Design Fundamentals Framework

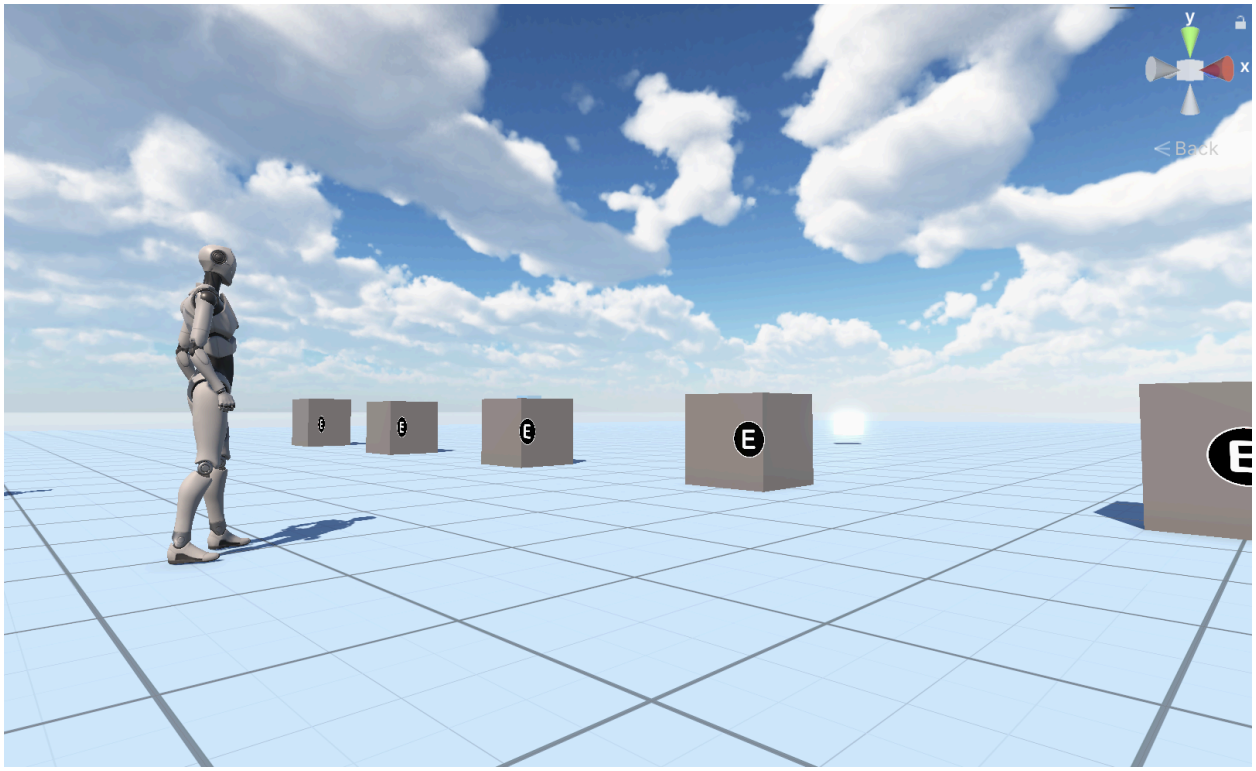
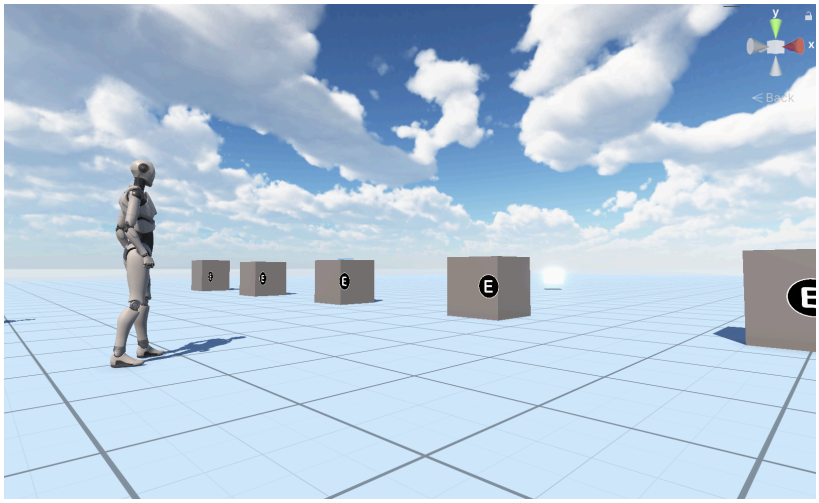


Table of contents

What is the framework?

The Design Fundamentals Framework is a collection of components (scripts), a scene setup and the documentation that will help to create a story game.



The concept behind the framework is that you can duplicate the provided scene and edit it to fit your story's needs. You can duplicate the objects you need, remove those that you don't need, change the object's properties and adjust the script to craft your own story.

In the framework there are a number of Unity components that provide object interaction. These objects are called *"PlayerActivatables"*. They can be activated by touching the objects with the player or by holding the E button for a few seconds when the objects are in range *(a so-called "ProximityTrigger")*. The project's Demonstration scene shows these components in Play mode.

