Unity Lab

Remi, Lukas, Jan, Danvy, Hussein

2023-11-26

COMP 3504

### Introduction

In this activity, you will create a unity project, on Unity to run a 3D game which then can be implemented into augmented reality (AR)

### Prerequisites:

Download [Unity](https://unity.com/download) and [Microsoft Visual Studio](https://visualstudio.microsoft.com/thank-you-downloading-visual-studio/?sku=Community&channel=Release&version=VS2022&source=VSLandingPage&cid=2030&passive=false) and make sure you have Unity editor version: 2022.3.10f1(<https://unity.com/releases/editor/archive>)

Group discussions: Encourage participants to discuss their insights and learn from each other.

Feedback: Provide feedback and answer questions as needed.

### Setting up Project on Unity:

Step one: install unity hub and a version of unity. You can do this by going to unity hub and clicking installs on the left, clicking install editor in the top right corner. Install any of the official versions. The version we used was 2022.3.10f1.



Go back to projects(above installs). Click new project, 3D and Create Project.  
Wait like 3 mins

You will see something akin to this:



Main Unity Resources Used

There are 6(8) major tabs that you need to know about.

The first is the **Scene (1)** tab. It is in the middle of the screen and contains a view of a plane of the game world. This is where you can move and select objects within your project.

In the top left corner of the screen you can see 6 tools: View tool(good for looking around), move tool(good for moving objects), rotate tool(good for rotating objects), scale tool(good for making objects bigger or smaller), rect tool(good for shaping rectangles), transform tool(all the manipulation tools in one). You can press ctrl while manipulating to move or rotate in increments. This is useful when you want to move it 1m exactly or rotate 90 degrees exactly.

The **Game (2)** tab located right next to the Scene is a useful tab that shows you what someone would see in the game. This is just what the camera object sees.

The **Project (3)** tab (which we will be referring to as the **Asset (3)** tab from now on) at the bottom of your screen. It is where you will put any objects, scripts, scenes and other things you would like to keep track of during a unity project.

To add an object, there are 3 main ways of doing it:

1. Grab an FBX file, unity package, or any asset you want from your file explorer and drag it into the assets tab
2. At the bottom-middle area of the screen and located under the large Assets area (as seen with the single Scenes folder), right click and choose Import New Asset and navigate to the file you want to import.
   1. In the toolbar, you may also click “Assets” in the top left corner next to “File” and “Edit” to import the file

Try importing the Carl.fbx file provided along with the activity.

Creating a Script for a Game Object

If you want to create a script, right click in the Assets tab, select Create and click C# script (Highly recommended that you explore the rest of the create menu).

On the left side of your screen you will see a tab named **Hierarchy (4)**. This is where you can find all the items within the scene (hidden or not), such as your objects, cameras, etc.

Drag the Carl.fbx that you have imported into the Hierarchy to make it into an object. Double click the newly made Carl object and it will bring you to where Carl is in the Scene tab.

Select Carl and look at the right side of the screen you will see a tab named **Inspector (5)**, this is where you can find all the details about Carl. Currently there is only the transform component in the Inspector.

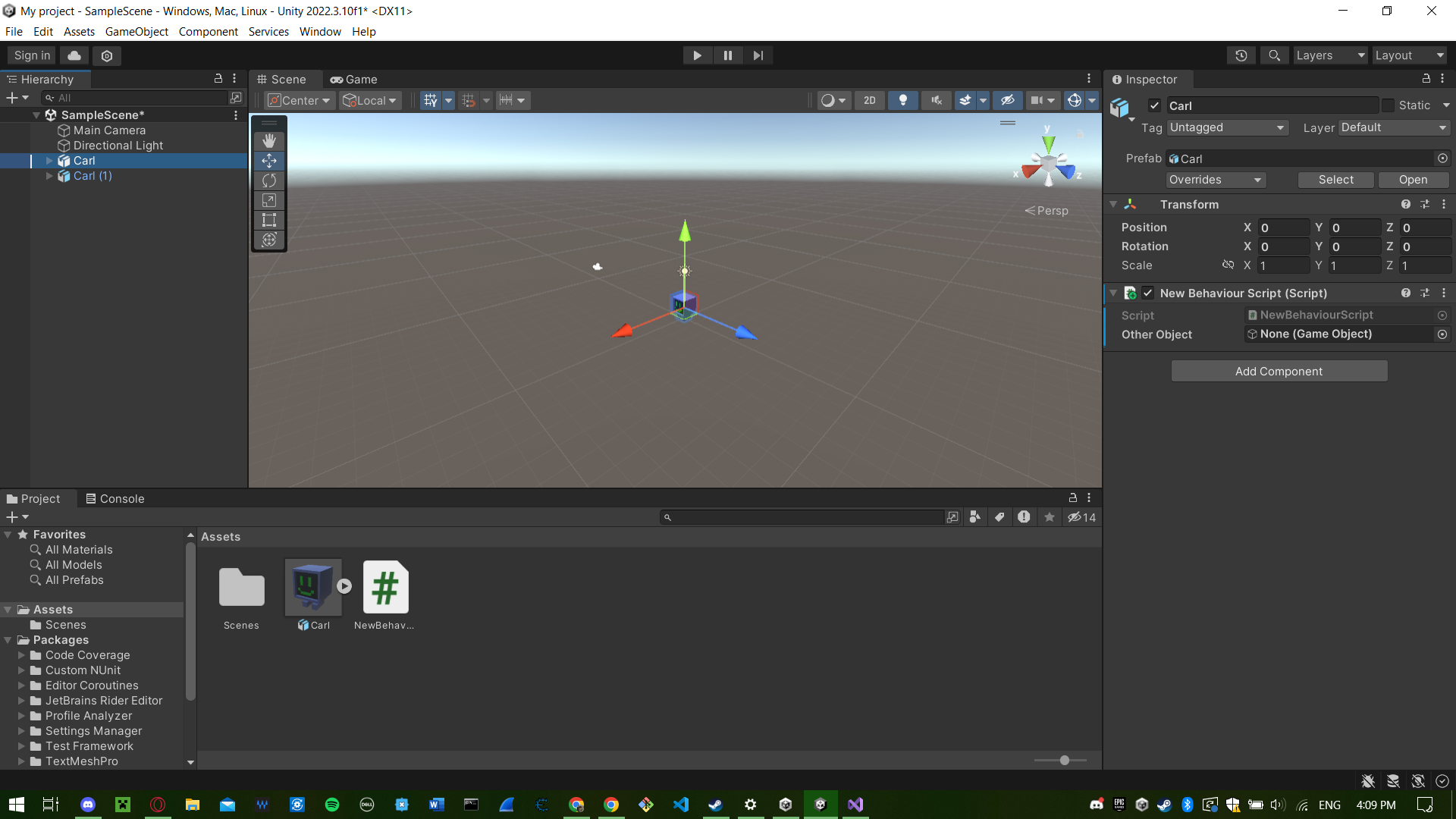
We will add a script to Carl. To do this, create a script like we showed before and name it anything. Now grab the new script and drag it onto Carl in either the Hierarchy or the Scene. Once you add this script, you can open and edit it by double clicking it in the asset tab.

