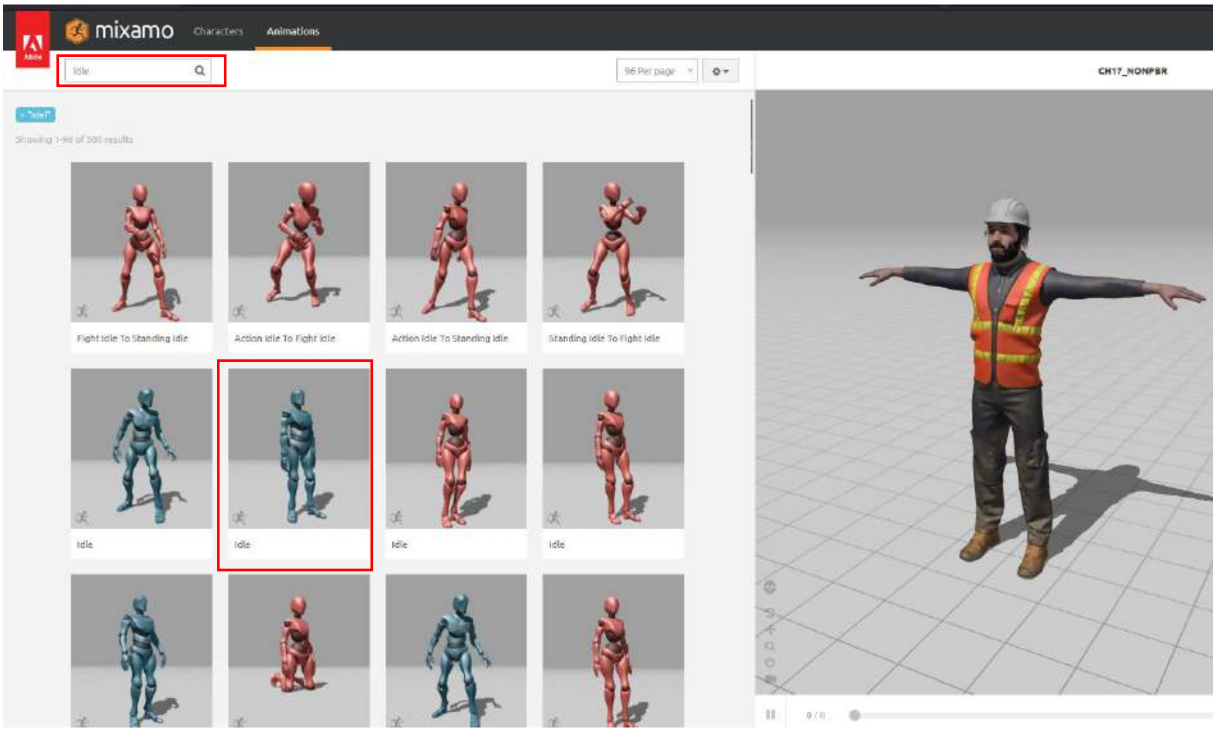
In this application, we can download a **worker armature** (*body*) and some animations from Mixamo and how we can control it with **C# coding** developed in Unity.



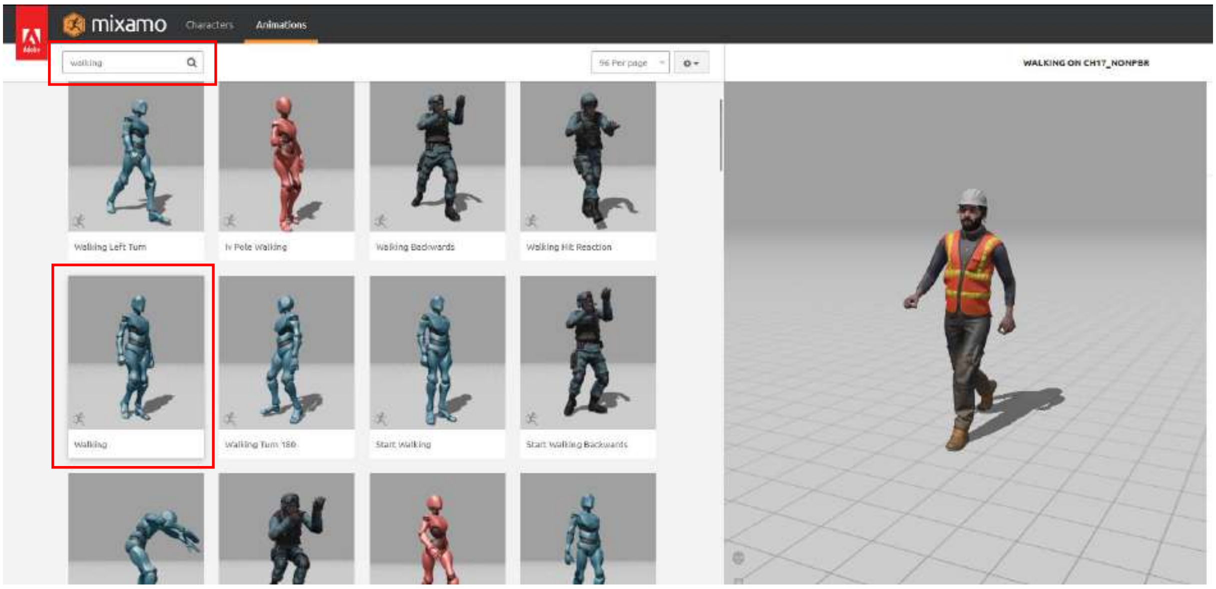
Let's download the root stance of the selected character, **T-Pose**. Here, from the window that appears with the **Download** button, download it by selecting the **FBX For Unity** (.Fbx) line.

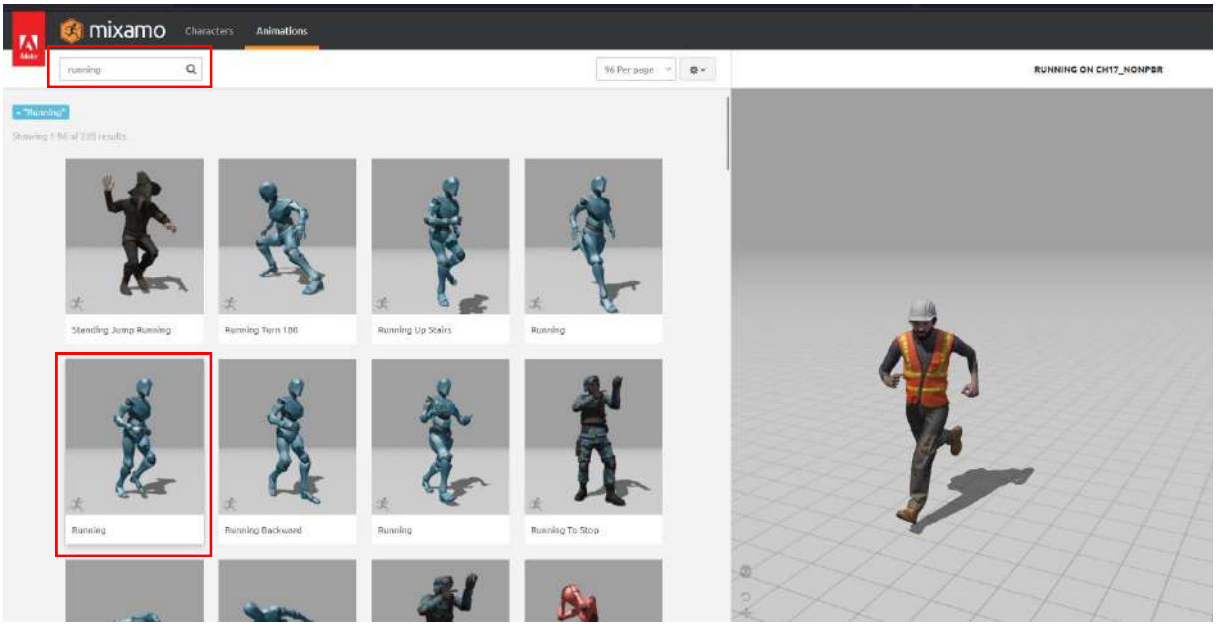


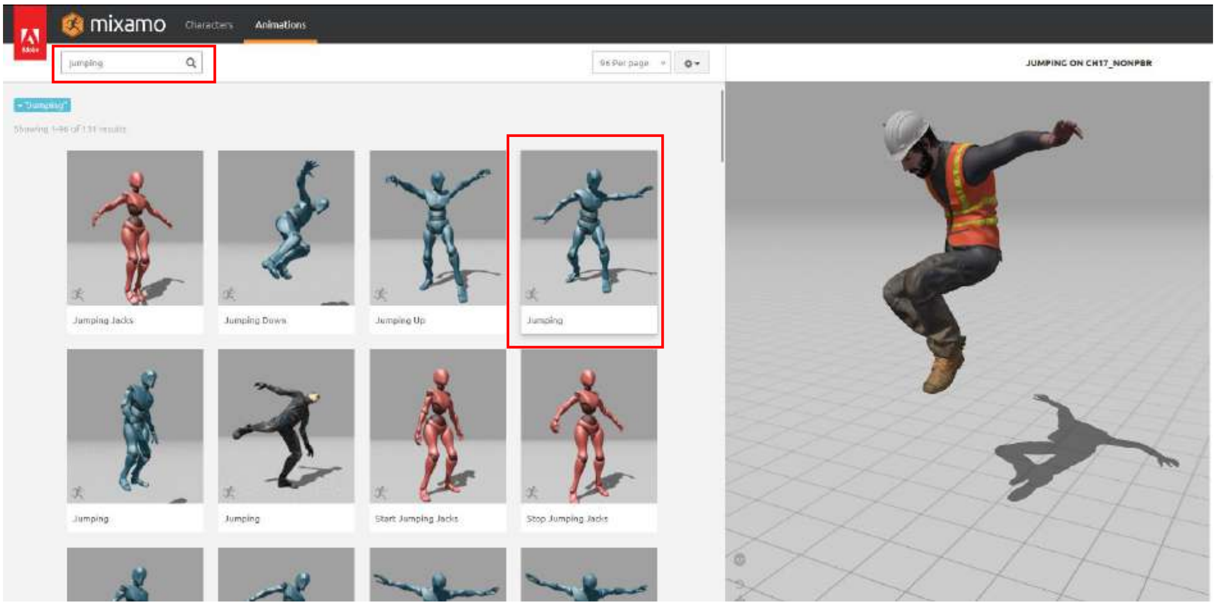
Now, search for the word **Idle** under the **Animations** tab and choose one of the many options that appear that we find appropriate. With this choice, the animation is combined with our character in the **T-Pose** state and plays on it. Download this by clicking the **Download** button and selecting Unity.



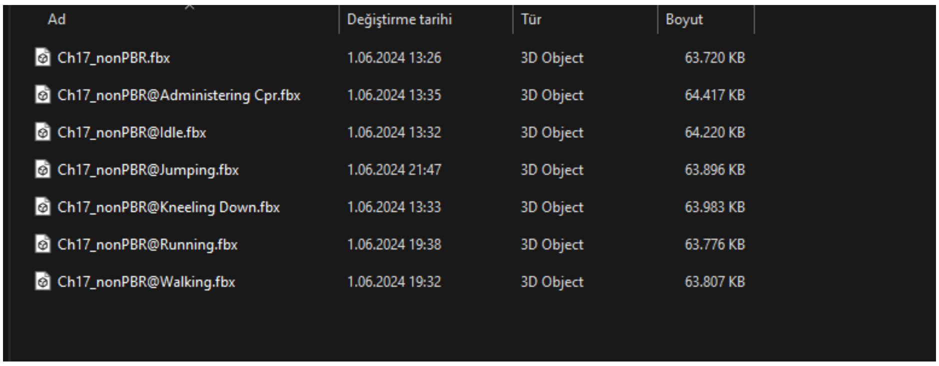
Similarly, search for **Walking, Running and Jumping** animations, observe them on the worker and **download** them in the same way.





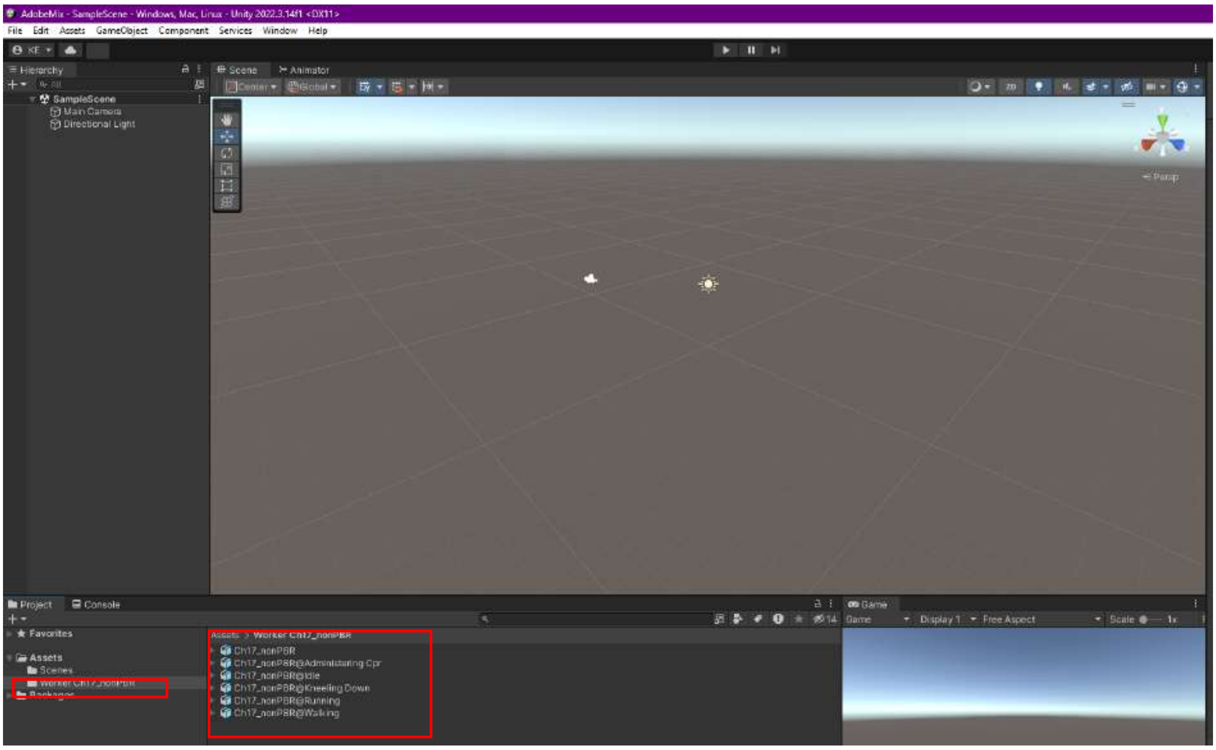


Now, collect the downloaded files in a folder to place in the **Assets** section of the Unity project we will create. [*Extra kneeling and CPR are also included here*]