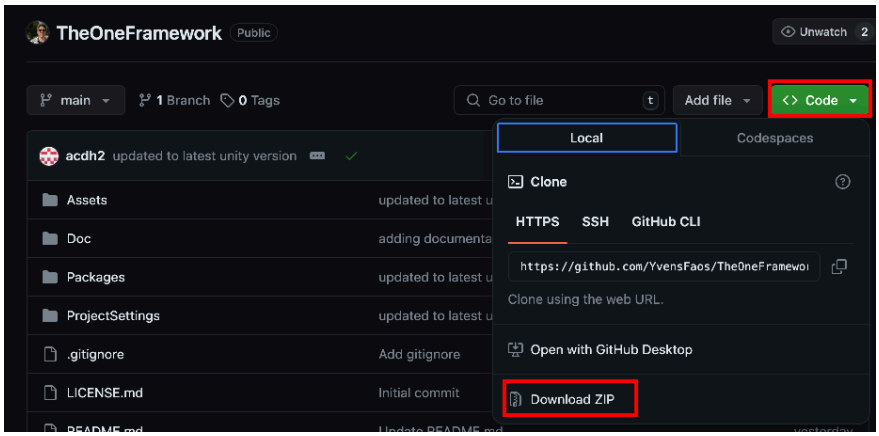
The framework also offers *YarnSpinner* integration. YarnSpinner is a library that can set up an interactive dialog in Unity. The integration is done by a number of functions that you can use in your YarnSpinner script, for instance to make an NPC *(non-playable character)* say something or walk somewhere, to put items in the player's inventory or check if they are there. Any PlayerActivable can also be triggered from your YarnSpinner scripts.

In the manual we will describe how to install the framework and how to open the demonstration scene. After that, it will explain how to set up your own scene and script. The User Guide section will explain in step by step tutorials how to set up certain things yourself using the framework. The function guide will describe all the YarnSpinner functions that the framework offers with a brief code example per function.

Where applicable, the framework will reference documentation from YarnSpinner. We hope that the framework makes it easier to start setting up your story games in Unity, that it will show what the role of a designer is in a development process and we also hope you will find it inspiring and you will make it your own. Any code that was written for the framework is MIT license, which allows you to use it in any project you like.

Installation

Visit the link: https://github.com/YvensFaos/TheOneFramework/

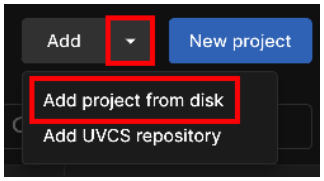


Go to Code->Download ZIP to download the project. Extract the file at a convenient location on your hard drive.

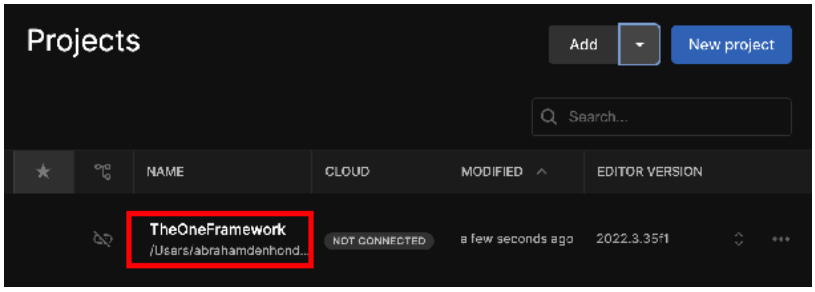
Make sure you have the matching Unity version installed. The current version is *2022.3.44f1. Your version should be the same or higher. Opening the project in an earlier version may lead to unexpected behavior.*

Also, please ensure you have *git* installed. Unity needs that to install YarnSpinner. Instructions on how to install git can be found here: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

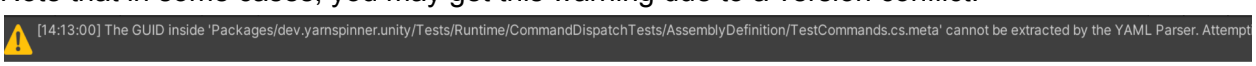
In Unity HUB, choose Add->Add project from disk to add the project. Select the project folder.



Then, click on the project name in Unity HUB to open it.



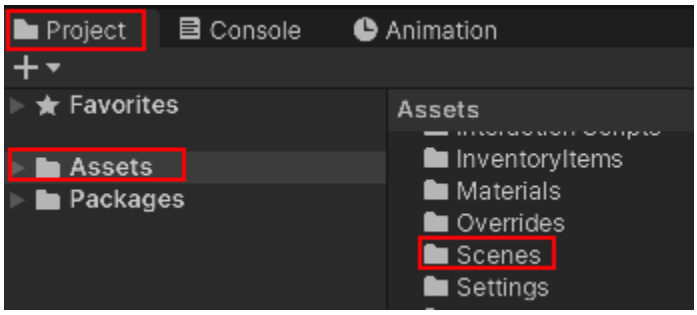
Note that in some cases, you may get this warning due to a version conflict:



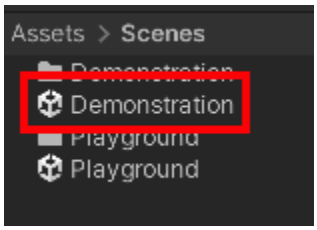
To resolve it, re-install YarnSpinner. See the chapter *Installing YarnSpinner from git.*