Inside of the script, under public void start(), write “Debug.Log(“Hello World”);”.

Save it and now in the unity editor, look at the top middle of the screen (above “Scene” and “Game” tabs) and you will see a play button, a pause button and a step button. Click the play button.

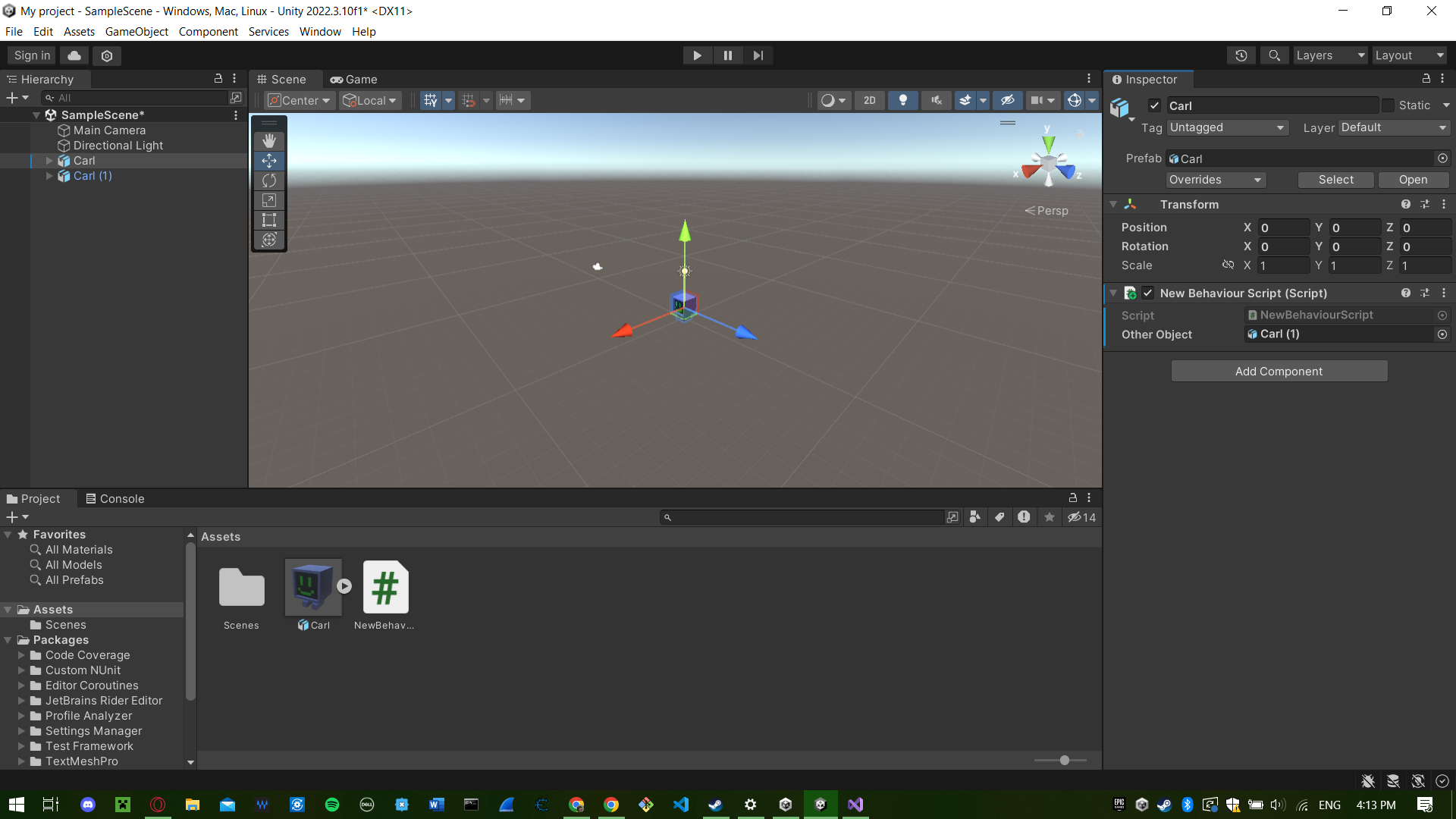
In the bottom window (the one with Assets), pick the **Console (6)** tab. You should now see a log saying “Hello World”.