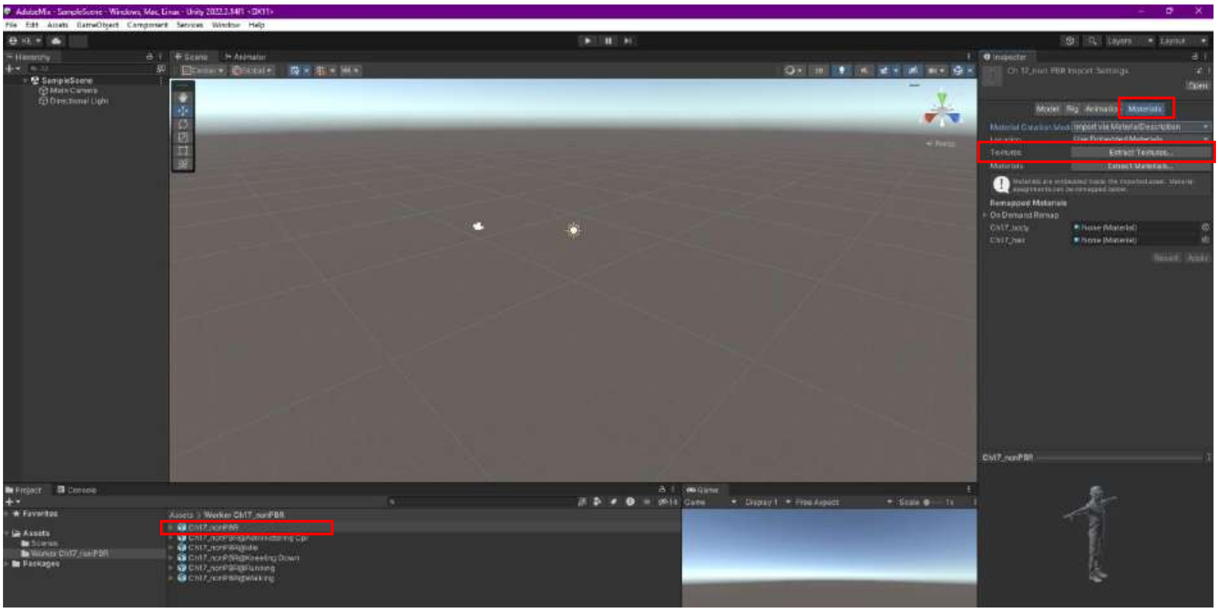
Ad	Değiştirme tarihi	Tür	Boyut
Ch17_nonPBR.fbx	1.06.2024 13:26	3D Object	63.720 KB
Ch17_nonPBR@Administering Cpr.fbx	1.06.2024 13:35	3D Object	64.417 KB
Ch17_nonPBR@Idle.fbx	1.06.2024 13:32	3D Object	64.220 KB
Ch17_nonPBR@Jumping.fbx	1.06.2024 21:47	3D Object	63.896 KB
Ch17_nonPBR@Kneeling Down.fbx	1.06.2024 13:33	3D Object	63.983 KB
Ch17_nonPBR@Running.fbx	1.06.2024 19:38	3D Object	63.776 KB
Ch17_nonPBR@Walking.fbx	1.06.2024 19:32	3D Object	63.807 KB

Open our Unity project. Place our folder (*Worker Ch17_nonPBR*) in the Assets section.

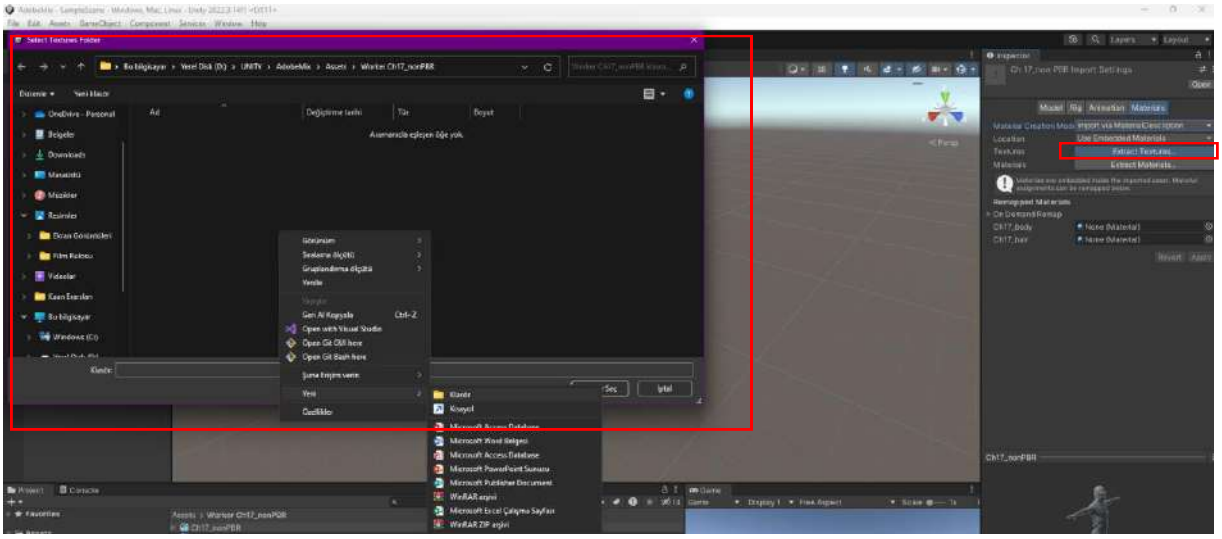


Before we import the character into our scene, we need to make some presets on the FBX files.

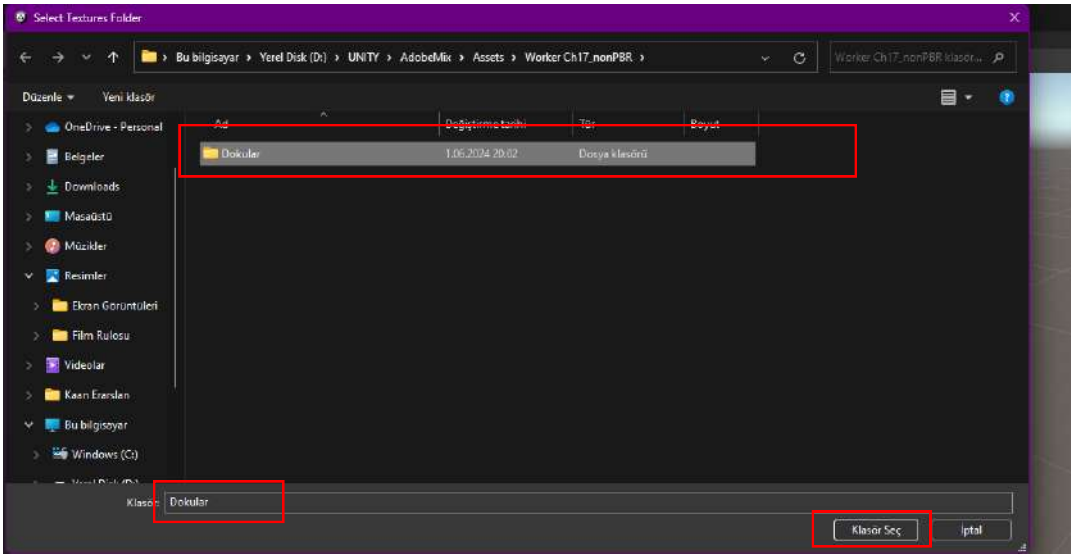
When we select the **Ch17_nonPBR** file with the **T-Pose** position, we see that it has no colors. Let's assign its textures. Extract the textures to a folder by clicking **Materials>Textures>Extract Textures**.



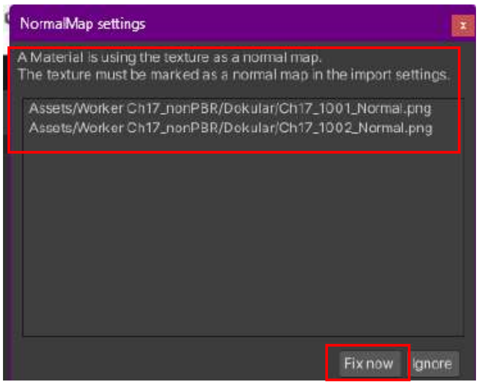
When you click on **Extract Texture**, you will be asked which folder the textures will be extracted to. We will need to open this folder, right-click, give it a name, and select it.



Here, the folder was created by naming it **Textures**. Select it with **Select Folder**.

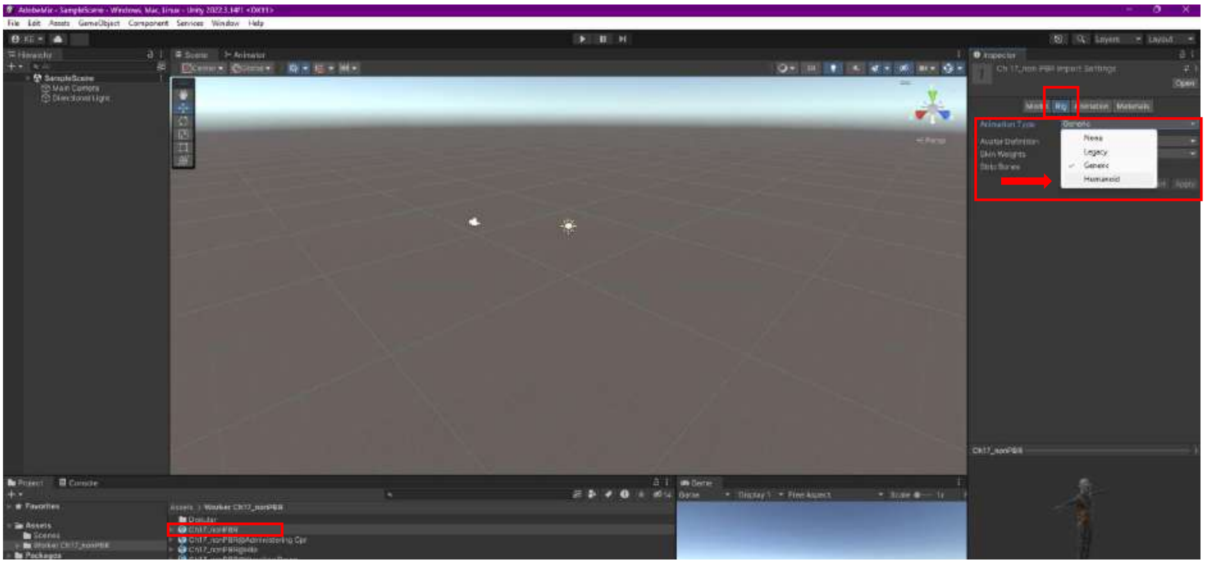


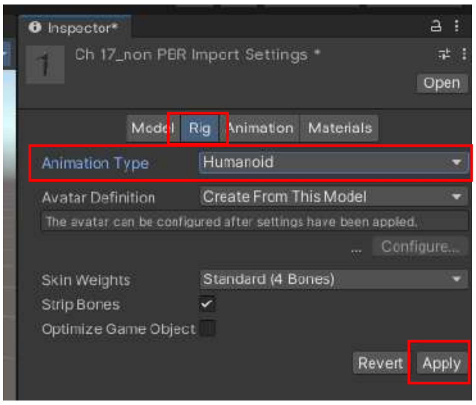
If there is a need for correction in texture assignments, it warns. Fix this by saying **Fix Now**.



Thus, the colors of the worker character appear.

Now, see a series of operations that need to be done on all **FBX** files. The **Animation Types** in the **Rig** section of all downloaded character files must be converted to **Humanoid** form.





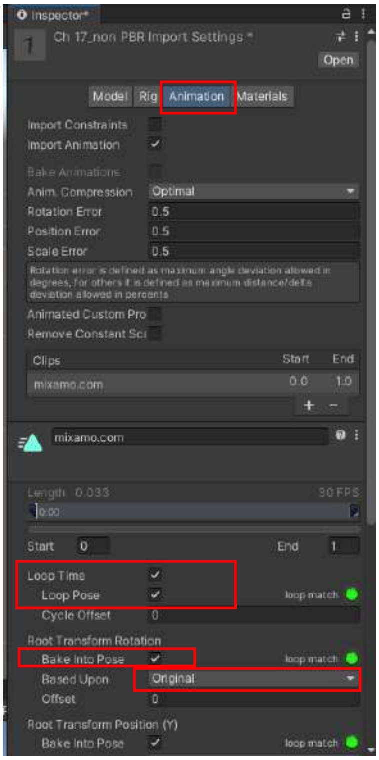
Also, some animations need to have some repetition motion. Below, the changes made under **Animations** are shown in two consecutive frames.

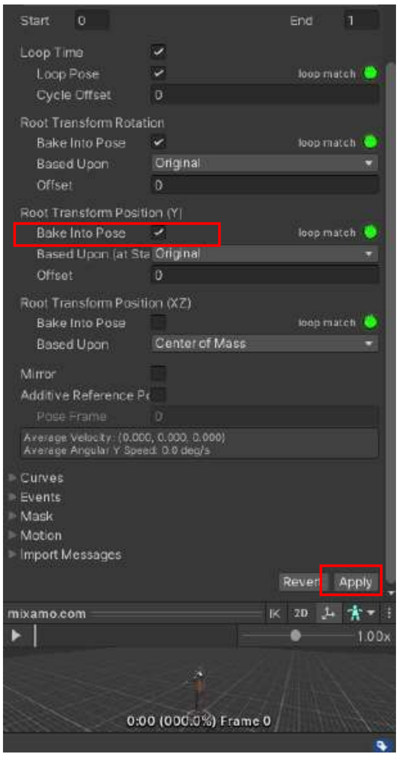
Here;

* **Loop Time** and **Loop Pose** boxes should be activated (checked).

* **Root Transform Rotation > Bake Into Pose** should be active and turn **Based Upon** into **Original**.

* **Root Transform Position (Y) > Bake Into Pose** should be active.



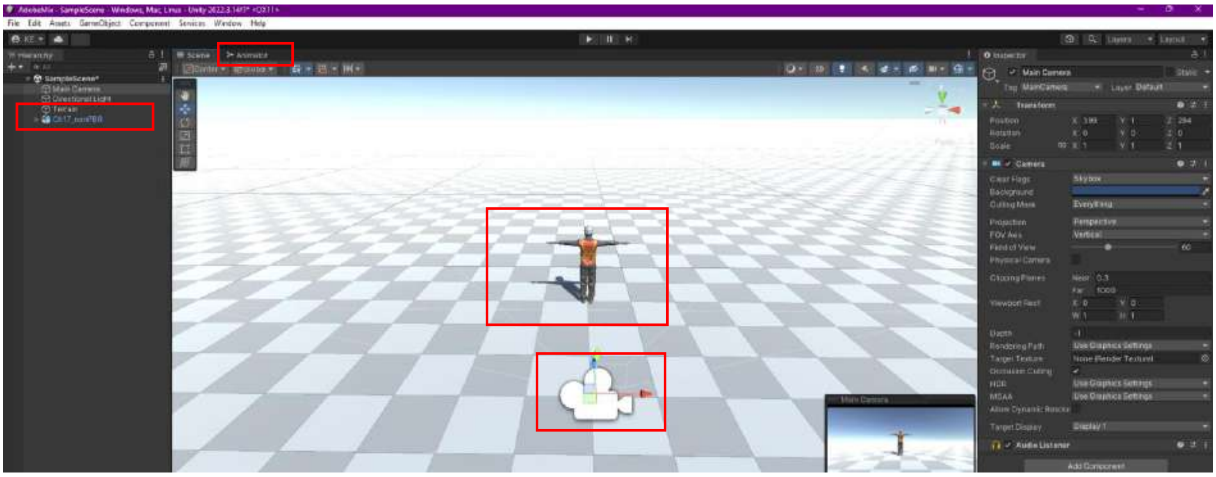


In this way, all **FBX** files should be converted to **Humanoid** form, changes should be made in the **Animations** section and **Apply** should be done.

After the operations on the files included in the project are completed, we can move on to the animation and animator phase.