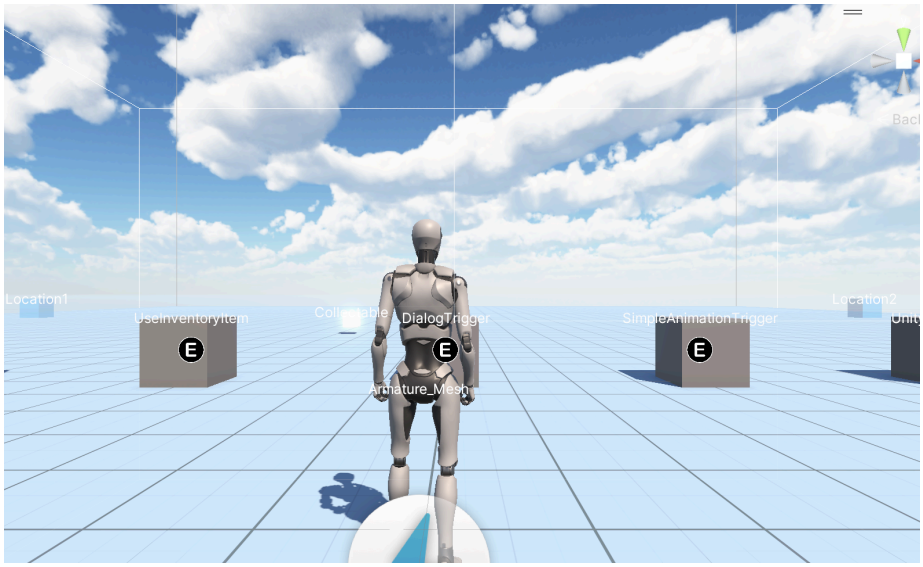
Getting started

Open the project window. Locate the “Assets/Scenes” folder.



In the folder there are two scenes: Demonstration and Playground. Double-click the scene named “Demonstration” to open it.

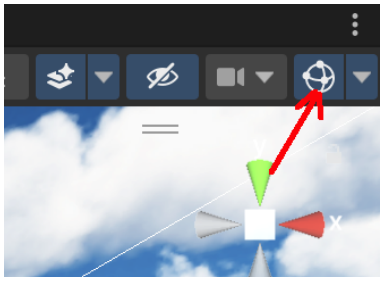




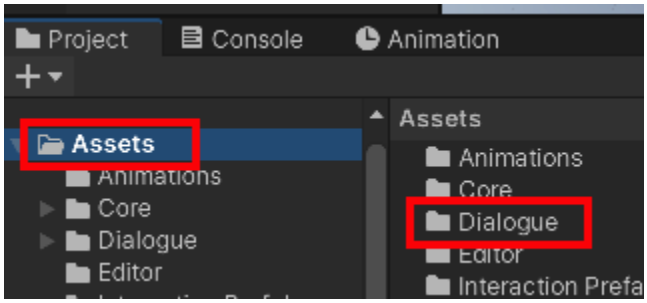
Press Unity’s play button to play an interactive demonstration of the PlayerActivatables.

After the demonstration, press the same button again to stop playmode.

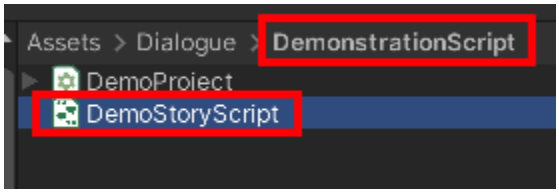
At the top-right of the game window, you will see this icon. Use this to toggle gizmo visibility.



By default, object names are shown. You can toggle this on or off by disabling the ShowObjectNamesEditor in the Gizmo list. (The triangle next to the above icon)
If you press play, you will see an interactive tutorial showing the framework's features.To see the script for the tutorial, open the folder Dialogue in the Project window.

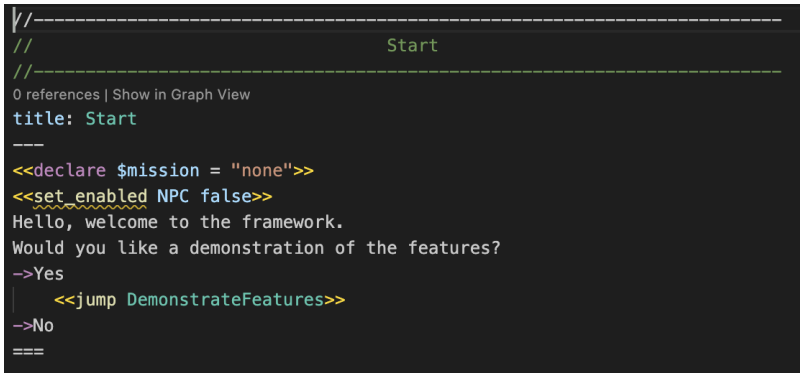


In this folder, there is a folder called DemonstrationScript and in there you will find DemoStoryScript.



Open the script by double-clicking it. *You may need to install Visual Studio or Visual Studio Code to open it.*

The script is divided in “nodes”, such as these:



```
//---------------------------------------------------------------------------
//                                Start
//---------------------------------------------------------------------------
0 references | Show in Graph View
title: Start
---
<<declare $mission = "none">>
<<set_enabled NPC false>>
Hello, welcome to the framework.
Would you like a demonstration of the features?
->Yes
    <<jump DemonstrateFeatures>>
->No
===
```

Note that I installed the YarnSpinner extension in Visual Studio Code. That gives me syntax highlighting (the text color changes based on the semantics) See the link here:
https://docs.yarnspinner.dev/getting-started/editing-with-vs-code/installing-the-extension