At the top of the script class write “public GameObject otherObject;”

Go back to the unity editor, click on Carl, look at the Inspector tab and look at the script component you made. It should look like this:  


Now put another Carl object into the Scene tab. It should name itself Carl(1).

Drag Carl(1) from the Hierarchy tab into the “None (Game Object)” field under your script located in the Inspector tab.

It should now look like this:  
 

You have now made a reference object to another object in the scene.

Congratulations. You have learned the very basics of unity.

### 

### Activity Instructions:

For this activity you will create a script and edit the object you made to scroll across the screen.

Open the assets folder that you got from GitHub, The C Sharp files will tell you what needs to be done

Clone this repository it will contain everything you need to do this lab:

git hub repo: <https://github.com/theonlyhussein/UnityLabEX>

Go over Unity Scripting Basics.pdf file provided to learn Unity scripting basics. To learn more about scripting visit this website: <https://docs.unity3d.com/ScriptReference/>

There is a file on the github repo called Unity\_Scripting\_Basics\_1 or on page 10 of this document please follow along we will be referencing it in the instructions.

Step 1: In the GameObject Add component Rigidbody and turn off Gravity. Then create a script inside your game object.

Step 2 In the script create private variables to manage:

* The Rigidbody reference to the Rigidbody Component (p. 11)
* Movement between each time interval (p.14-15)
* Total time for the object to move (p.14-15)
* Timer to keep track of elapsed time (p.14-15)
* Block distance that the game object will move. (p.15-15)

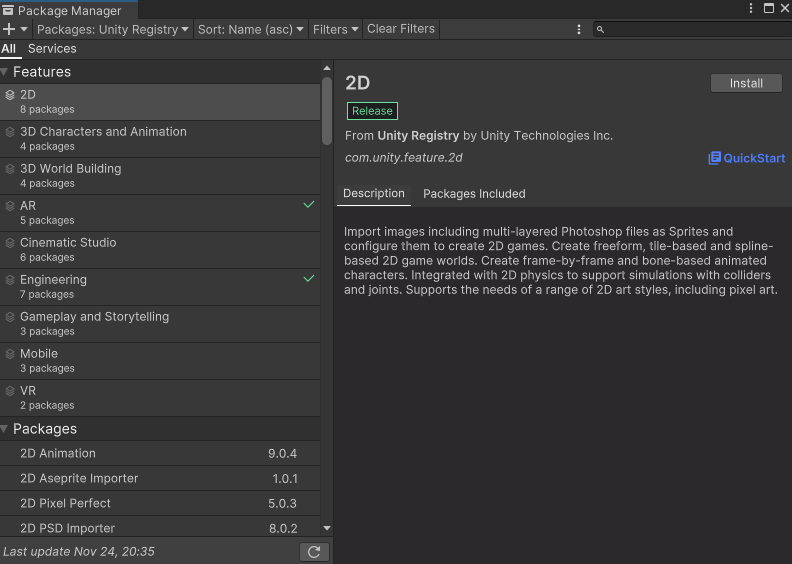
Step 3: Connect the GameObject to the Rigidbody in Start() (p.13)

Step 4: Update the timer for the amount of time it takes for each frame (p. 14)

Step 5: Use Vector3 to manipulate the GameObject by calculating the force needed to move the object forward. Add the Vector3 object to the Rigidbody object to apply the movement force. (p. 18)

Step 6: Set if statements to keep track of how long the script has run and when the next movement should take place.

### Setting up AR on Unity:

For implementing AR go to the Window tab and select packages manager make sure Packages tab is set to Unity Registry. 

Install AR packages wait for it to request reboot, and don’t forget to save (this may take awhile).

* Go to Player Settings:
* (You also can reach through edit -> project settings -> player)

If using iPhone select iOS tab



Under the other settings tab scroll down to Configuration then find Target minimum IOS version ans set to: 12.0 or higher

* Scroll up to Identification and check Override Default Bundle Identifier and type in Bundle Identifier: com.companyname.appname

If using Android select the android icon tab

:

* Under the other settings tab scroll down to Identification then type in package name: com.companyname.appname
* The select in minimum API level: Android 7,0 ‘Nougat’ (Api Level 24)

* Scroll up to Rendering and uncheck autographics API (if checked)
* Scroll Down to Configuration and select Scripting Backend IL2CPP.