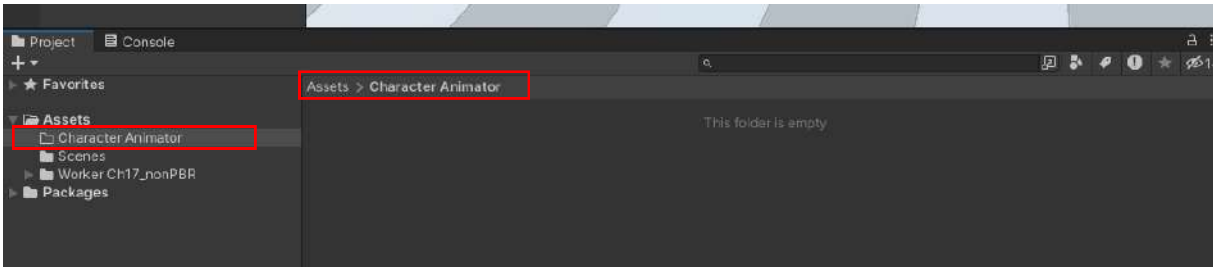
Let's add a Terrain to our scene to provide freedom of movement.

Then, position the **Idle** (**fbx with T-Pose**) file and the **Main Camera** to see it.

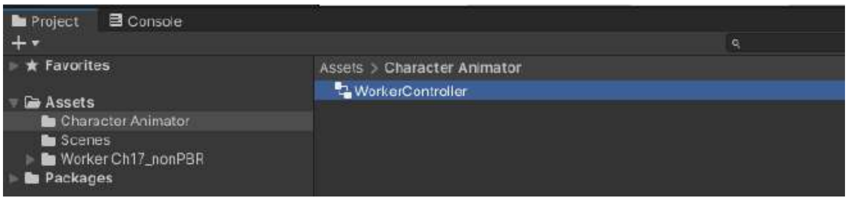


If you notice, since the characters have animation, the **Animator** tab also appeared next to the **Scene** tab. If this is not opened, we can open it by pressing **Windows>Animation>Animator**.

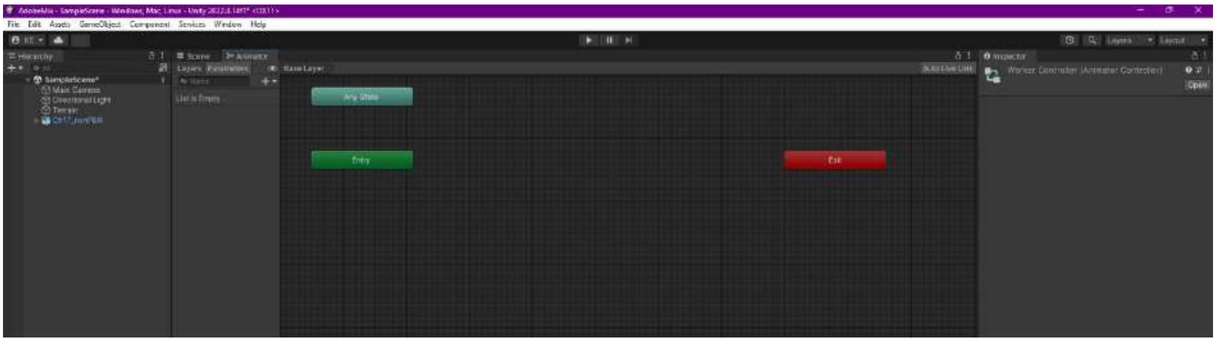
Create a folder under **Assets** named **Create>Folder>Character Animator** and go into it.



Here, create an animator controller with **Create>Animator Controller** and name it **WorkerController**.



When this process is done, **three states** are created in the **editor** in the **Animator** tab, **including Any State, Entry** and **Exit** blocks.

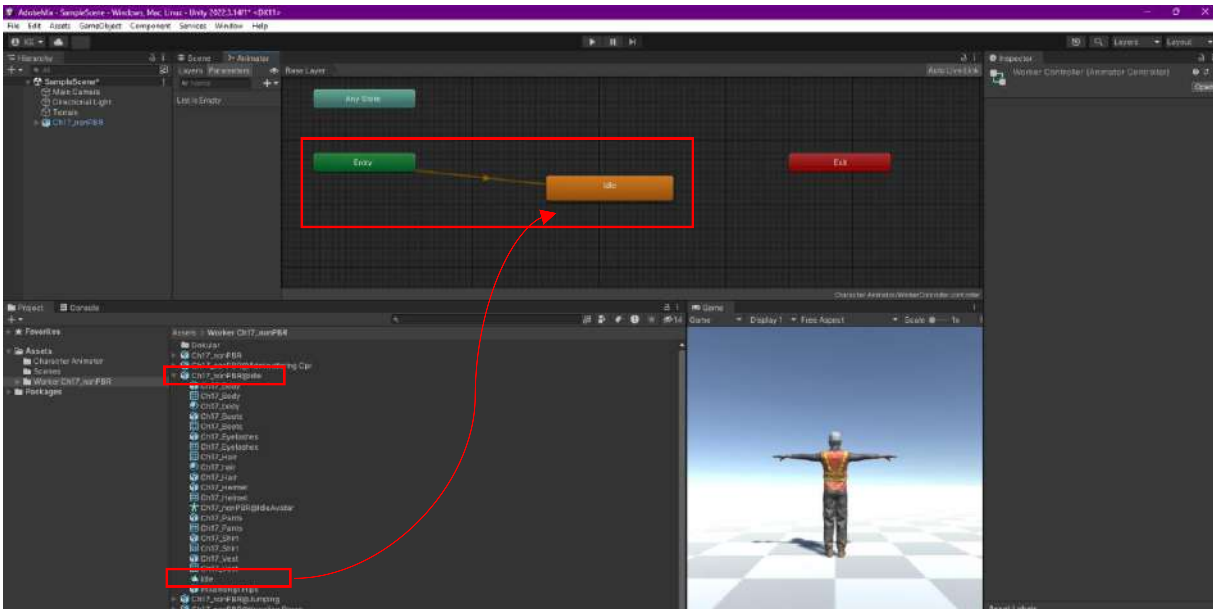


First, make animations of the transition from the **idle** position to **walking** and **running** states.

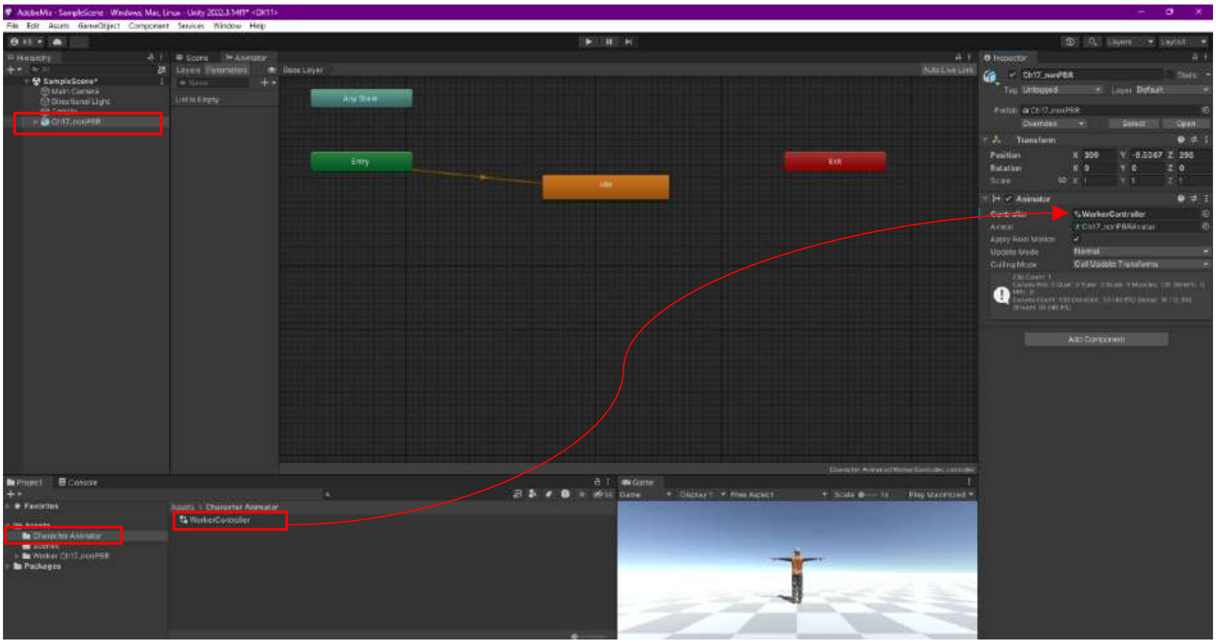
Drag and drop the **Idle** animation 

listed under the **Idle fbx** file to the animator editor. The **Idle** status will appear in the editor.

Also, a link will be automatically established from **Entry** to **Idle** status.

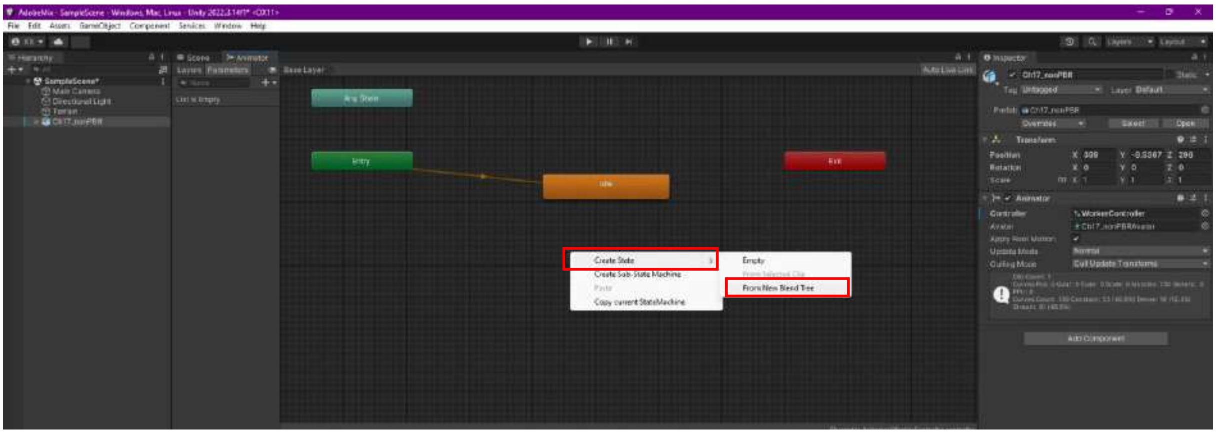


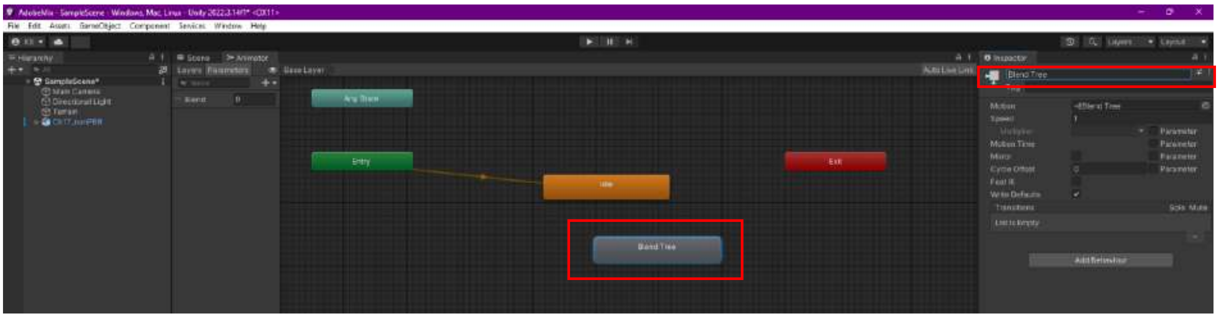
Another process is to connect the controller named **Assets>Character Animator>WorkerController** to the empty **Inspector>Animator>Controller** section of our **Ch17_nonPBR** object in **Hierarchy**.



When we switch to **Player Mode**, we see that our worker no longer stands in **T-Pose** and makes slight oscillations in the **Idle** position.

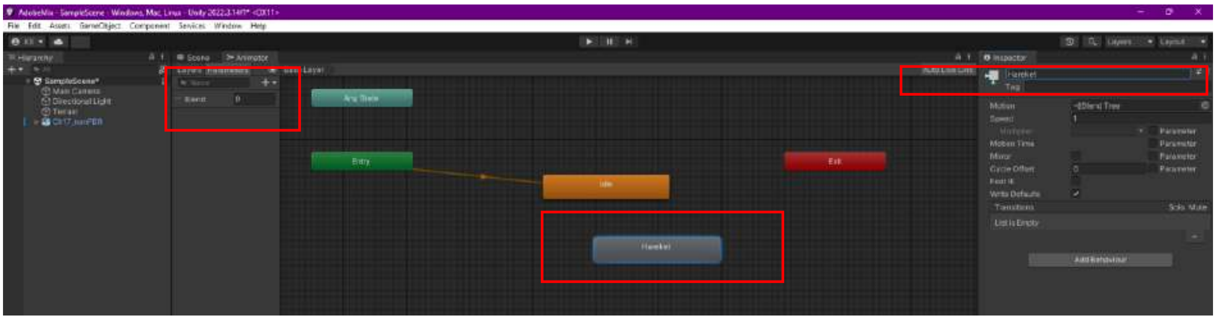
See the method that is normally used to establish a connection between states in the next step. Since the **walking** and **running** movements are connected and **transitional**, create a state that will **blend** them. **Right-click** in the editor and select **Create State>From New Blend Tree**.



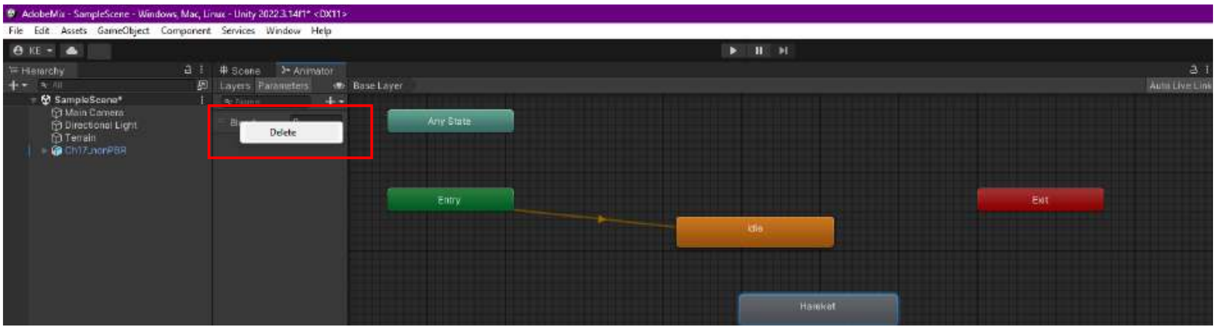


Change the name of this state to **Hareket**. The **Blend Tree** state can also be thought of as an organizer where a mix of movements can be made and passed.

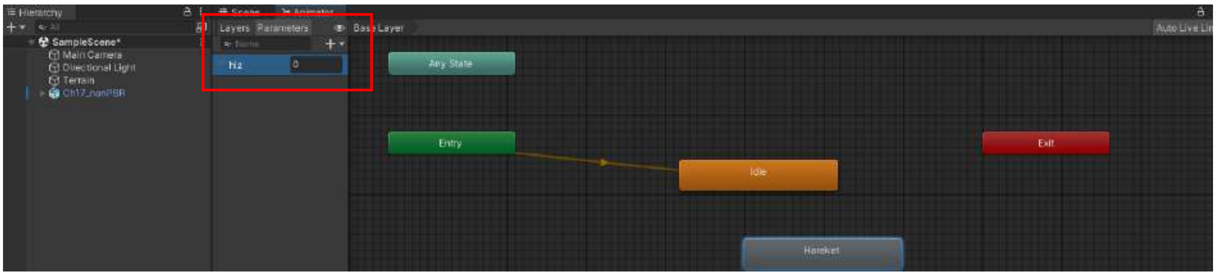
Another noteworthy point is that a parameter called **Blend** has been created in the **Parameters** section of the **Animator** tab.