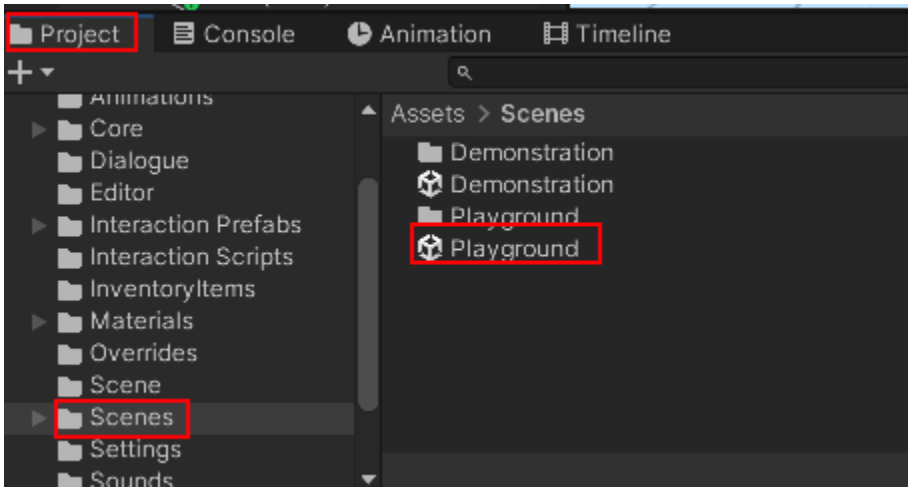
Nodes contain a title and a body. The body is the part between - - - and = = =.
In the body, you will see all the dialogue that belongs to the scene, as well as commands for YarnSpinner/the framework. You can recognize commands because they are surrounded by <<< and >>>.

The lines that start with a -> are options the user can choose from. The lines directly under those show how the script will respond to this choice. So in the above example, if the player chooses “Yes”, the script will continue at the node with the title “DemonstrateFeatures”.

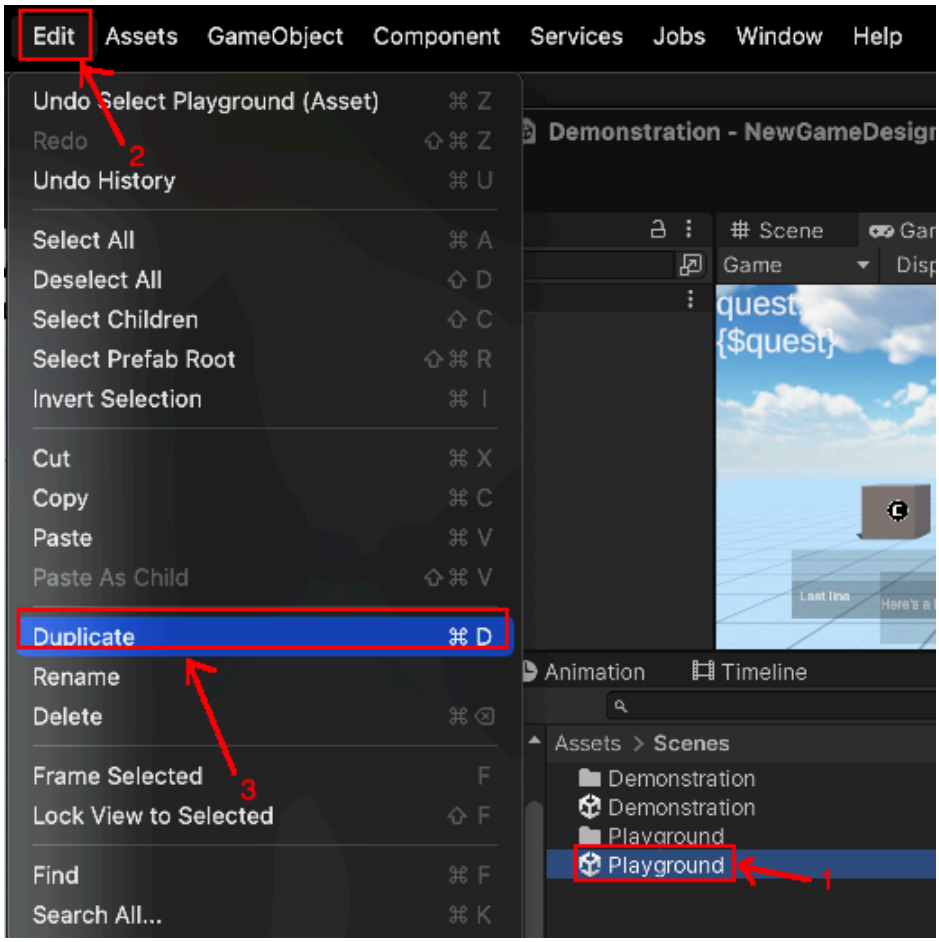
Check out the YarnSpinner documentation for more syntax basics:
https://docs.yarnspinner.dev/beginners-guide/syntax-basics

Creating your own scene and adding a script

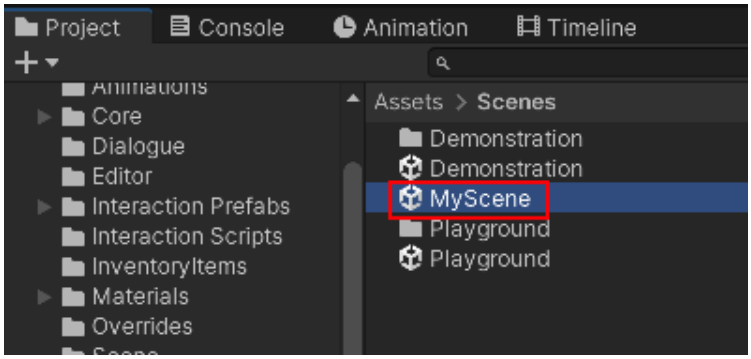
If you want to add your own scene, open the folder Assets/Scenes.
In there you will find a scene that is called Playground. This is a scene that contains a player, a camera and a terrain. Also, it has some light settings and the required UI elements set up.