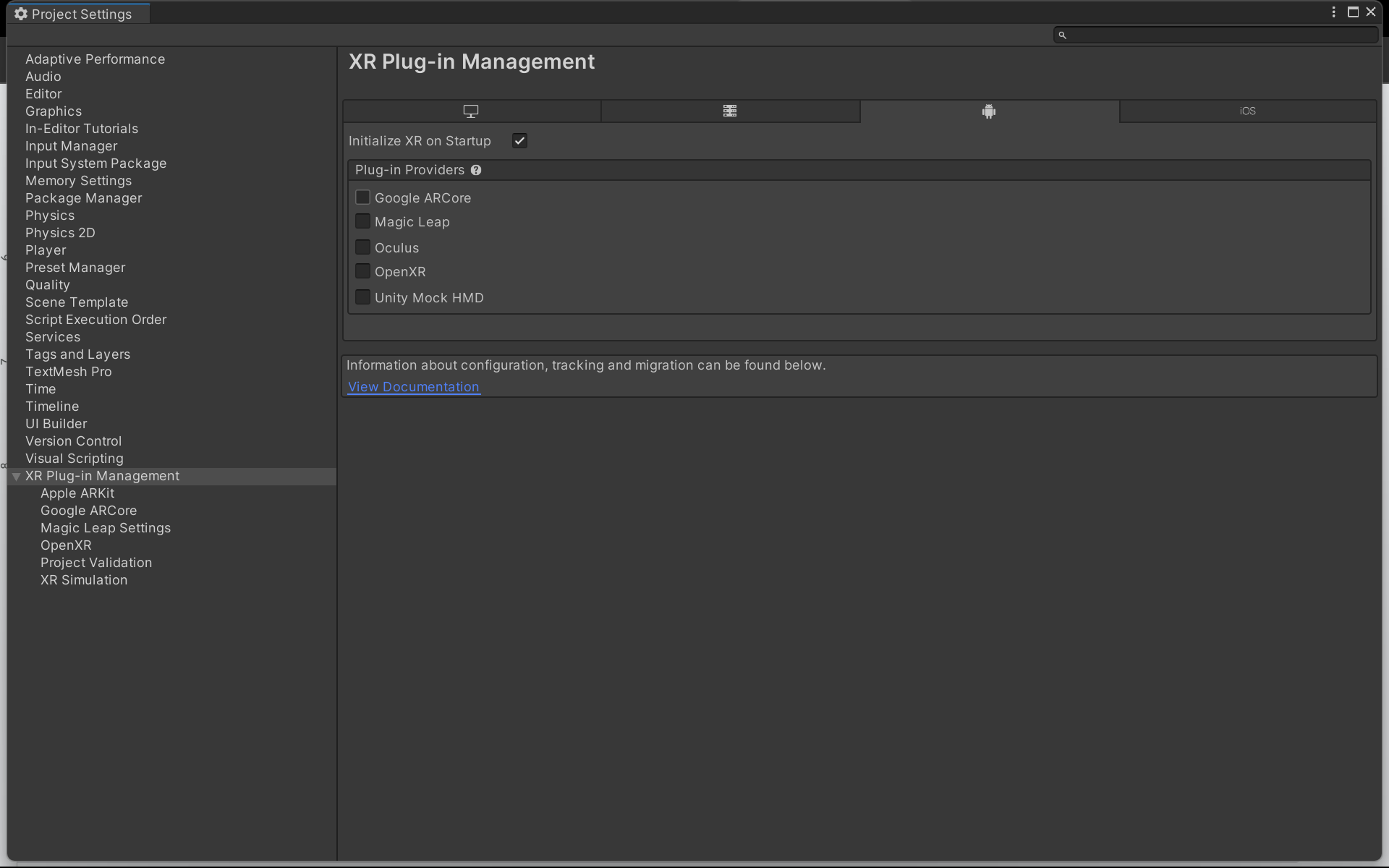
Plugin Management

1. Go to XR Plugin Management tab in Project Settings

If android check android logo tab:



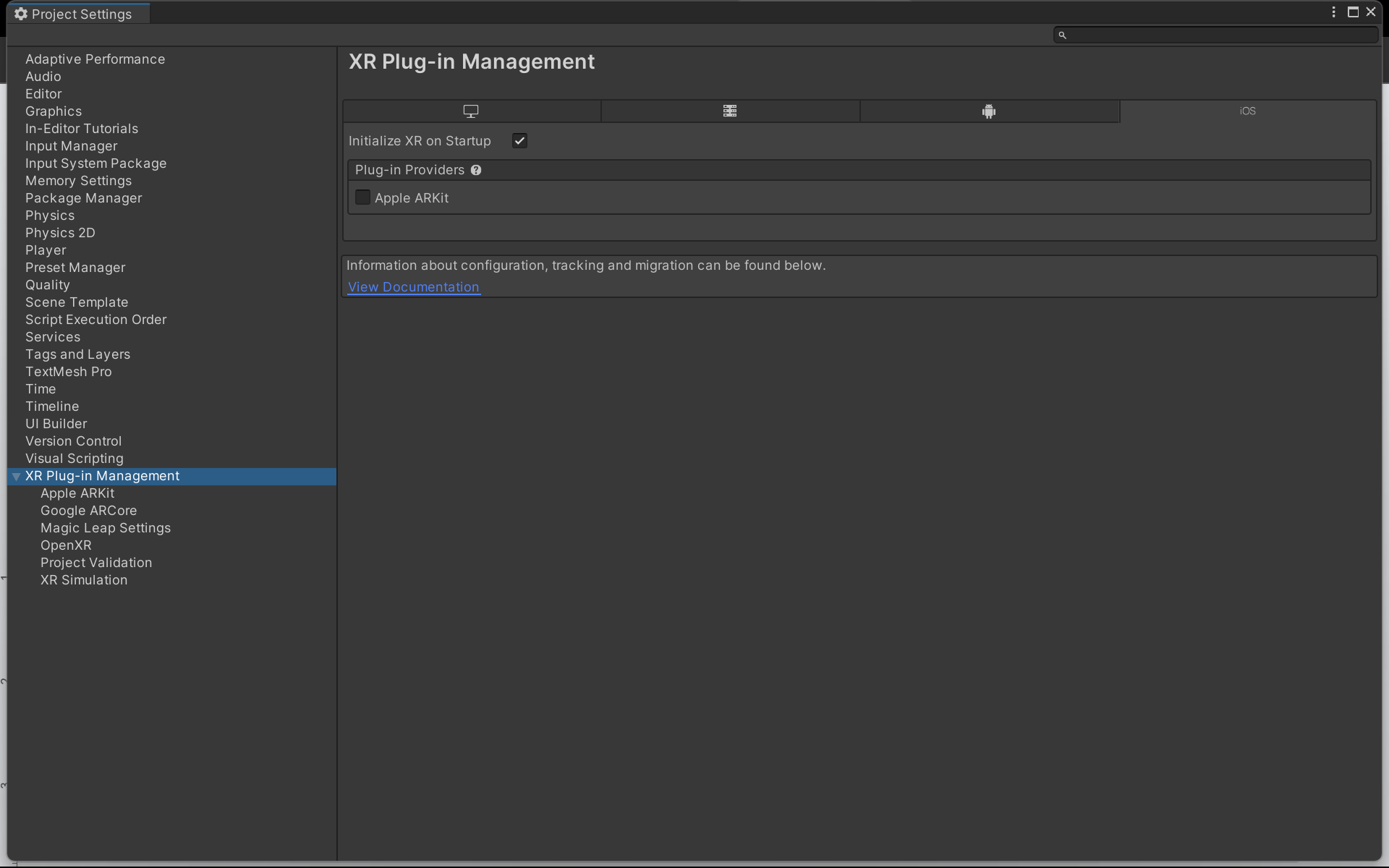
* Check off Google ARCore



If IO check iOS tab

:

* Check off ARKit



And it will install the package yippee

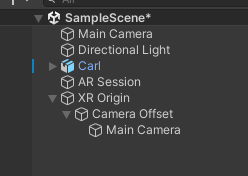
Go back to scene:

AR Session and AR Session Origin

1. The AR Session manages the lifecycle of the AR App.

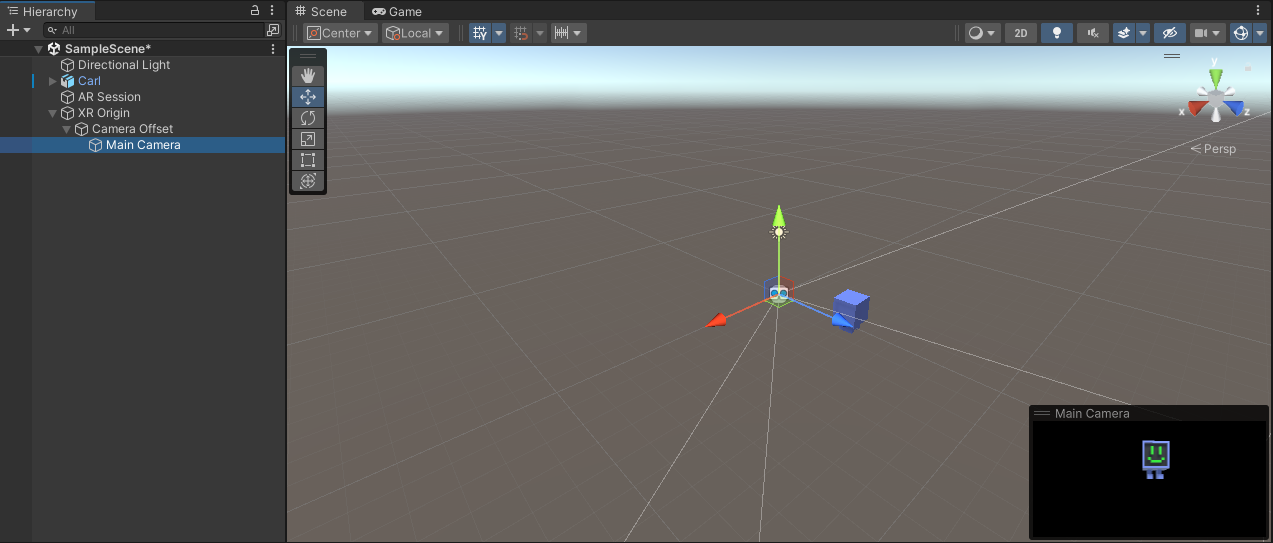
* Go to Hierachy tab right click and under XR click AR session to create a new game object

1. AR Session Origin takes AR Session space into Unity World Coordinates.

* Add game object XR origin under the XR object creator
* Under XR origin object there is a Camera Offset and under that is the Main Camera. If you still have the original Main Camera you can delete that.
* 
* Under the XR origin Camera there should be be 4 added components: Audio Listener, AR Camera Manager (Script), AR Camera Background (Script), Tracked Pose Driver (input System.
  + Under AR Camera Manager ensure Facing Direction is world, instead of user. This is the which camera on your phone will be used

1. And there you have it, now you’re done with the settings. Try adding something to the scene. Make sure you position it in front of the AR Camera.