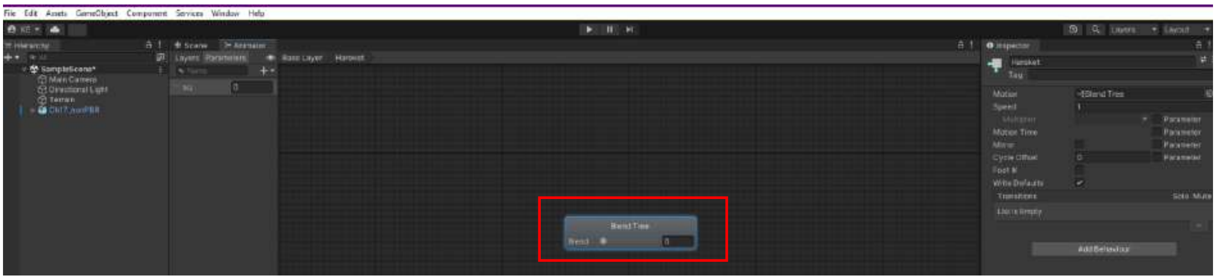
Right-click on the **Parameters>Blend** parameter and delete it with **Delete**.



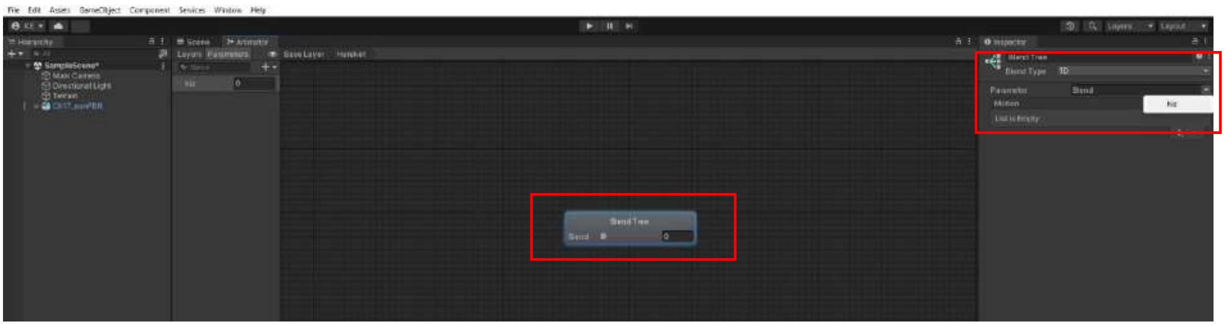
Instead, let's press **+** and add a **Float** type parameter, which is preferred in transition **animations** and which we will call **hiz**.



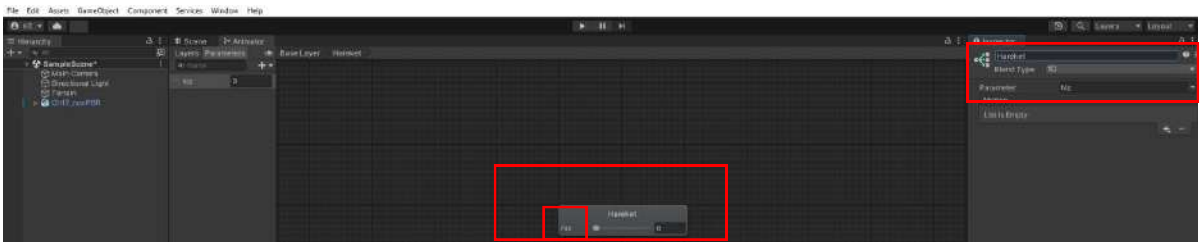
When we double-click on the movement, a layout and inspector will appear.



When the **Blend Tree box** is selected, **hiz** must be selected as the **Parameter**.

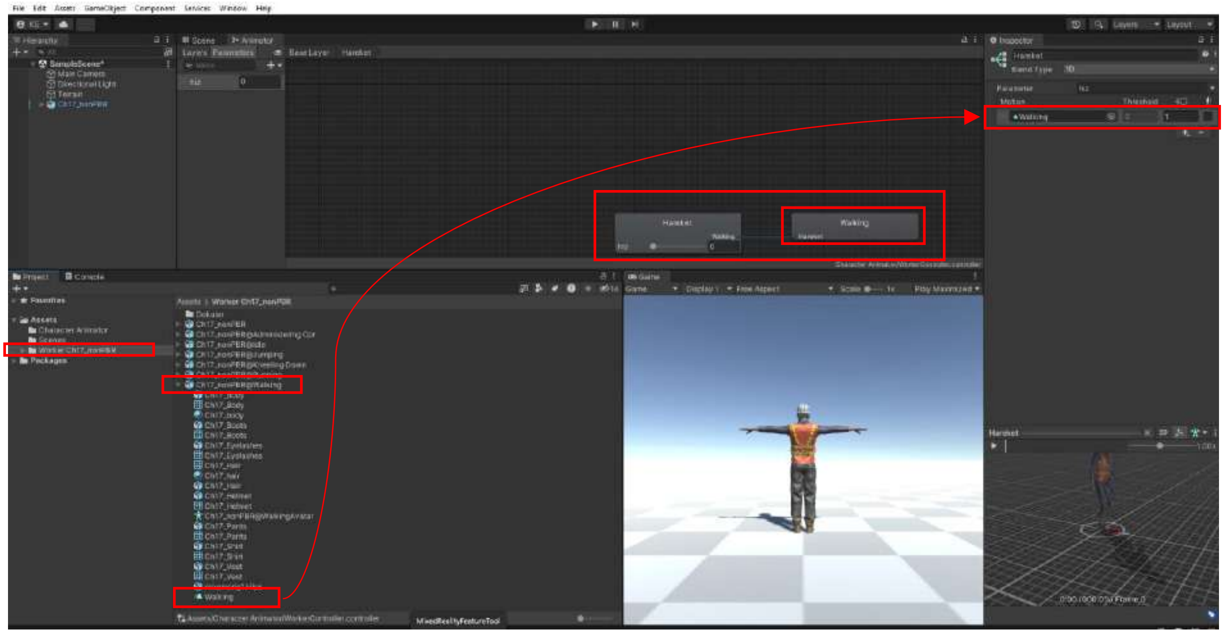


In this case, the adjustment line is also displayed in the **Blend Tree** box.

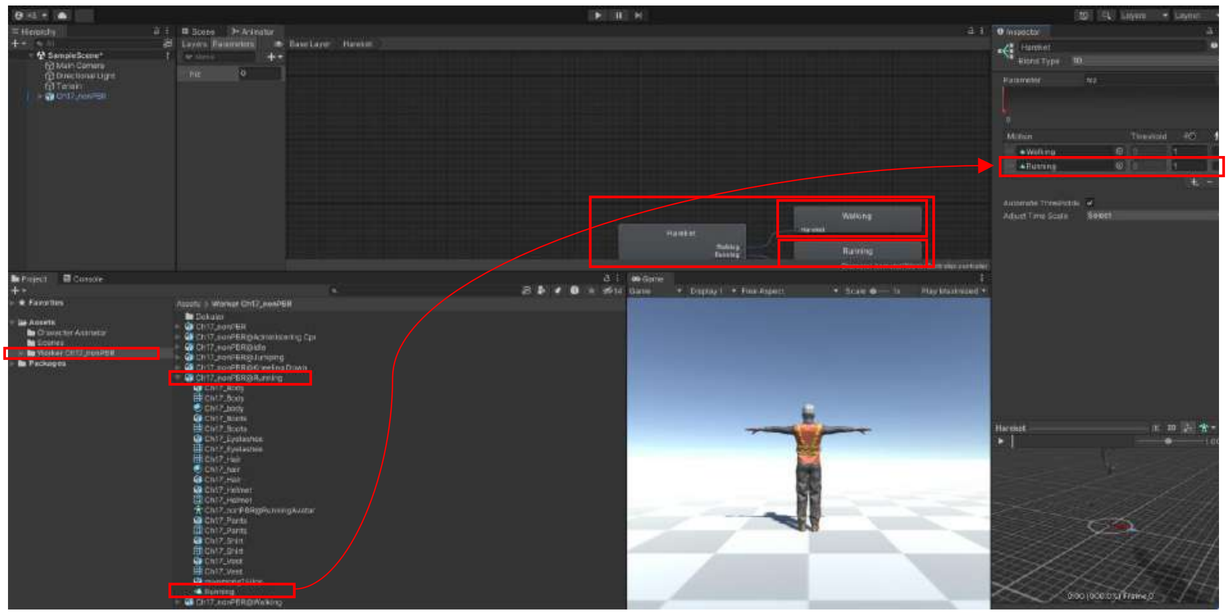


We can start adding motions to blend. Add a motion field with **Inspector>Motion>+>Add Motion Field**.

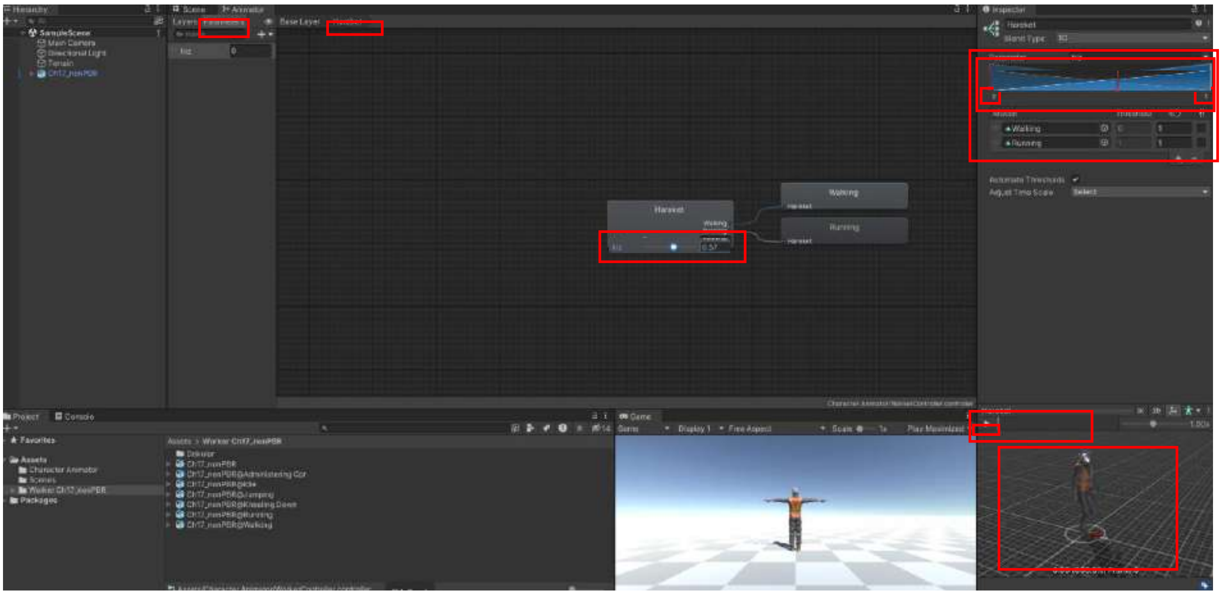
Let's **drag** the **Assets>Worker Ch17_nonPBR>Walking** animation to the opened area. This process also added the **Walking** animation to the **Blend Tree (Hareket)**.



Drag our other animation **Running** to **Inspector>Motion>+>Add Motion Field**.

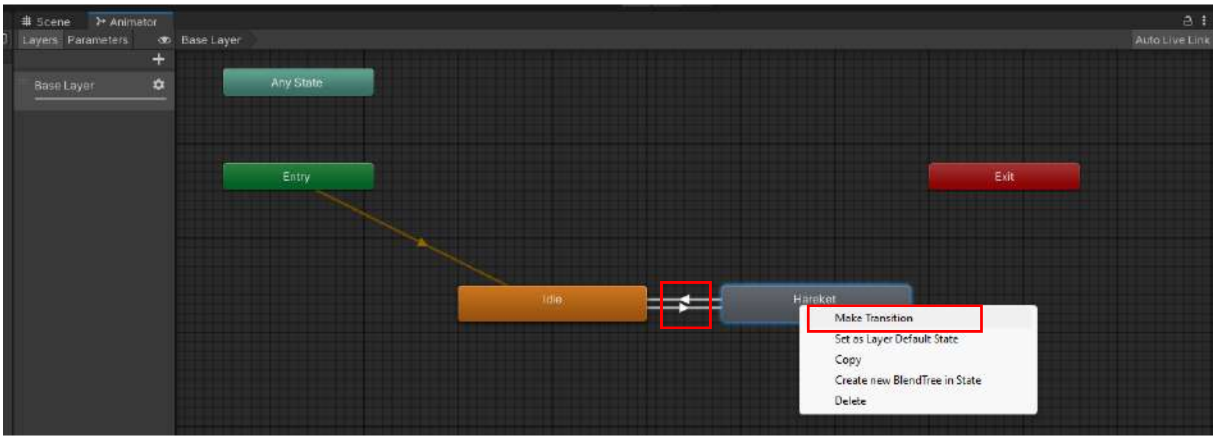


Here, the **Parameter** graph shows the **Walking-Running** transition **from 0 to 1**. Start the animation in the **Hareket** animation preview window below and test how it accelerates from **walking** to **running** by increasing the **hiz** setting on the **Hareket** box in the animator **from 0 to 1**.

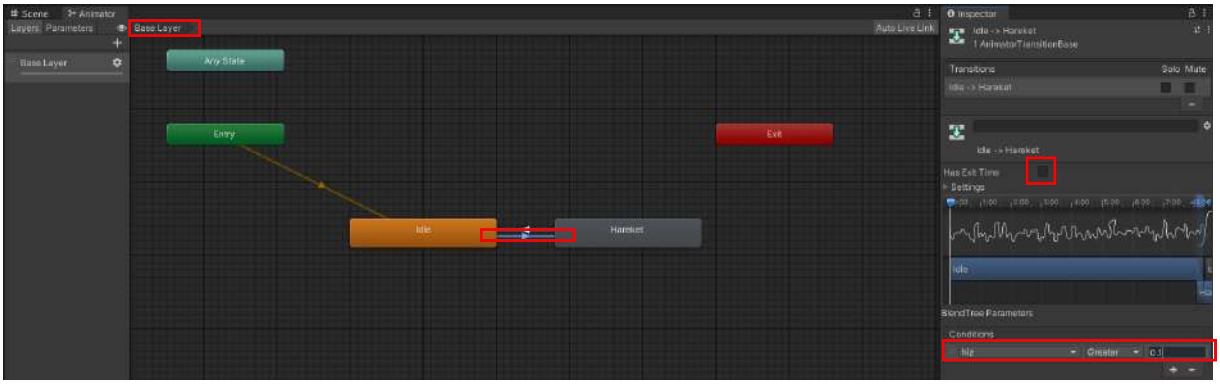


So, with **Blend Tree** we have achieved the **blending** and **transition** of two animations.