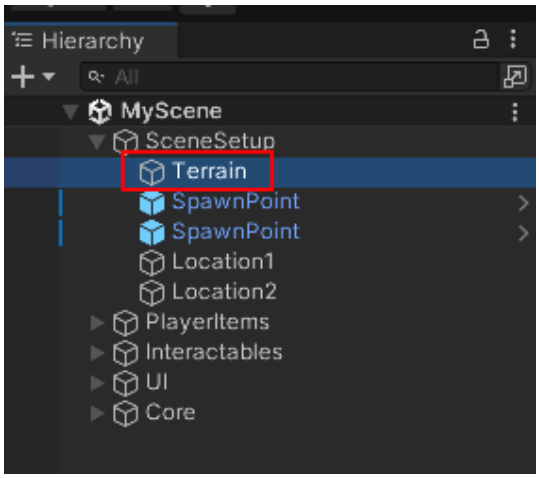
To set up your own story, you can make a duplicate of this scene and open it.
Select the scene, then go to Edit->Duplicate (or press CMD/CTRL+D)



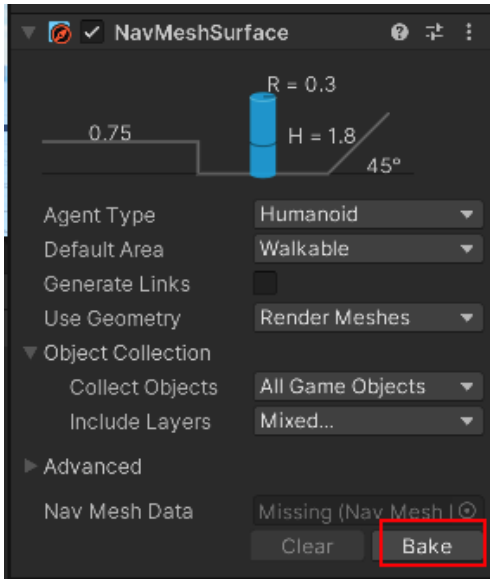
Then, rename the newly created scene and double-click it to open it.



In the Hierarchy window, select the Terrain gameobject. It can be found under SceneSetup:



Go to the Inspector window and scroll down. There you will see a button marked “Bake”. Press it to bake the navigation mesh.

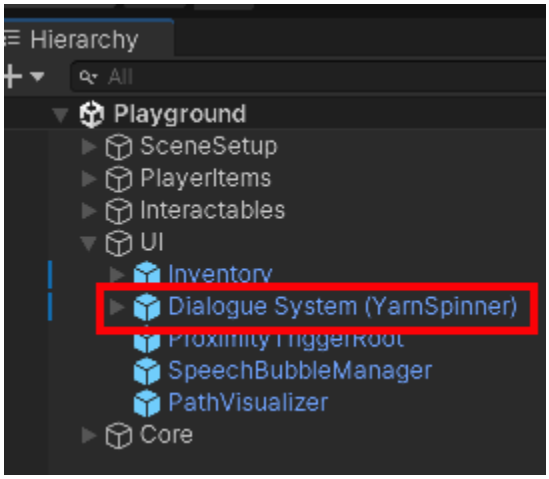


This is needed for the walk_to and the show_path functions. Unity needs to scan the terrain in order to allow automatic navigation for the NPC’s. (See walk_to and show_path functions in the Function Guide chapter)

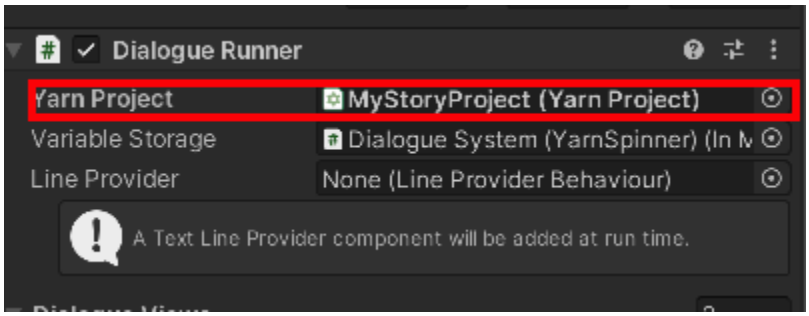
Yarnprojects

YarnSpinner uses YarnProjects to organize scripts. A project is a collection of scripts.

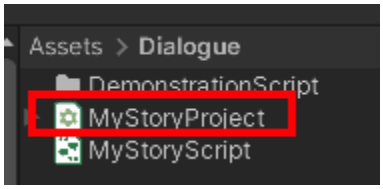
You can assign these to the DialogRunner GameObject that is in the scene. This is a GameObject that is called "Dialog System (Yarnspinner)"



If you select it, and go to the Inspector, you will find a DialogRunner component. The property “Yarn Project” will reference the YarnProject in the Assets folder.