* Go to the XR Main Camera and select to see the view of the camera.
* You should have your object created in the previous part, and by using the directional movement arrows move the game object into the view
  + If XR camera is selected a window will appear with the view ensure you see your game object in that window.



Deploying the Build:

For mac users wishing to deploy to IOS:

* Install Xcode if you don’t already have it: https://developer.apple.com/xcode/
* Open Xcode.
* Make sure you downloaded IOS build support at the beginning of the tutorial.

In Unity project:

* Go to File -> Build Settings -> Select IOS -> click add open scene -> click build -> make new folder called Builds and select it and it will start to build our scene into build folder.
* Connect your phone with a charger or something. And if app didn’t launch. Select the Xcode file in your Builds folder.
* Make sure developer mode on iphone is on. Change “Any IOS Device” on top tab to your phone. Press play.
* Final step: Trust app. On iPhone: go to General settings. Scroll down to VPN and Device management and select developement app and click trust.

For Android users:

* It’ll just build directly to your phone. Lucky you.
* Select your device under “run device” under Build Settings Then Press the play button.

Unity has some sample AR projects. Download from their GitHub: <https://github.com/Unity-Technologies/arfoundation-samples>

Congratulations you have successfully implemented AR from unity!

**Unity Script Basics**

1. **MonoBehaviour** is a fundamental class in Unity's scripting API. It acts as the base class from every Unity script derived. Scripts that are attached to GameObjects in Unity are typically subclasses of MonoBehaviour. This class provides access to many useful functions and events that are essential for game development, such as Start, Update, and many others.

Code Example:

using UnityEngine;

public class ExampleScript: MonoBehaviour

{

// Start is called before the first frame update

void Start()

{

Debug.Log("Hello, World!");

}

// Update is called once per frame

void Update()

{

// Code to be executed every frame

}

}

1. The **Start** Method is called once in the lifetime of the script, just before the first frame where the script is active. It's typically used for initialization tasks.

Code Example:

void Start()

{

Debug.Log("The game has started!");

// Initialization code here. For example, setting initial values.

}

1. **Rigidbody** in Unity is a component that allows a GameObject to be affected by the engine's physics system. It adds properties like mass, drag, and allows the object to react to forces, collisions, and gravity. This component is essential for simulating realistic movement and interaction with other objects in the game world.

Code Example:

using UnityEngine;

public class BallController: MonoBehaviour

{

Rigidbody rb;

void Start()

{

// Get the Rigidbody component

attached to this GameObject

rb = GetComponent<Rigidbody>();

}

void Update()

{

// Check for a key press

if (Input.GetKeyDown(KeyCode.Space))

{

// Apply an upward force to the Rigidbody

rb.AddForce(Vector3.up \* 5, ForceMode.Impulse);

}

}

}

/\*In this script:

* 1. Start Method: In the Start method, we get the Rigidbody component attached to the same GameObject as this script and store it in the rb variable.
  2. Update Method: In the Update method, we check if the spacebar key (KeyCode.Space) is pressed.
  3. Applying Force: If the spacebar is pressed, we apply an upward force to the Rigidbody using rb.AddForce(). The Vector3.up \* 5 part specifies the direction and magnitude of the force (upward in this case), and ForceMode.Impulse applies the force immediately, making it suitable for sudden impacts like jumps. \*/

1. A **GameObject** in Unity is the fundamental entity in a scene. It can represent characters, props, scenery, cameras, waypoints, and more. Essentially, a GameObject is a container for components such as meshes, colliders, scripts, and other Unity behaviors. Here are some simple examples to illustrate how GameObject is used in Unity scripting:

Code Example 1, Creating a GameObject in Code:

// Create a new GameObject with a name

GameObject myGameObject = new GameObject("MyGameObject");

// You can now add components to this GameObject, like a Rigidbody or a custom script

myGameObject.AddComponent<Rigidbody>();

//This code creates a new GameObject named "MyGameObject" and adds a Rigidbody component to it, making it subject to physics.

Code Example 2, Accessing a GameObject by Name:

// Find a GameObject in the scene by its name

GameObject foundGameObject = GameObject.Find("MyGameObject");

// You can now manipulate this GameObject, like changing its position

foundGameObject.transform.position = new Vector3(0, 0, 0);

//Here, GameObject.Find is used to find a GameObject named "MyGameObject" in the scene. Once found, its position is set to the origin (0,0,0).

Code Example 2, Attaching a Script to a GameObject:

public class MyScript : MonoBehaviour

{

void Start()

{

// Access the GameObject that this script is attached to

GameObject thisGameObject = this.gameObject;

// Do something with the GameObject, like changing its color

Renderer renderer = thisGameObject.GetComponent<Renderer>();

renderer.material.color = Color.blue;

}

}

//In this script, this.gameObject refers to the GameObject to which MyScript is attached. The script then changes the color of the GameObject to blue.