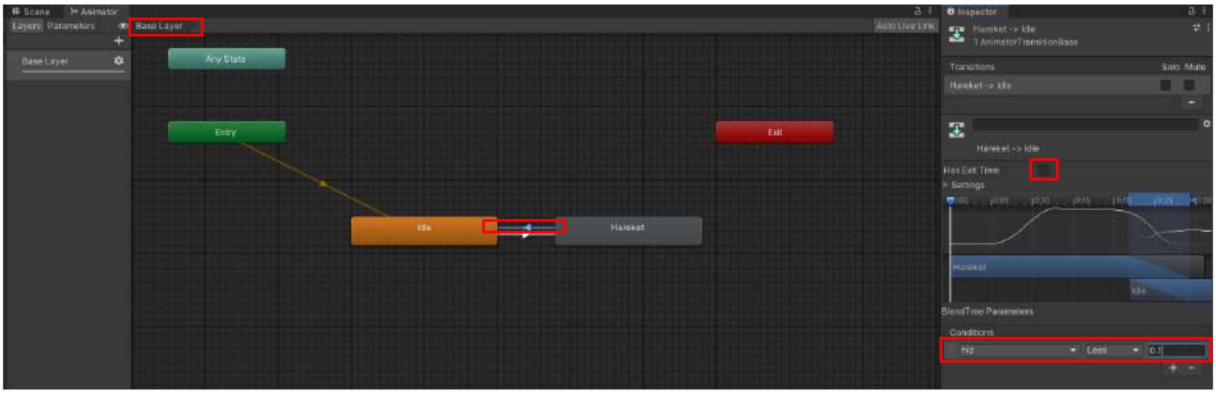
Now, go to the **Base Layer** and create a link between **Idle** and **Hareket** (**Blend Tree**). To do this, right-click on **Idle** and add a link to the **Hareket** box with **Make Transition**. Similarly, add a link to **Idle** from the **Hareket** box (**state**) with **Make Transition**.



Click on the direction arrow between **Idle** and **Hareket** to switch from **Idle** to **Hareket** (Blend Tree). In the Inspector section, **uncheck** the **Has Exit Time** box. We should add the condition that will provide the transition to the **Conditions** section. By pressing **+**, the number box where we will specify the number that is the **hiz** parameter and the **walking** condition [**Greater**] comes to the list. We can enter a value like **0.1** here.



For the worker to stop, the **hiz** parameter must be less than 0.1 (**Less**). For this,select the arrow in the opposite direction. Unselect the **Has Exit Time** box and add to **Conditions** with **+**. Here, select **Less** instead of **Greater** and write 0.1 in the **hiz** box.

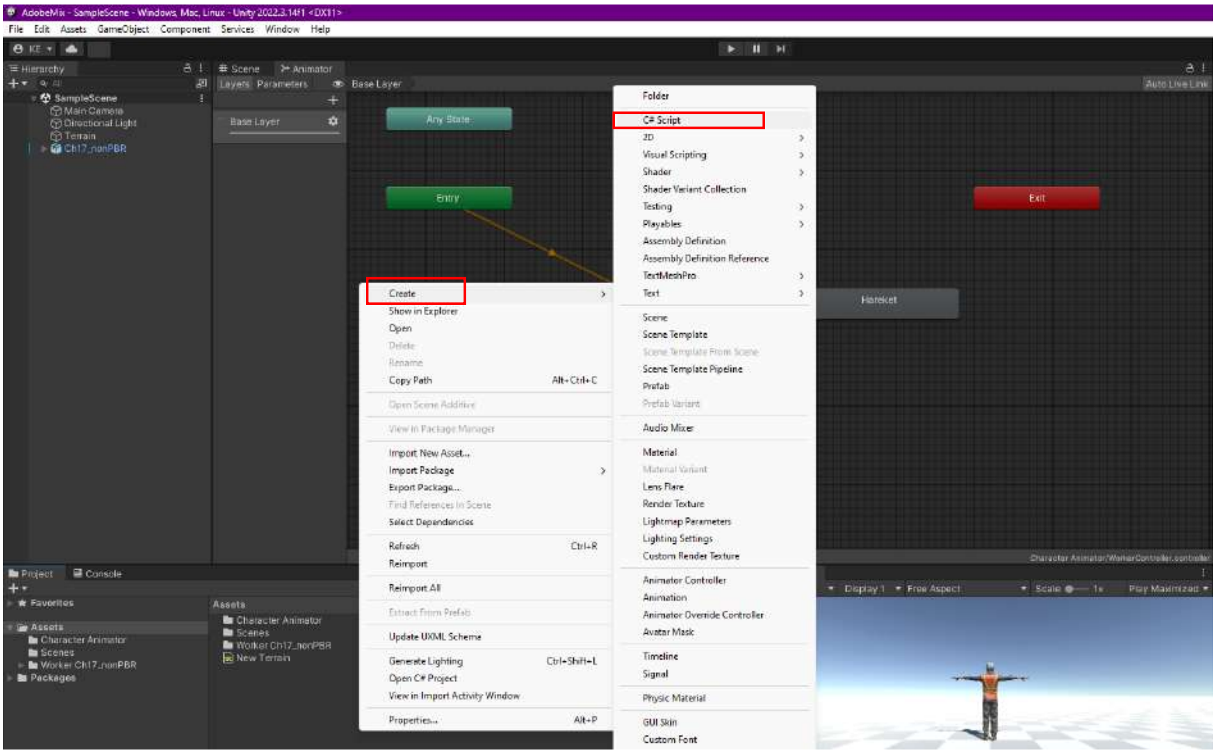


Thus, we have completed the necessary operations in the animator section. Now, when the speed (**hiz**) goes above 0.1, it walks (**Walking**); above 0.5, it starts running (**Running**); and below 0.1, it stops (**Idle**).

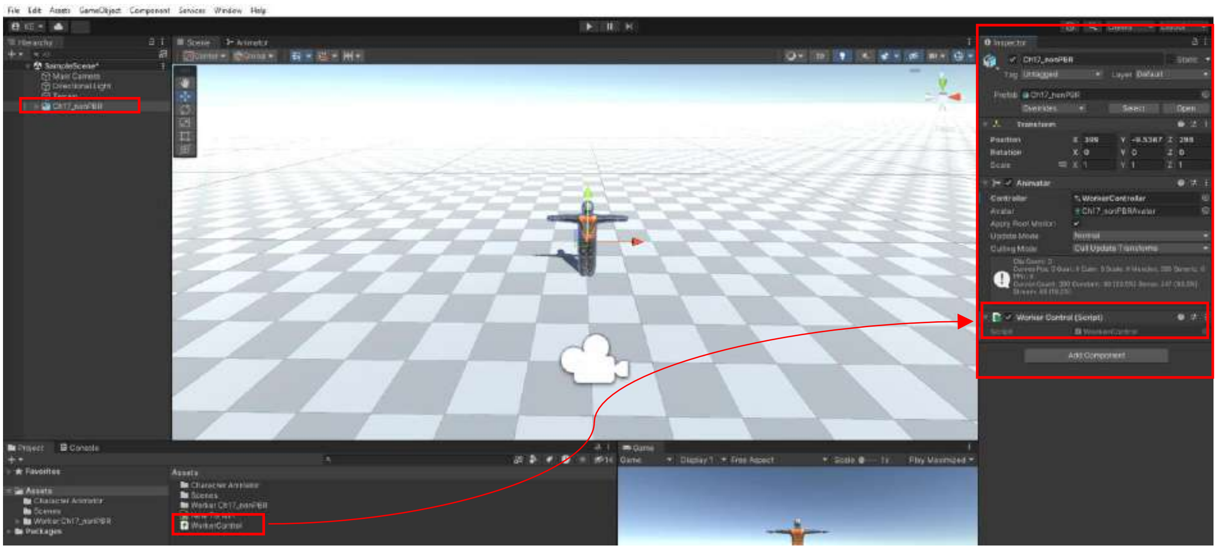
Now it's time for the keystroke to trigger these movement processes. Here, we need to provide **stop-walk-run** controls when a **key** is pressed on the **keyboard**. Therefore, we have to code the **C#** script.

Let's create a file under **Assets** with **Create>C# Script**. **WorkerControl** is used as the file name here.

Double-click on the **WorkerControl.cs** file created under **Assets** and open it in Visual Studio. If Visual Studio is not installed, it will open in the editor we specified as **Preferences>External Tools>External Script Editor** (example: Notepad).



Now, connect the script file we created to our worker on the scene.

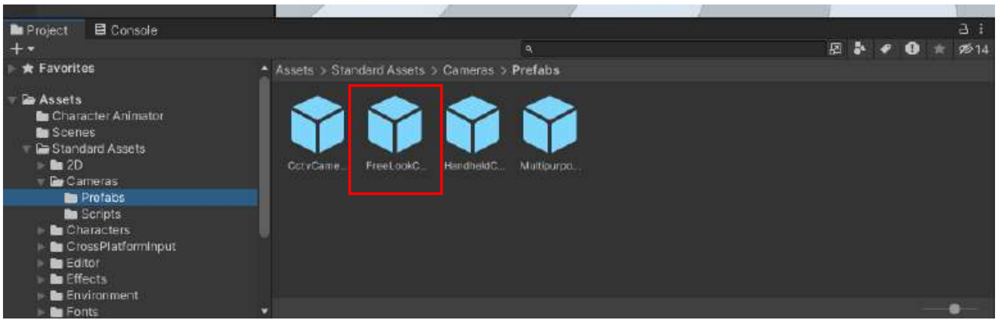


The **WorkerControl.cs** file content is given at the end of the topic. However, before using it, **Standard Assets** should be imported for camera control. This asset belongs to **Unity Technologies** and is not currently available in the Asset Store. However, it is available on the course's **Google Drive**.

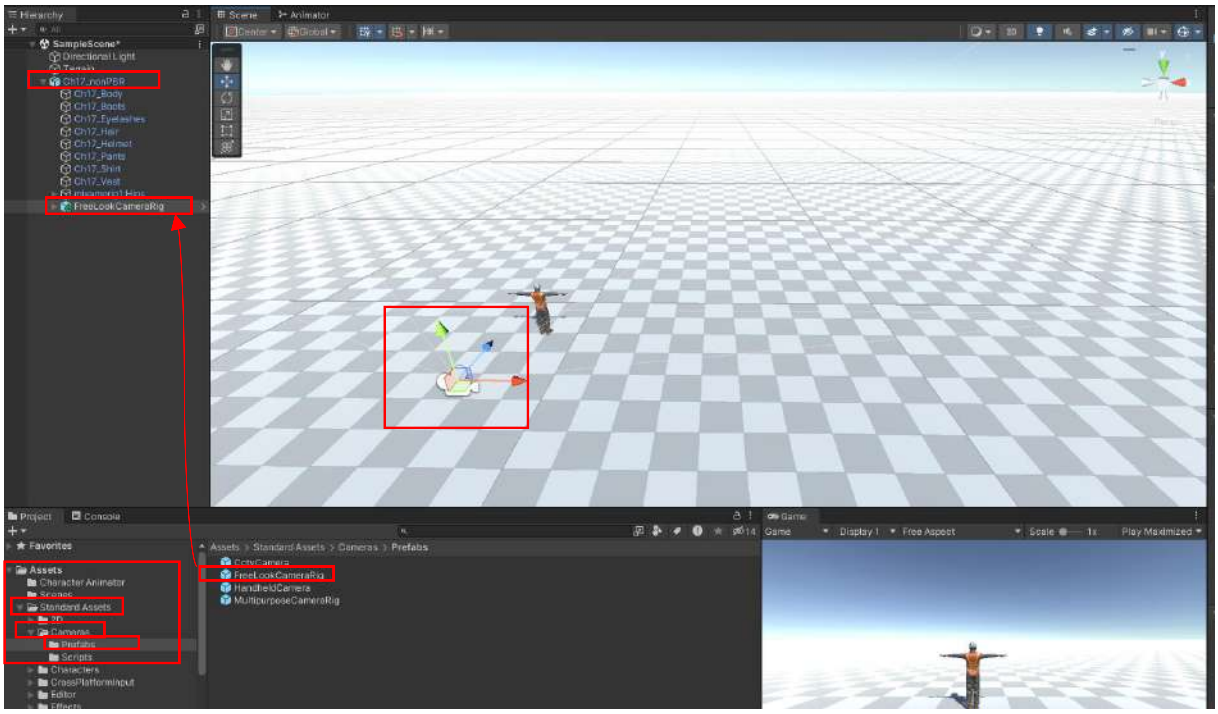
After the import process of the Standard Assets package is completed.

Assets>Standard Assets>Cameras>Prefabs>FreeLookCameraRig

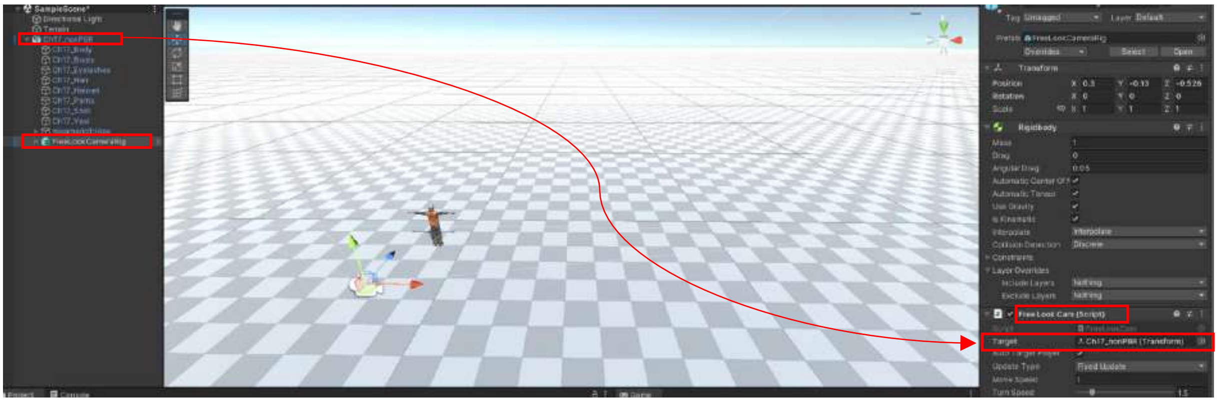
We will add the camera named above to our scene. The purpose here is to follow the worker's movements from different angles depending on the mouse.



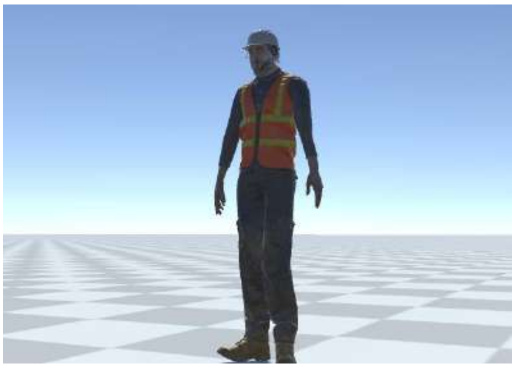
Connect this camera to our scene, to our **Ch17_nonPBR** worker object. Remove the **Main Camera** to avoid conflicts.



In order for the camera to follow the worker, the worker object (**Ch17_nonPBR**) in the **Hierarchy** must be connected to the **FreeLookCameraRig**'s **Inspector>Free Look Cam (Script)>Target**.



In this way, the camera can be rotated around the worker with mouse movements.



Another thing that is missing is the worker being able to turn in the direction the camera is pointed at and look at the camera.

This has been achieved by adding the **FreeLookCameraRig** camera and coding accordingly.

Before switching to play mode, it will work after advancing to the end of the topic, writing the code file completely or downloading it from **Google Drive** and connecting it to our worker object. It is important to follow the live narration in the course to avoid any problems.