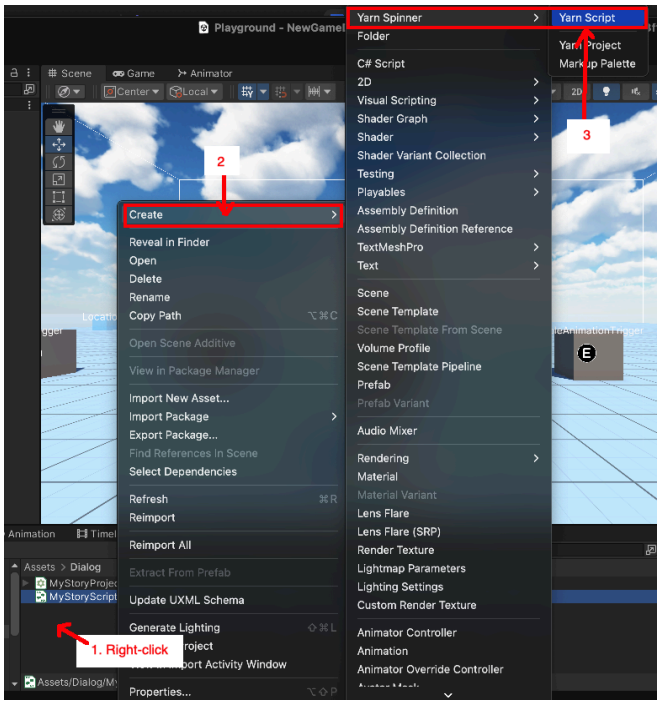


The project can be found in the “Assets/Dialogue” folder:

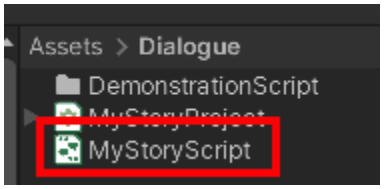


By default, MyStoryProject will collect all the YarnScripts in the same folder. In this case, that is only MyStoryScript, but you can add more scripts yourself.

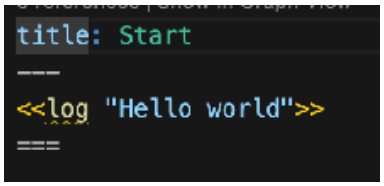
If you want to add your own script, you can add it by right-clicking the Dialogue folder (1). Then go to Create (2), then Yarn Spinner->Yarn Script (3).

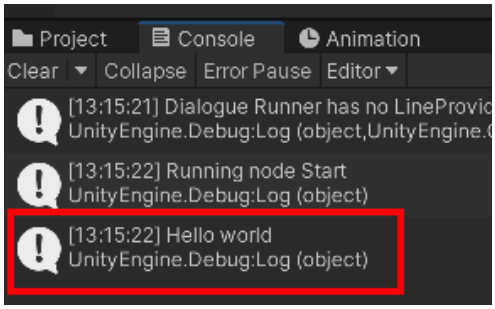


Double-click the script to edit it. *If you didn't open a script file before, Unity may prompt you to install Visual Studio.*



The script that is shipped with the framework (MyStoryScript) will show "Hello world" in the console window. It is added to the DialogueRunner in the "Playground" scene.





To learn more about YarnSpinner scripts, visit this page:
https://docs.yarnspinner.dev/beginners-guide/syntax-basics

Overview of demonstration scene elements

SceneSetup	
Terrain	The Unity terrain
SpawnPoint	A player spawn
SpawnPoint	A player spawn
Location1	Transparent cube, used for walk_to example
Location2	Transparent cube, used for walk_to example
PlayerItems	
PlayerFollowCamera	A cinemachine virtual camera (that follows the player)
Player	The player GameObject
Interactables	
DialogueTrigger	Starts a certain dialogue node in YarnSpinner
Collectable	Adds item to inventory, plays audio and goes away
NPC	Can be talked to
SimpleAnimationTrigger	Checks inventory for item, activates one animation
UseInventoryItem	Animates only if used picked up the collectable
UnityEventTrigger	Can spawn any unity event (in this case, activates particles)
SimpleAudioTrigger	Plays an audio event
MenuTrigger	Shows a UI node (the letter)
MenuTrigger (CodeLock)	Shows a UI node (the code lock)
AnimationToggle	Can play multiple animation states
UI	
Inventory	Shows your inventory (press TAB)
Dialogue System (YarnSpinner)	System YarnSpinner root note
ProximityTriggerRoot	Used at runtime
Letter	The letter that is shown
HUD	Displays the current mission (a variable from YarnSpinner)
SpeechBubbleManager	Used at runtime to generate speech bubbles
CodeLock	The codelock puzzle (code = 1234), it contains an event that gets fired when right/wrong
PathVisualiser	Displays arrows on the terrain, used for navigating the player
Core	
Main Camera	The scene camera
Directional Light	A basic light
PostFXVolume	Used for the neon effect on the pickup

PlayerActivatable Overview

“PlayerActivatables” are scripts that are in the framework that allow player interaction with items. They can be activated by a ProximityTrigger *(The player must hold the “E” button for that)* or on collision. (When the players touches the item)
These scripts are located in the folder “Assets/Interaction Scripts”. You can use them to activate various things, such as sounds, animations, dialogues and particle effects.

Here is an overview of the PlayerActivatables in the folder:

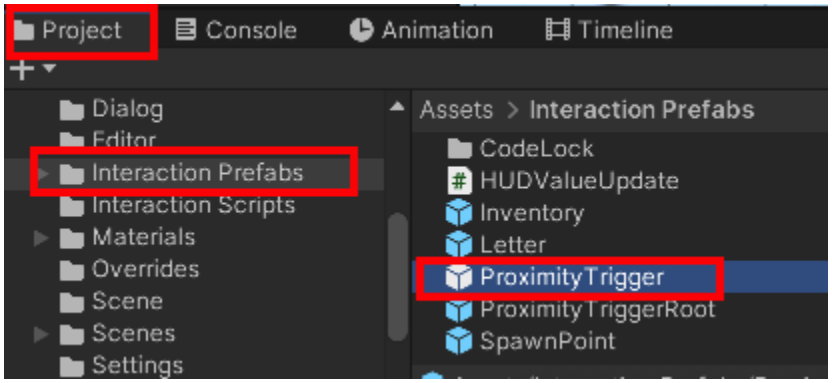
AddInventoryItem	This component will add an item to the inventory.
AnimationToggle	This component will toggle between two animations.
Collectable	This component will destroy the GameObject when the user activates it.
DialogueTrigger	This component will start a certain node in the YarnSpinner project.
FMODEventTrigger	This component will trigger an FMOD event.
LoadLevel	This component will load another level. *Levels must first be added in File->Build Settings*
ShowMenu	This component will activate a UI object when activated.
SimpleAnimationTrigger	This component will play a single animation when it is activated.
SimpleAudioTrigger	This component will play a sound when it is activated.
UnityEventTrigger	This component will fire a Unity Event.
UseInventoryItem	This component will check if a certain item is in the inventory before it allows the other scripts to be activated.
UseInventoryItemOnTarget	This component will do the same as UseInventoryItem, but it will activate scripts on another GameObject instead of itself.
VariableSetter	This component will set a variable in the YarnSpinner project.