1. **Vector3** in Unity is a structure that represents a three-dimensional vector or point in space, used for storing positions, rotations, and scales. It contains three float components: x, y, and z.

Code Example:

using UnityEngine;

public class ExampleScript : MonoBehaviour

{

void Start()

{

// Create a new Vector3 to represent a position.

Vector3 startPosition = new Vector3(0, 0, 0);

// Set the position of this GameObject to startPosition.

transform.position = startPosition;

// Move the GameObject by adding to its current position.

Vector3 movement = new Vector3(1, 0, 0); // Move 1 unit along the x-axis.

transform.position += movement;

}

}

/\*In this Script:

* + 1. A Vector3 named startPosition is created and set to (0, 0, 0), representing the origin in 3D space.
    2. The transform.position of the GameObject this script is attached to is set to startPosition.
    3. Another Vector3 named movement is created, representing a movement alonthe x-axis.
    4. The GameObject's position is then updated by adding movement to its current position, effectively moving it 1 unit along the x-axis. \*/

1. The **Transform** component in Unity is a fundamental part of every GameObject. It represents the object's position, rotation, and scale in the game world. Here's a simple explanation of how Transform is typically used in Unity scripts.

Code Examples:

using UnityEngine;

public class ExampleScript : MonoBehaviour

{

void Update()

{

// Move the GameObject 1 unit upwards every frame

transform.position += new Vector3(0, 1, 0) \* Time.deltaTime;

// Rotate the GameObject by 30 degrees around the y-axis every second

transform.Rotate(new Vector3(0, 30, 0) \* Time.deltaTime, Space.World);

}

}

/\* In this

1. transform.position += new Vector3(0, 1, 0) \* Time.deltaTime;: This line moves the GameObject upwards at a constant speed. Time.deltaTime is used to make the movement frame-rate independent.
2. transform.Rotate(new Vector3(0, 30, 0) \* Time.deltaTime, Space.World);: This line rotates the GameObject around its y-axis. We use Space.World to rotate it in world space. \*/
3. A **Quaternion** in Unity is a data structure used to represent and manipulate rotations. It's often preferred over traditional Euler angles (which use degrees to represent rotations around the X, Y, and Z axes) because it avoids problems like gimbal lock and provides smoother rotational interpolation.

Code Example:

using UnityEngine;

public class ExampleScript: MonoBehaviour

{

void Update()

{

// Rotate the GameObject by 45 degrees around the Y axis every second

Quaternion rotation = Quaternion.Euler(0, 45 \* Time.deltaTime, 0);

transform.rotation = transform.rotation \* rotation;

}

}

/\* In this Script:

1. Quaternion.Euler(0, 45 \* Time.deltaTime, 0) creates a Quaternion representing a rotation. The rotation is 45 degrees around the Y-axis. Time.deltaTime is used to make the rotation frame-rate independent.
2. transform.rotation = transform.rotation \* rotation; applies this rotation to the current rotation of the GameObject. The multiplication of Quaternions results in a combined rotation.\*/
3. **deltaTime** in Unity's context refers to the time in seconds it took to complete the last frame. It's a crucial concept in game development as it helps in creating frame rate independent movement and animations. This means that regardless of how fast or slow the game is running, movements will appear consistent.

Code Example:

void Update()

{

float speed = 5.0f; // Speed of the GameObject

// Move the GameObject forward continuously at speed per second

transform.Translate(Vector3.forward \* speed \* Time.deltaTime);

}

/\* In this Script:

1. Update() is a method that Unity calls once per frame.
2. transform.Translate is a method that moves the GameObject in a specified direction.
3. Vector3.forward represents the forward direction (along the Z-axis in Unity's 3D space).
4. speed is a variable that determines how fast the GameObject moves.
5. Time.deltaTime is multiplied with speed to ensure the movement speed is consistent across different frame rates. Without Time.deltaTime, the GameObject's speed would vary depending on the frame rate, moving faster on higher frame rates and slower on lower ones. \*/
6. **Linear interpolation**, often referred to as Lerp in Unity, is a method used to smoothly transition between two values over a certain period. In Unity, Lerp is commonly used for positions, rotations, scales, colors, and more, ensuring that these transitions appear smooth and natural.

Code Example:

using UnityEngine;

public class ExampleScript : MonoBehaviour

{

public Transform startMarker;

public Transform endMarker;

public float speed = 1.0F;

private float startTime;

private float journeyLength;

void Start()

{

startTime = Time.time;

journeyLength = Vector3.Distance(startMarker.position, endMarker.position);

}

void Update()

{

float distCovered = (Time.time - startTime) \* speed;

float fractionOfJourney = distCovered / journeyLength;

transform.position = Vector3.Lerp(startMarker.position, endMarker.position, fractionOfJourney);

}

}

/\* In this Script:

1. startMarker and endMarker are the starting and ending points of the movement.
2. speed determines how fast the object moves from the start to the end point.
3. startTime records the time when the movement started.
4. journeyLength calculates the distance between the start and end points.
5. In the Update method, the script calculates how far along the journey the object should be based on the elapsed time.
6. Vector3.Lerp is then used to smoothly interpolate the object's position between the start and end points based on the calculated fraction of the journey. \*/
7. **MovePosition and MoveRotation** are methods in Unity used to move and rotate Rigidbody objects. These methods are particularly useful when you need to move or rotate objects in a physics-based environment, as they allow for smooth movement and rotation while taking into account the physics simulation.