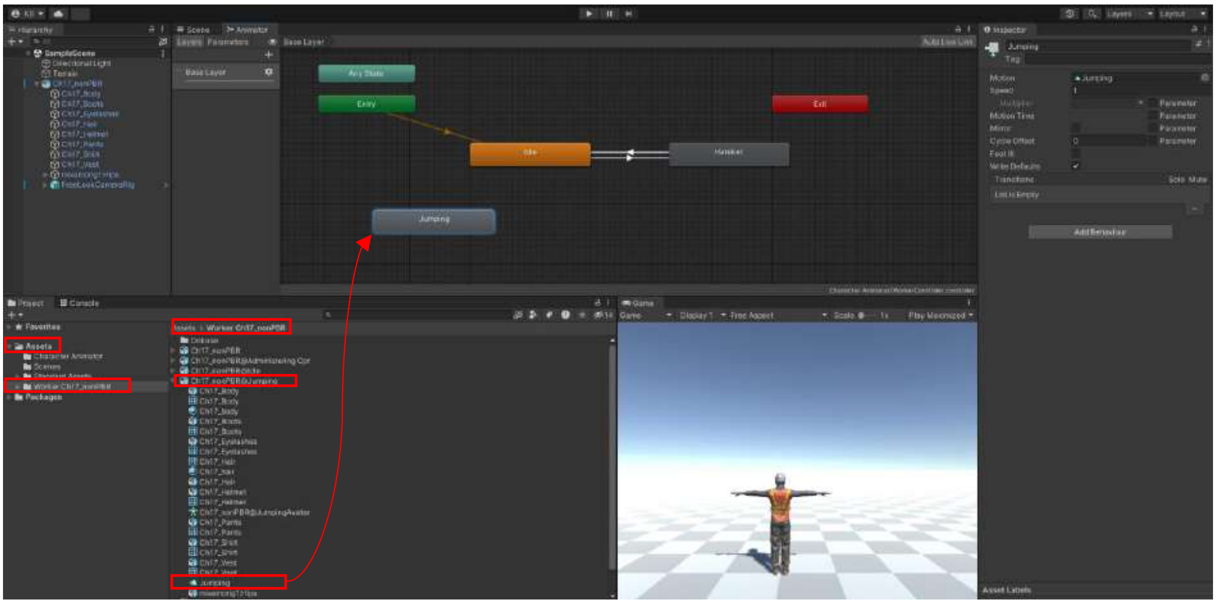
The operations performed up to this point were performed for **standing, walking** and **running** movements. Due to the transitional nature of **walking** and **running**, **Blend Tree** and **Float** were used as parameter type.

Now, see how to integrate some animations that we downloaded from **Mixamo**. Here, assuming that there is no transitional movement, we will cover the system binding of the **jumping**, **kneeling** and **administering Cpr** animations. These movements will be linked to the **Idle** position. **Jumping** and **kneeling** will be linked directly to **Idle**, while **Administering Cpr** will be linked to the **Kneeling** movement.

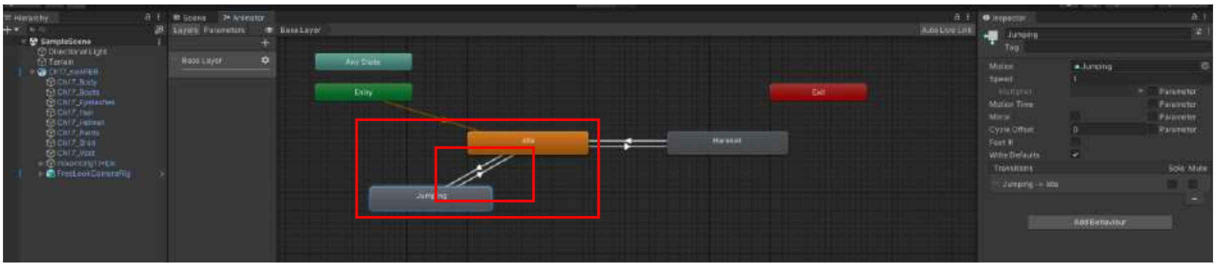
Let's start with **Jumping**.

Come to our animator window.

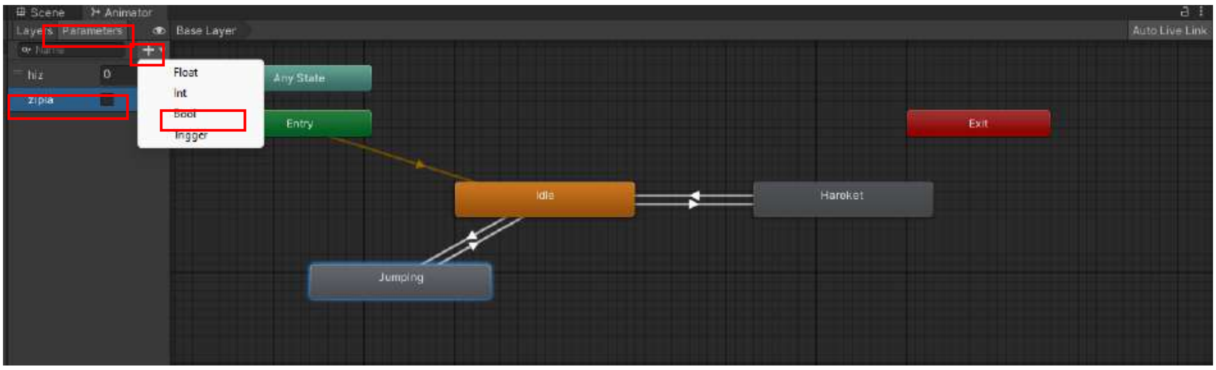
Drag the animation named **Jumping** under **Assets>WorkerCh17_nonPBR>Ch17_nonPBR>Ch17_nonPBR@Jumping** to the animator.



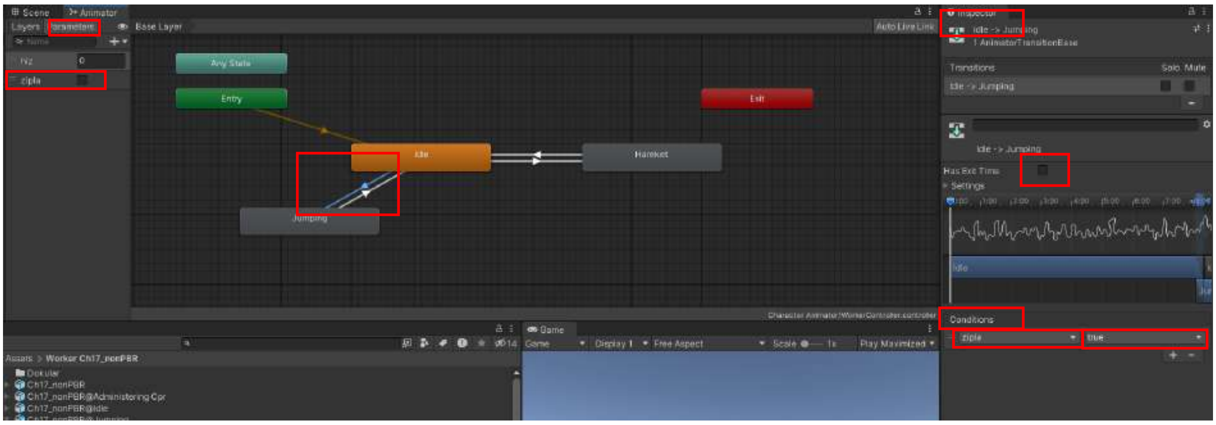
To establish the connection between **Idle** and **Jumping**, right click on **Idle** and use the direction line with **Make Transition**. Similarly, right click on **Idle** from **Jumping** and establish the connection line with **Make Transition**.



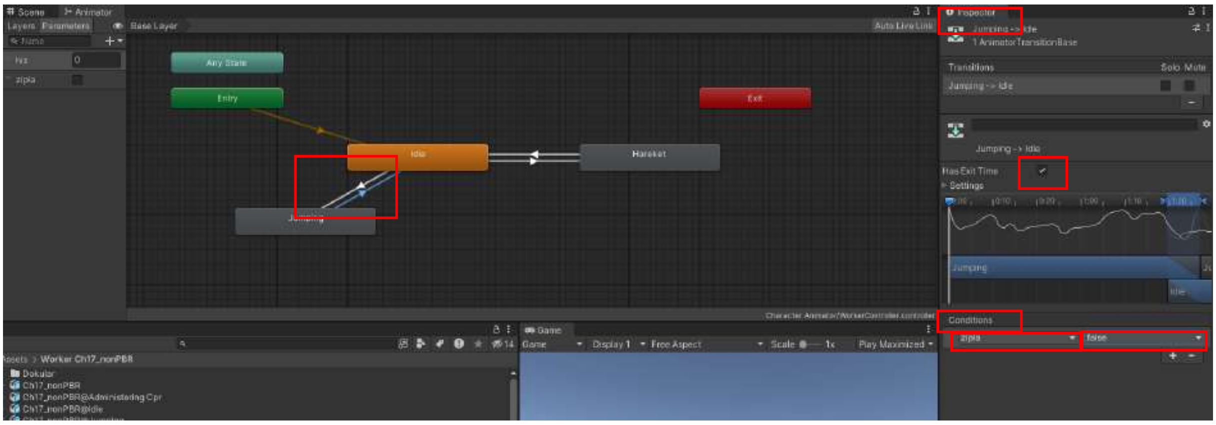
Now, we need to specify which parameter we will use to provide this connection. Add a **Bool** type parameter to the **Parameters** section with **+**. Here the name **zipla** is given.



At this stage, select the **Idle->Jumping** link. In the **Inspector**, unselect **Has Exit Time**. In **Conditions**, select **zipla** from the list by pressing **+**. We don't change the **true** selection because it will **jump** when the key is pressed.



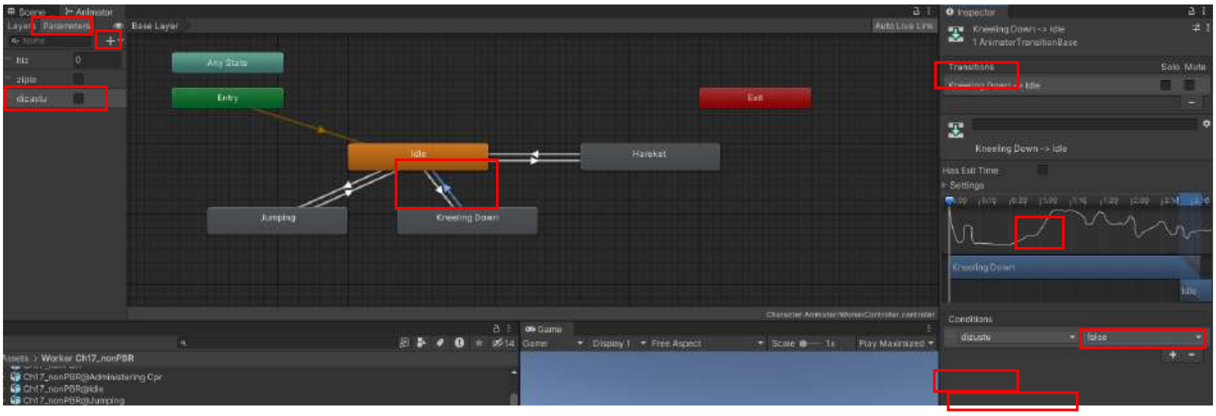
Similarly, select **Jumping->Idle** link. In the **Inspector**, unselect **Has Exit Time**. In **Conditions**, select **zipla** from the list that opens with **+** and set its condition to **false**.



Test this **animation**, which is connected to the **Space** bar key in the code line, by pressing the **Space** bar while in **Idle** mode in **Play** mode.

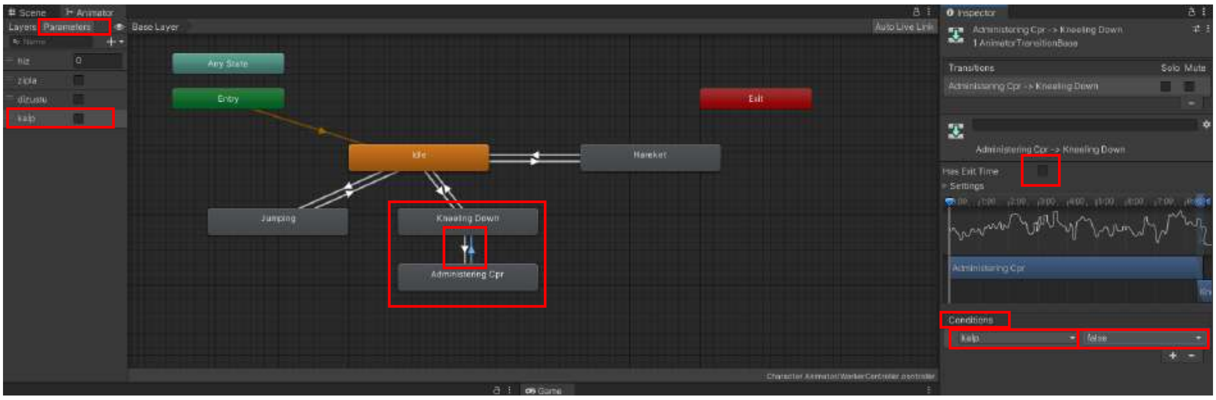
Similarly, add the **Kneeling Down** animation. Drag the relevant animation file from **Assets** to the **animator**. Establish the connections with **Idle**. **Add** the **Bool** type variable to the **Parameters** section.

Make the **true** and **false** selections by adding the **Has Exit Time** and **Conditions** lists in the connections.



In the coding, the **Z** key was used in the laptop collapse animation. When you press **Z**, the laptop will collapse and when you pull it, it will rise. After testing in player mode, let's move on to our other movement.

Here, connections were established between **Kneeling Down** and **Administering Cpr** with **Make Transition's**. **Bool** type **kalp** variable was added to **Parameters**. In the **Inspector**, **Has Exit Time** and **Conditions->kalp**; **true** and **false** operations were done with similar repetitions.



After setting the codes to press the **X** key, test it in **Play** mode.


12.1.Relevant C# Script Codes

Let's create a C# file named **WorkerControl.cs** under **Assets**. The file name should not be different from this.