Step-by-step descriptions

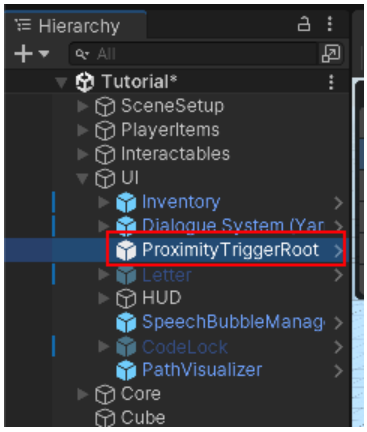
Adding ProximityTriggers

A Proximity Trigger is the E that shows up when you are close to a PlayerInteractable. You can use it to activate all the scripts on a GameObject.

1. Locate the prefab in the folder Assets/Interaction Prefabs

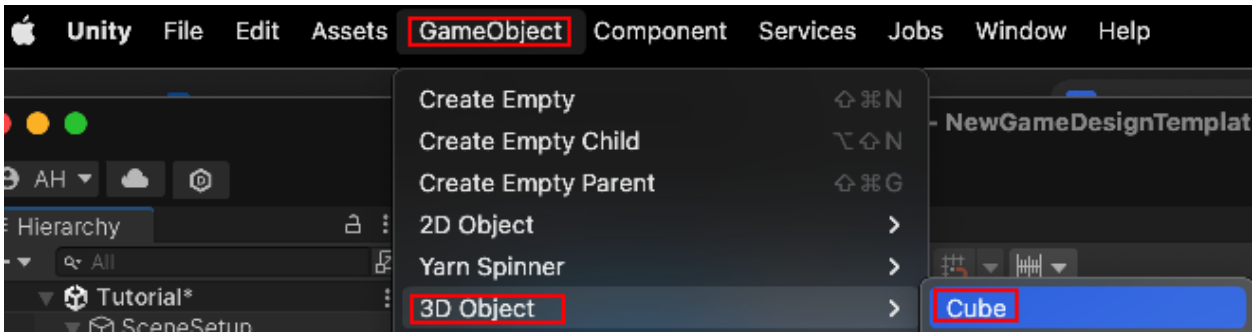


2. Ensure that in the hierarchy (i.e. under the UI node) there is a ProximityTriggerRoot.

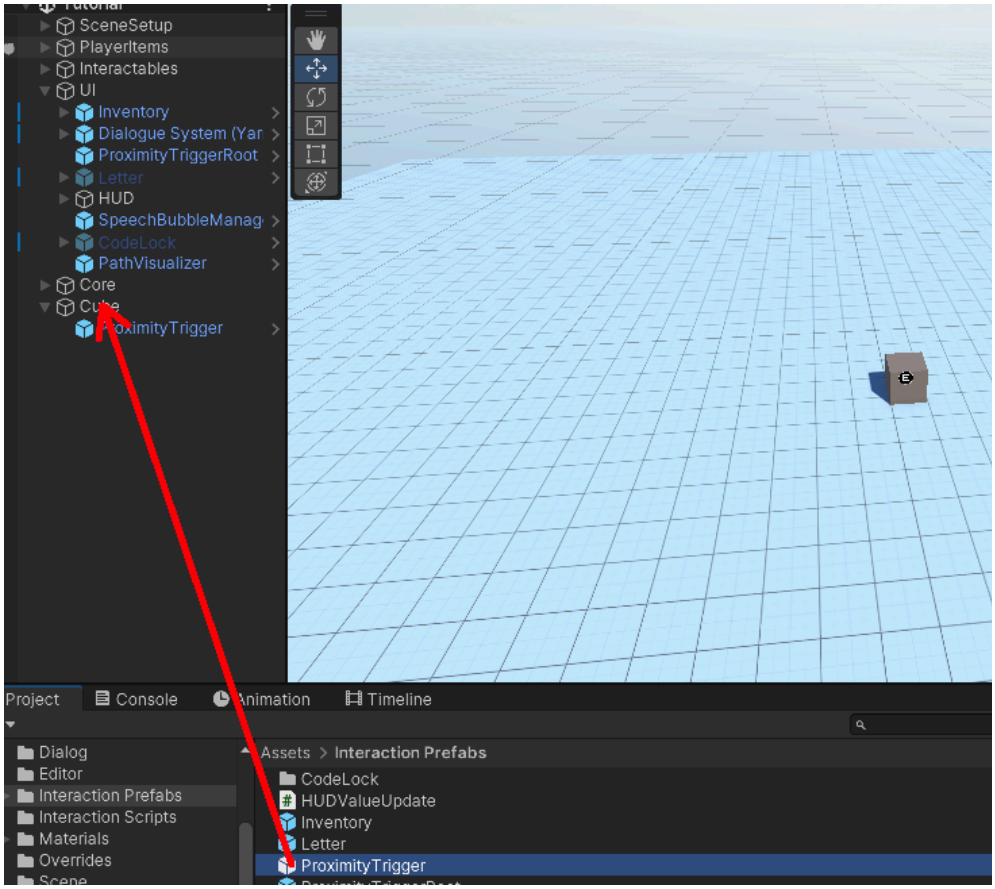


   If it is not there, add the ProximityTriggerRoot to the scene.