* MoveRotation is is used to rotate a Rigidbody. Similar to MovePosition, it interacts with the physics engine, making it suitable for physics-based rotations.

Code Example:

public class ExampleScript : MonoBehaviour

{

public Rigidbody rb;

public Quaternion newRotation;

void Update()

{

// Rotate the Rigidbody to the new rotation

rb.MoveRotation(newRotation);

}

}

//In this example, rb.MoveRotation(newRotation) rotates the Rigidbody to newRotation. This rotation is often calculated using Quaternion.Euler if you want to rotate in degrees, or it might be a direct assignment from another rotation value.

* Move Position is used to move a Rigidbody to a new position. It's a physics-based method, so it moves the object in a way that interacts properly with the physics engine, unlike directly setting the position of the transform which can lead to unrealistic or buggy behavior in physics simulations.

Code Example:

public class ExampleScript : MonoBehaviour

{

public Rigidbody rb;

public Vector3 newPosition;

void Update()

{

// Move the Rigidbody to the new position

rb.MovePosition(newPosition);

}

}

//In this example, rb.MovePosition(newPosition) is used to move the Rigidbody attached to this object to newPosition. This would typically be updated in the Update method or within a coroutine for smooth movement.

1. **Mathf.Abs** is a method in Unity's Mathf class that returns the absolute value of a number. The absolute value of a number is its distance from zero on the number line, without considering the direction (positive or negative). This method is often used in scenarios where you need to work with the magnitude of a value irrespective of its sign.

Code Example:

using UnityEngine;

public class ExampleScript: MonoBehaviour

{

void Start()

{

float negativeValue = -10f;

float absoluteValue = Mathf.Abs(negativeValue);

Debug.Log("The absolute value of " + negativeValue + " is " + absoluteValue);

}

}

/\* In this Script:

1. We have a negative float value, -10f.
2. Mathf.Abs(negativeValue) is used to get the absolute value of negativeValue.
3. The result is 10f, which is the absolute value of -10f.
4. This value is then printed to the Unity console using Debug.Log.\*/
5. **IEnumerator and StartCoroutine** are key components of Unity's scripting used for creating coroutines. Coroutines in Unity allow you to pause the execution of a function and resume it in the next frames. This is particularly useful for implementing delays, animations, or any sequence of actions that occur over time.

* **IEnumerator** is an interface that methods (usually referred to as coroutines) must implement to be used with StartCoroutine. A method that returns IEnumerator can yield execution back to Unity and then continue from where it left off in the next frame.
  + Within an IEnumerator method, yield return is used to pause the coroutine. After the specified condition or time, the coroutine resumes.
* **StartCoroutine** is a method used to start a coroutine. It takes an IEnumerator method as an argument.

Code Example:

using UnityEngine;

using System.Collections;

public class ExampleScript: MonoBehaviour

{

void Start()

{

// Start the coroutine named "ExampleCoroutine".

StartCoroutine(ExampleCoroutine());

}

IEnumerator ExampleCoroutine()

{

// Print the time of when the function is first called.

Debug.Log("Started Coroutine at timestamp : " + Time.time);

// Yield execution of this coroutine and return to the main loop until next frame.

yield return null;

// After one frame, print the time again.

Debug.Log("After one frame, the timestamp is : " + Time.time);

// Yield execution for 2 seconds.

yield return new WaitForSeconds(2);

// After 2 seconds, print the time again.

Debug.Log("Finished Coroutine at timestamp : " + Time.time);

}

}

/\*In this Script:

1. The coroutine ExampleCoroutine is started in the Start method using StartCoroutine.
2. Inside ExampleCoroutine, yield return null causes the coroutine to pause until the next frame.
3. yield return new WaitForSeconds(2) pauses the coroutine for 2 seconds.
4. After each yield, execution resumes from the point of the last yield. \*/
5. **GetComponent** is a method in Unity used to access components attached to a GameObject. Components are essentially parts of a GameObject, like a Rigidbody, Collider, or a custom script. This method is extremely useful in Unity scripting, allowing scripts to interact with and manipulate other components on the same GameObject or on other GameObjects in the scene.

Code Example:

//Imagine you have a GameObject in your Unity scene, like a player character. This GameObject has a Rigidbody component attached to it, which allows it to interact with Unity's physics system.

using UnityEngine;

public class PlayerController : MonoBehaviour

{

private Rigidbody playerRigidbody;

void Start()

{

// Get the Rigidbody component attached to this GameObject

playerRigidbody = GetComponent<Rigidbody>();

// Now you can use playerRigidbody to manipulate the physics of the GameObject

}

void Update()

{

// Example: Add a force to the Rigidbody in the upward direction

if (Input.GetKeyDown(KeyCode.Space))

{

playerRigidbody.AddForce(Vector3.up \* 10, ForceMode.Impulse);

}

}

}

/\* In this Script:

1. GetComponent<Rigidbody>(): This line gets the Rigidbody component attached to the same GameObject the script is attached to.
2. Start Method: In the Start method (which is called when the game starts), the Rigidbody component is fetched and stored in the playerRigidbody variable.
3. Update Method: In the Update method (which is called once per frame), we're checking for a key press (space bar in this case) and using the playerRigidbody to apply an upward force, making the player "jump". \*/