The reason for this is that the file name and the **class** name inside must be the same.

public class WorkerControl : MonoBehaviour

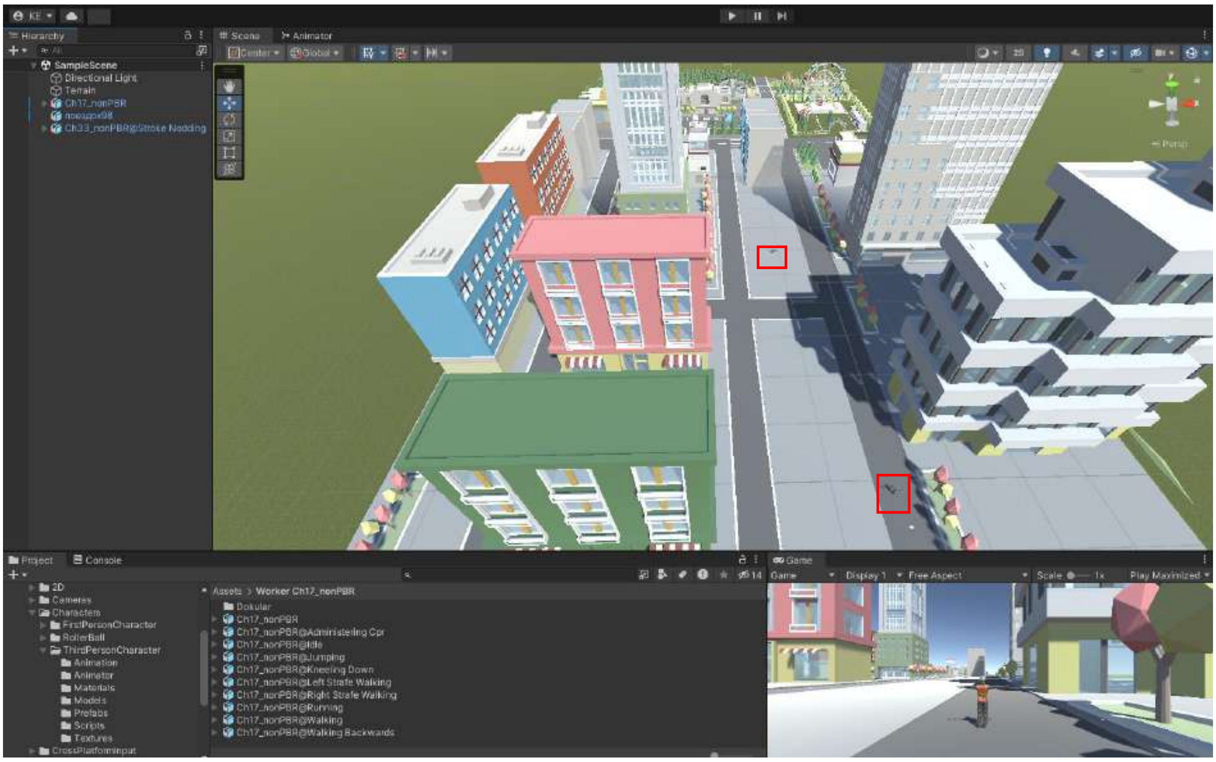
Codes can be created by writing them in the Visual Studio editor or an editor such as Notepad, or by copying/pasting the file content given below.

The more practical way is to use the ready-made version of the file.

The script file is also located in the **C# Script Files** folder of **Google Drive**, where the course documents are located.

After downloading this file from **Drive**, it can be placed in the Assets section of the project and linked to our **Ch17_nonPBR** object.

The scene was tested in this environment as well, by adding a low-poly city (from Sketchfab) and a casualty lying on the ground (from Mixamo) to make it an **emergency response** scenario for a casualty.



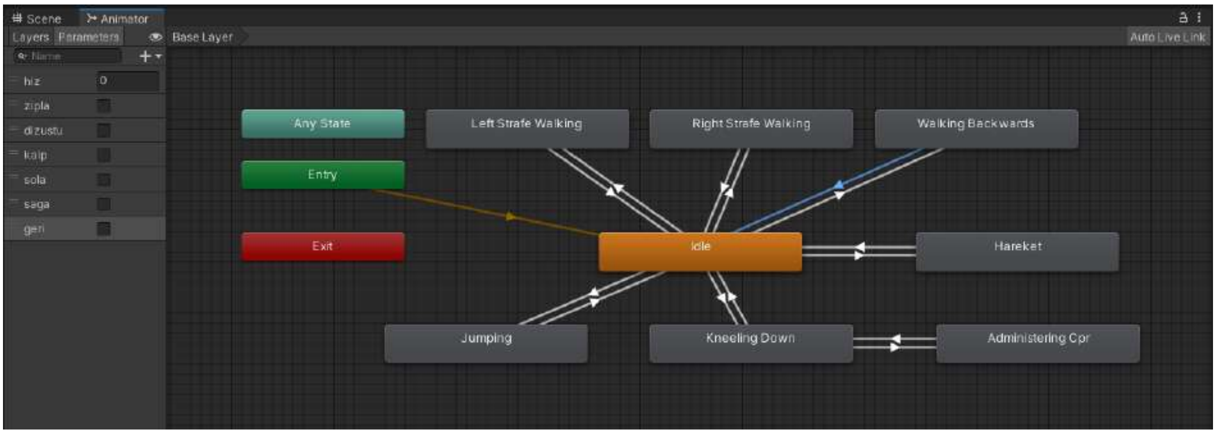
 

It is also possible to download animations for right, left and backward movements from Mixamo, add them to the animator and add coding.

Here;

A: left, **S:** back, **D:** right, **R:** 180^{o} rotation

It is coded to ensure orientation.



The final version of the **WorkerControl.cs** code file with these additions is below and on **Google Drive**.

```
/*** WorkerControl.cs ***/

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

public class WorkerControl : MonoBehaviour
{
    Animator adamAnim;
    float maxspeed;
    float axisZ;
    Camera mainCam;

    // Start is called before the first frame update
    void Start()
    {
```

```
    adamAnim = GetComponent<Animator>();
    mainCam = Camera.main;

}

// Update is called once per frame
void Update()
{
    if (Input.GetKey(KeyCode.W))    // walking
    {   maxspeed = 0.3f;
        axisZ = maxspeed * Input.GetAxis("Vertical");
     if (Input.GetKey(KeyCode.W) && Input.GetKey(KeyCode.LeftShift))  // running
        {   maxspeed = 1.0f;
            axisZ = maxspeed * Input.GetAxis("Vertical"); }
    }
    else    // standing
    {   maxspeed = 0.0f;
        axisZ = maxspeed* Input.GetAxis("Vertical");
    }
    // walking to right
    if (Input.GetKeyDown(KeyCode.D))
    { adamAnim.SetBool("saga", true); }
    if (Input.GetKeyUp(KeyCode.D))
    { adamAnim.SetBool("saga", false); }

    // walking to left
    if (Input.GetKeyDown(KeyCode.A))
    { adamAnim.SetBool("sola", true); }
    if (Input.GetKeyUp(KeyCode.A))
    { adamAnim.SetBool("sola", false); }

    // walking back
    if (Input.GetKeyDown(KeyCode.S))
    { adamAnim.SetBool("geri", true); }
    if (Input.GetKeyUp(KeyCode.S))
    { adamAnim.SetBool("geri", false); }

    // jumping
    if (Input.GetKeyDown(KeyCode.Space))
    { adamAnim.SetBool("zipla", true); }
    if (Input.GetKeyUp(KeyCode.Space))
    { adamAnim.SetBool("zipla", false); }

    // kneeling
    if (Input.GetKeyDown(KeyCode.Z))
    { adamAnim.SetBool("dizustu", true); }
    if (Input.GetKeyUp(KeyCode.Z))
    { adamAnim.SetBool("dizustu", false); }

    // heart massage-artificial respiration
    if (Input.GetKeyDown(KeyCode.X))
    { adamAnim.SetBool("dizustu", true);
      adamAnim.SetBool("kalp", true);   }
    if (Input.GetKeyUp(KeyCode.X))
    { adamAnim.SetBool("dizustu", false);
        adamAnim.SetBool("kalp", false);  }

    // to switch between the animations Vector3.ClampMagnitude can be used
    // to do this a variable like Vector3 is necessary; it is named (vektor) here

    Vector3 vektor = new Vector3 (0, 0, axisZ);

    adamAnim.SetFloat("hiz", Vector3.ClampMagnitude(vektor, 1f).magnitude, 1f, Time.deltaTime * 3f);
```

```
        //adamAnim.SetFloat("hiz", maxspeed);

        // to make the worker and the assigned Free Look Camera follow the mouse

        Vector3 kameraYon = mainCam.transform.TransformDirection(vektor);
        kameraYon.y = 0.0f;
        transform.forward = Vector3.Slerp(transform.forward, kameraYon, Time.deltaTime * 3f);

        //transform.forward = kameraYon;
    }
}
```

In this way, some animations required for a wounded intervention scenario have been fulfilled.

It should not be forgotten that a healthy project is more possible in a **classroom** environment and in live lessons. Because in this way, there is a chance to intervene in problems that may occur in the processes of the trainees.

BIBLIOGRAPHY

Abbaszadeh, S., and Rastiveis, H. (2017). "A comparison of close-range photogrammetry using a non-professional camera with field surveying for volume estimation," in ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, XLII-4/W4, (Heipke: International Society for Photogrammetry and Remote Sensing), 1–4. DOI: 10.5194/isprs-archives-XLII-4-W4-1-2017.

Aerial-Craft (2022). Rawang Mine Quarry, Malaysia. Retrieved from https://sketchfab.com/3d-models/rawang-quarry-2a-11jan17-malaysia-f3eae47abe694f36a9a98578315acc1e

Adams, D. M., Pilegard, C., & Mayer, R. E. (2016). Evaluating the cognitive consequences of playing portal for a short duration. Journal of Educational Computing Research, 54(2), 173–195.

Ahmad, S. M. S., Fauzi, N. F. M., Hashim, A. A., & Zainon, W. M. N. W. (2013). A study on the effectiveness of computer games in teaching and learning. International Journal of Advanced Studies in Computers, Science and Engineering, 2(1), 1.

Ahn, J., Ahn, E., Min, S., Choi, H., Kim, H., and Kim, G. J. (2019). "Size perception of augmented objects by different AR displays," in HCI International 2019—Posters, Vol. 1033, ed. C. Stephanidis (London: Intech Open), 337–344. DOI: 10.1007/978-3-030-23528-4_46

Al-Ansi, A.M., Jaboob, M., Garad, A., Al-Ansi, A., 2023. Analyzing augmented reality (AR) and virtual reality (VR) recent development in education. Social Sciences & Humanities, Volume 8, Issue 1, 1-10.

Aldrich, C. (2009). Virtual worlds, simulations, and games for education: A unifying view. Innovate: Journal of Online Education, 5(5), 1.

Alhalabi, W. S. (2016). Virtual reality systems enhance students' achievements in engineering education. *Behaviour & Information Technology*, *35*(11), 919-925. https://doi.org/10.1080/0144929X.2016.1212931.

Aloqaily, M, Bouachir, O and Karray, F, 2023. Digital twin for healthcare immersive services: fundamentals, architectures, and open issues, Digital Twin for Healthcare, Ed. El Saddik, A., Academic Press, 39-71, ISBN 9780323991636, https://www.sciencedirect.com/science/article/pii/B9780323991636000081.

American Society for Photogrammetry and Remote Sensing (2019). What is ASPRS?. Available online at: https://www.asprs.org/organization/what-is-asprs.html

Apple Vision Pro, 2024. https://www.apple.com/apple-vision-pro/

Atay, (2024). Atay Holding, https://www.atay.com.tr/atay-holding-alm-m.

Azuma, R.T., (1997). A survey of augmented reality. Presence: Teleoperators and Virtual Environments, 6(4), 355-385.

Bartlett, J. D., Lawrence, J. E., & Khanduja, V. (2018). Virtual reality hip arthroscopy simulator demonstrates sufficient face validity. Knee Surgery, Sports Traumatology, Arthroscopy. https://doi.org/10.1007/s00167-018-5038-8.

Beckem, J. M., & Watkins, M. (2012). Bringing life to learning: Immersive experiential learning simulations for online and blended courses. Journal of Asynchronous Learning Networks, 16(5), 61–70.

Bellanca, J. L., Orr, T. J., Helfrich, W. J., MacDonald, B., Navoyski, J., & Demich, B. (2019). Developing a virtual reality environment for mining research. 2019 SME Annual Conference and Expo and CMA 121st National Western Mining Conference, 597–606.

Bhai, R., (2024). Factory 3D Model, https://sketchfab.com/3d-models/factory-3d-model-e723f4fe48ec4b9da52ec6e4a442286b

Buckless, F. A., Krawczyk, K., & Showalter, D. S. (2014). Using virtual worlds to simulate real-world audit procedures. Issues in Accounting Education, 29(3), 389–417.

Capturingaworld, (2024). Roadstone Quarry, Allenwood, Sketchfab. https://sketchfab.com/3d-models/roadstone-quarry-allenwood-3d-mesh-1e14c3c1c67347d7b1e0de7eeb3de996.

Cheng, K.H. and Tsai, C.C. (2012). Affordances of augmented reality in science learning: Suggestions for future research. Journal of Science Education and Technology, 22, 449-462. DOI: 10.1007/s10956-012-9405-9.

Cohen, L., Manion, L., & Morrison, K. (2007). *Research methods in education*. London: Routledge.

DePorres, D., & Livingston, R. E. (2016). Launching new doctoral students: Embracing the Hero's journey. Developments in Business Simulation and Experiential Learning, 43(1), 121–128.
Engati, 2024. Augmented Reality application. https://www.engati.com/glossary/augmented-reality

El Jamiy, F., and Marsh, R. (2019). Survey on depth perception in head mounted displays: distance estimation in virtual reality, augmented reality, and mixed reality. IET Image Process. 13, 707–712. DOI: 10.1049/iet-ipr.2018.5920.

Erarslan, K., (2022). Augmented Reality Applications on Quarries and Mines, Journal of Scientific Reports-B, 3, June, 13-24.

Fedorko, G. (2021). Application possibilities of virtual reality in failure analysis of conveyor belts. Engineering Failure Analysis, 128(May), 105615. https://doi.org/10.1016/j.engfailanal.2021.105615

Fei, D., & Anbi, Y. (2011). Safety education based on virtual mine. Procedia Engineering, 1922-1926.

Fischer, X., (2024). Asperge Cavity full 3D Network and Landmarks, Sketchfab, https://sketchfab.com/3d-models/realistic-underground-basecave-40-46466ec0558945e9aac9dad15aaeb9f3.

Fonseca, H, (2024). Coal Mill, Grabcad, https://grabcad.com/library/13-coal-mill-department-1.

Fonstad, M. A., Dietrich, J. T., Courville, B. C., Jensen, J. L., and Carbonneau, P. E. (2013). Topographic structure from motion: a new development in photogrammetric measurement. Earth Surf. Process. Landforms 38, 421–430. DOI: 10.1002/esp.3366.

Foster, P. J., & Burton, A. (2006). Modelling potential sightline improvements to underground Mining vehicles using virtual reality. Institution of Mining and Metallurgy. Transactions. Section A: Mining Technology, 115(3), 85–90. https://doi.org/10.1179/174328606X128714

Franklin, G., (2024). Latform for Jaw Crusher, Grabcad, https://grabcad.com/library/plataform-for-jaw-crusher-1

Freina, L., & Ott, M. (2015). A Literature Review on Immersive Virtual Reality in Education: State of the Art and Perspectives. eLearning & Software for Education, (1).

Game Developer, 2024. https://www.gamedeveloper.com/game-platforms/exploring-the-pc-game-engine-landscape

Garcia, F., (2024). Platform for Jaw Crusher, Grabcad. https://grabcad.com/library/plataform-for-jaw-crusher-1

Gegenfurtner, A., Quesada-Pallarès, C., & Knogler, M. (2014). Digital simulation-based training: A meta-analysis. British Journal of Educational Technology, 45(6), 1097–1114. HTC Vive VR Headset. https://www.vive.com/us/

Geithner, S., & Menzel, D. (2016). Effectiveness of learning through experience and reflection in a Project Management simulation. Simulation & Gaming, 47(2), 228–256 (2016). Doi: 1046878115624312.

GEOPS U-Paris-Saclay, (2024). Malpierre Quarry, Sketchfab. https://sketchfab.com/3d-models/malpierre-34085fa5c4d04335882f25f3b4ee94e7

GeoWeek News, 2024. Hololens mining 3d data revolution. https://www.geoweeknews.com/news/hololens-mining-3d-data-revolution

Giovanello, S. P., Kirk, J. A., & Kromer, M. K. (2013). Student perceptions of a role-playing simulation in an introductory international relations course. Journal of Political Science Education, 9(2), 197–208.

Glen, I., (2024). A longwall coal mine layout with shearing machine, 3D Warehouse, https://3dwarehouse.sketchup.com/model/5dfdd8d8f02863aa873da0b62456d2f9/UG-Longwall-coal-mine-layout-with-shearing-machine-track-roof-support

Grabcad, (2024). https://grabcad.com

Gupta, P., 2021. VR simulation solutions for the mining industry-2. Tecknotrove. https://tecknotrove.com/newsletter/vr-simulation-solutions-for-the-mining-industry-2/

Gül, Y (2019). Açık Maden İşletmelerinde İnsansız Hava Aracı (İHA) Uygulamaları. Jeoloji Bülteni. Cilt 62, Sayı 1, 99 - 112, 01.01.2019. https://doi.org/10.25288/tjb.519506.

Gürer, S., Sürer, E., Erkayaoğlu, 2023. MINING-VIRTUAL: A comprehensive virtual reality-based serious game for occupational health and safety training in underground mines. Safety Science, 166, 1-13.

Gürer, S., 2021. Development of a Virtual Reality-Based Serious Game for Occupational Health And Safety Training in Underground Mining, Middle East Tech Univ, MSc Dissertation, 91 pg.

Huang, H.-M., Rauch, U., & Liaw, S.-S., 2010. Investigating learners' attitudes toward virtual reality learning environments: Based on a constructivist approach. *Computers and Education*, *55*(3), 1171-1182. https://doi.org/10.1016/j.compedu.2010.05.014.

Huang, Hsiu-Mei, & Liaw, S.-S., 2018. An Analysis of Learners' Intentions Toward Virtual Reality Learning Based on Constructivist and Technology Acceptance Approaches. *The International Review of Research in Open and Distributed Learning*, *19*(1). https://doi.org/10.19173/irrodl.v19i1.2503

Hugues, O., Gbodossou, A., & Cieutat, J.-M. (2012). Towards the Application of Augmented Reality in the Mining Sector: Open-Pit Mines. International Journal of Applied Information Systems, 4(6), 27–32. https://doi.org/10.5120/ijais12-450760

IoT & Industry 4.0, (2020). b.telligent. https://www.btelligent.com/en/portfolio/industry-40/

Kanani, H. (2019). AR and VR in Mining Industry : Transforming the Future. Plutomen, Oct.1, https://pluto-men.com/ar-and-vr-in-mining-industry-transforming-the-future/#

Kavanagh, S., Luxton-Reilly, A., Wuensche, B., & Plimmer, B. (2017). A Systematic Review of Virtual Reality in Education. *Themes in Science and Technology Education*, *10*(2), 85-119.

Kebo, V., & Staša, P. (2012). Mining processes control and virtual reality. Proceedings of the 2012 13th International Carpathian Control Conference, ICCC 2012, 274–277. https://doi.org/10.1109/CarpathianCC.2012.6228653

Kelly, l, 2022. The Drift: Device bringing augmented reality to the mining face. Northern Ontario Business. https://www.northernontariobusiness.com/industry-news/mining/the-drift-device-bringing-augmented-reality-to-the-mining-face-5772449

Kesim, M. & Özarslan, Y. (2012). Augmented reality in education: current technologies and the potential for education. Procedia - Social and Behavioral Sciences, 47, 297-302.

Martin, S., Diaz, G., Sancristobal, E., Gil, R., Castro, M., & Peire, J. (2011). New technology trends in education: Seven years of forecasts and convergence. Computers and Education, 57(3), 1893-1906.

Kim, H., & Choi, Y. (2019). Performance comparison of user interface devices for controlling mining software in virtual reality environments. Applied Sciences (Switzerland), 9(13). https://doi.org/10.3390/app9132584.

Kirner, T.G., Reis, F.M.V., & Kirner, C. (2012). Development of an interactive book with Augmented Reality for teaching and learning geometric shapes. Information Systems and Technologies (CISTI), 1-6.

Kunz, B. R., Wouters, L., Smith, D., Thompson, W. B., and Creem-Regehr, S. H. (2009). Revisiting the effect of quality of graphics on distance judgments in virtual environments: a comparison of verbal reports and blind walking. Atten. Percept. Psychophys. 71, 1284–1293. DOI: 10.3758/APP.71.6.1284.

Jones, R., & Bursens, P. (2015). The effects of active learning environments: How simulations trigger affective learning. European Political Science, 14(3), 254–265.

Kızıl, M. ve Joy, J. (2001). What can Virtual Reality do for Safety? University of Queensland, St. Lucia QLD.

Kızıl, M. S., Kerridge, A. P., and Hancock, M. G. (2004). Use of virtual reality in mining education and training. CRCMining Conference, Noosa Head, Queensland, Australia, 15-16 June 2004. Brisbane, Australia: Cooperative Research Centre - Mining (CRCMining).

Li M., Sun, Z., Jiang, Z., Tan, Z. ve Chen, J., 2020. A Virtual Reality Platform for Safety Training in Coal Mines with AI and Cloud Computing, Discrete Dynamics in Nature and Society, vol. 2020, ID 6243085, 7 pg.

Lin, T.-J., & Lan, Y.-J. (2015). Language learning in virtual reality environments: Past, present, and future. *Educational Technology and Society*, *18*(4), 486-497.

Martin, S., Diaz, G., Sancristobal, E., Gil, R., Castro, M., and Peire, J. (2011). New technology trends in education: Seven years of forecasts and convergence. Computers and Education, 57 (3), 1893-1906.

Michalak, D. (2012). Applying the augmented reality and RFID technologies in the maintenance of mining machines. Lecture Notes in Engineering and Computer Science, 1, 256–260.

Microsoft Hololens, 2024. https://www.microsoft.com/tr-tr/hololens

Moore, P., 2021. Immersive Technologies boosts worksite VR platform new mine standards training tool. International Mining. https://im-mining.com/2021/03/30/immersive-technologies-boosts-worksite-vr-platform-new-mine-standards-training-tool/

Njamga, K. (2024). Gravel pit. Grabcad, https://grabcad.com/kelean.njamga-1.

Nickel, C., Knight, C., Langille, A., & Godwin, A. (2019). How Much Practice Is Required to Reduce Performance Variability in a Virtual Reality Mining Simulator? Safety(5(2)), 18.

Oculus Quest 3 Metaverse, 2024. https://www.meta.com/quest/quest-3https://www.meta.com/quest/quest-3/

Outcropwizard, Bonn, Germany (2022). Grube Theresia, Morshausen, Germany. Retrieved from https://sketchfab.com/3d-models/ppc-kgale-2016-09-08-677a883d45ea48fbb974a0470f98a2ed

Özalp, R., 2024. Blender ile Açık Ocak Tasarımı, Proje Çalışması. GSF, DPÜ.

Patrimoine, L. D., (2024). Coal Mine Gallery, Sketchfab. https://sketchfab.com/3d-models/coal-mine-gallery-2cc9dfc11fe243039f9900f0c31414ae

Pine, J. 2018. 10 Real Use Cases for Augmented Reality AR is set to have a big impact on major industries. https://www.inc.com/james-paine/10-real-use-cases-for-augmented-reality.html

Premier Mapping, Cullinan, South Africa, (2024). PPC KGale Quarry, Botswana, Retrieved from https://sketchfab.com/3d-models/ppc-kgale-2016-09-08-677a883d45ea48fbb974a0470f98a2ed

Premier Mapping, Cullinan, South Africa, (2024). Lyttleton Quarry, New Zealand. Retrieved from https://sketchfab.com/3d-models/2016-09-01-lyttelton-05612c2bc3844d249b905a21f05aa594

Rice, R. (2009). The augmented reality hype cycle. http://www.sprxmobile.com/the-augmented-reality-hype-cycle /2014.

Rigmodels, (2024). https://rigmodels.com

Ritz, L. T., & Buss, A. R. (2016). A Framework for Aligning Instructional Design Strategies with Affordances of CAVE Immersive Virtual Reality Systems. *TechTrends*, *60*(6), 549-556. https://doi.org/10.1007/s11528-016-0085-9.

Rutten, N., van Joolingen, W. R., & van der Veen, J. T. (2012). The learning effects of computer simulations in science education. Computers & Education, 58(1), 136–153.

Samsung Gear VR, 2024. https://www.samsung.com/tr/support/model/SM-R322NZWATUR/

Sanlab, 2024. https://sanlab.net/simulator-systems/vr-heavy-equipment-simulator/

Sap, 2024. https://www.sap.com/mena/products/scm/industry-4-0/what-is-augmented-reality.html

Sarkar A. S., (2024), Flotation Machine, Grabcad. https://grabcad.com/library/flotation-machine-1

Sawyer, B. 2002. Serious games: Improving public policy through game-based learning and simulation. USA: Woodrow Wilson International Center for Scholars.

Scales, M., 2019. VIRTUAL REALITY: MacLean offers VR training for bolter operators.Canadian Mining Journal (CIM). May 9. https://www.canadianminingjournal.com/news/virtual-reality-maclean-offers-vr-training-for-bolter-operators/

Shih, Y.-C. (2015). A virtual walk through London: culture learning through a cultural immersion experience. *Computer Assisted Language Learning*, *28*(5), 407–428.

Siewiorek, A., Gegenfurtner, A., Lainema, T., Saarinen, E., & Lehtinen, E. (2013). The effects of computer-simulation game training on participants' opinions on leadership styles. British Journal of Educational Technology, 44(6), 1012–1035.

Sketchfab Inc. (2024). https://sketchfab.com

Smallman, H. S., and St John, M. (2005). Naive realism: misplaced faith in realistic displays. Ergon. Design Q. Hum. Fact. Appl. 13, 6–13. DOI: 10.1177/106480460501300303

Sony Morpheus, 2024. Playstation VR. https://www.playstation.com/en-us/ps-vr/

SPH Engineering, Riga, Latvia (2022). Retrieved from https://www.ugcs.com/news-entry/announcing-release-of-ugcs-update-with-added-search-patterns-for-sar-operations

Squelch, A. (2001). Virtual reality for mine safety training in South Africa. Journal of the Southern African Institute of Mining and Metallurgy, 209-216.

Stothard, P.M., Squelch, A.P., Van Wyk, E.A., Schofield, D., Fowle, K., Caris, C., Kizil, M., & Schmid, M. (2008). Taxonomy of Interactive Computer-based Visualisation Systems and Content for theMining Industry - Part 1. Proceedings of the AUSIMM Future Mining Conference 2008, Sydney.

Şimşek, İ. ve Can, T., 2019. Yüksek Öğretimde Sanal Gerçeklik Kullanımı ile İlgili Yapılan Araştırmalara Yönelik İçerik Analizi, Folklor/edebiyat, cilt:25, sayı: 97-1, 2019/1.

Tan, B., Zhu, H., Wang, N., Jiang, Y., & Jia, G. (2012). Application of virtual reality in fire teaching of mining. ICCSE 2012 - Proceedings of 2012 7th International Conference on Computer Science and Education, Iccse, 1079–1081. https://doi.org/10.1109/ICCSE.2012.6295250

Tichon, J. ve Burgess-Limerick, B., 2021. A review of virtual reality as a medium for safety related training in the minerals industry, *Journal of Health & Safety Research & Practice*, vol. 1, no. 3, pp. 33–40.

Tiwari, S. R., Nafees, L., & Krishnan, O. (2014). Simulation as a pedagogical tool: Measurement of impact on perceived effective learning. The International Journal of Management Education, 12(3), 260–270.

Toraño, J., Diego, I., Menéndez, M., & Gent, M. (2008). A finite element method (FEM) - Fuzzy logic (Soft Computing) - virtual reality model approach in a coalface longwall mining simulation. Automation in Construction, 17(4), 413–424. https://doi.org/10.1016/j.autcon.2007.07.001

Unity 3D (2024). Unity Technologies Inc. https://unity.com

Unreal Engine (2022). Epic Games Inc. https://www.unrealengine.com/en-US/

Valsev, (2024). Realistic Underground Cave, Sketchfab, https://sketchfab.com/3d-models/realistic-underground-basecave-40-46466ec0558945e9aac9dad15aaeb9f3

Van Krevelen, D.W.F. & Poelman, R. (2010). A Survey of Augmented Reality Technologies, Applications and Limitations. The International Journal of Virtual Reality, 9(2):1-20.

Vangorp, P., Laurijssen, J., and Dutré, P. (2007). The influence of shape on the perception of material reflectance. ACM Trans. Graphics 26:77. DOI: 10.1145/1239451.1239528

Vuforia, PTC Inc., (2024). https://www.ptc.com/en/products/vuforia

Wang, D., He, L., & Dou, K. (2013). StoryCube: supporting children's storytelling with a tangible tool. The Journal of Supercomputing. DOI: 10.1007/s11227-012-0855.

Wang, X., Kim, M.J., Love, P.E.D., and Kang, S.C. (2013). Augmented Reality in built environment: Classification and implications for future research. Automation in Construction, 32, 1–13.

Warehouse, (2024). https://3dwarehouse.sketchup.com

Wikipedia, 2024a. https://tr.wikipedia.org/wiki/Stereoskop

Wikipedia, 2024b. https://en.wikipedia.org/wiki/Google_Cardboard#/media/File:Google-Cardboard.jpg.

Wojciechowski, R., Walczak, K., White, M., & Cellary, W. (2004). Building virtual and augmented reality museum exhibitions. Proceedings of 9th international conference on 3D web technology (Web3D 2004), 135-144.

Xie, J., Li, S., & Wang, X. (2022). A digital smart product service system and a case study of the mining industry: MSPSS. Advanced Engineering Informatics, 53(216), 101694. https://doi.org/10.1016/j.aei.2022.101694

Xie, J., Liu, S., & Wang, X. (2022). Framework for a closed-loop cooperative human Cyber-Physical System for the mining industry driven by VR and AR: MHCPS. Computers and Industrial Engineering, 168(February), 108050. https://doi.org/10.1016/j.cie.2022.108050

Yılmaz, M.R., Göktaş, Y. (2018). Using Augmented Reality Technology in Education. Journal of Çukurova University Education Faculty, 47(2), 510-537.

Yin, C., Song, Y., Tabata, Y., Ogata, H., & Hwang, G. J. (2013). Developing and implementing a framework of participatory simulation for mobile learning using scaffolding. Educational Technology & Society, 16(2), 137–150.

Zhang, H. (2017). Head-mounted display-based intuitive virtual reality training system for the mining industry. International Journal of Mining Science and Technology, 27(4), 717–722. https://doi.org/10.1016/j.ijmst.2017.05.005

Zhang, X., Wang, A., & Li, J. (2011). Design and application of virtual reality system in fully mechanized mining face. Procedia Engineering, 26, 2165–2172. https://doi.org/10.1016/j.proeng.2011.11.2421

Zhou, Z., Cheok, A.D., Pan, J., & Li, Y. (2004). Magic story cube: An interactive tangible interface for storytelling. Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology, 364-365. DOI: 10.1145/1067343.1067404.

AFTERWORD

We have entered a big world with this book, which has a tutorial-based design for beginners of Unity. After combination of the knowledge of computer graphics, animation, C# codes, it is possible to deploy Unity projects to computers, mobile devices, consoles, VR headsets and AR smart glass platforms.

Being an expert in this huge volume of work and interdisciplinary work with field experts enables us to obtain satisfactory and efficient results. However, it is also possible to get results without the opportunity to work together with Unity experts with knowledge at a level that can meet the needs. **"Introduction to Unity, a Real-Time Development Engine *with applications for mining*"** serves to this idea.

Unity information will be further deepened with mathematical computations, numerical analysis, optimization, statistics, VR headsets such as Meta Oculus, HTC Vive and AR smart glasses such as MS Hololens 2, Apple Pro Vision. In addition, multi-user applications, integration of artificial intelligence programs, and its use with IoT Internet of Things are included in the Unity's future perspective.

If you've gotten this far, thank you. Good luck and wish you success.