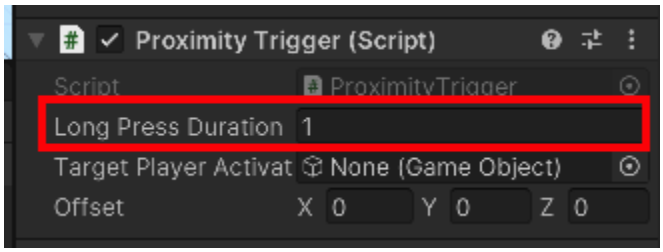
3. If you have no GameObject yet, go to the top bar menu in Unity and select GameObject->3D object->Cube first.



4. Drag the ProximityTrigger prefab to the GameObject you want to add it to.



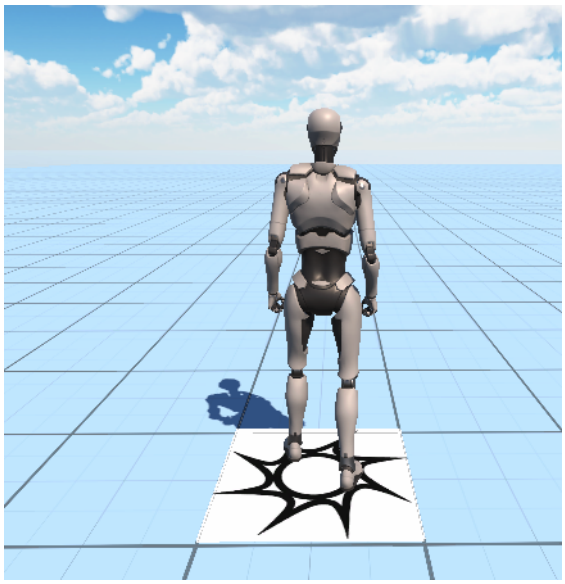
5. Optionally, set the Long Press Duration property in the inspector:



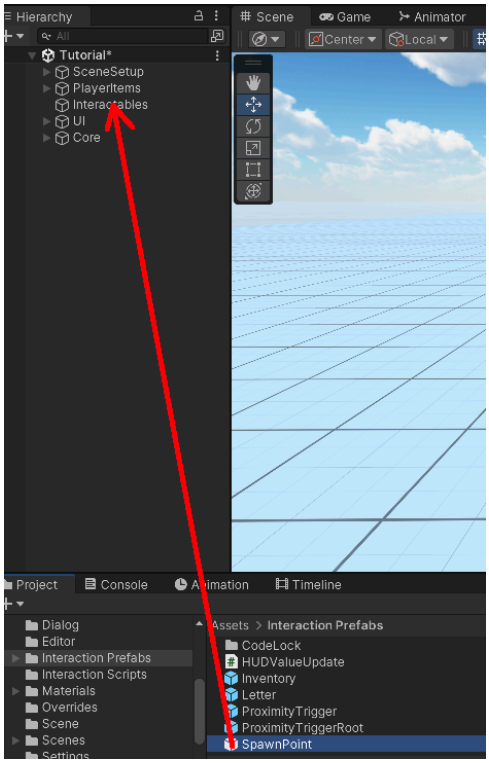
If you add any PlayerInteractable scripts to the cube, they will be activated.

Adding SpawnPoints

A SpawnPoint is a location where the player spawns. When the player touches a spawn point, it will be set as the next spawn position when the player dies.

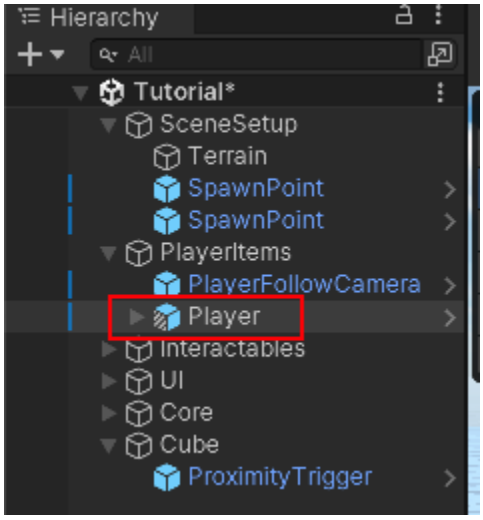


SpawnPoints can be added from the “Assets/Interaction Prefabs” folder.

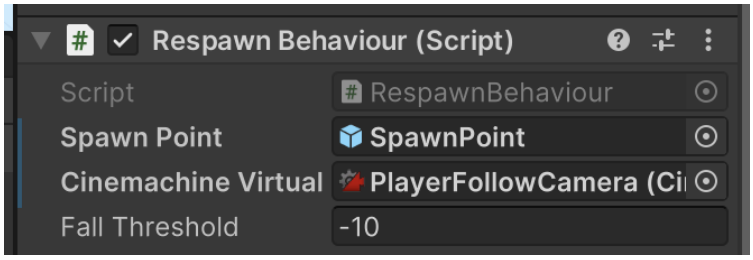


You can place them anywhere in your scene hierarchy.

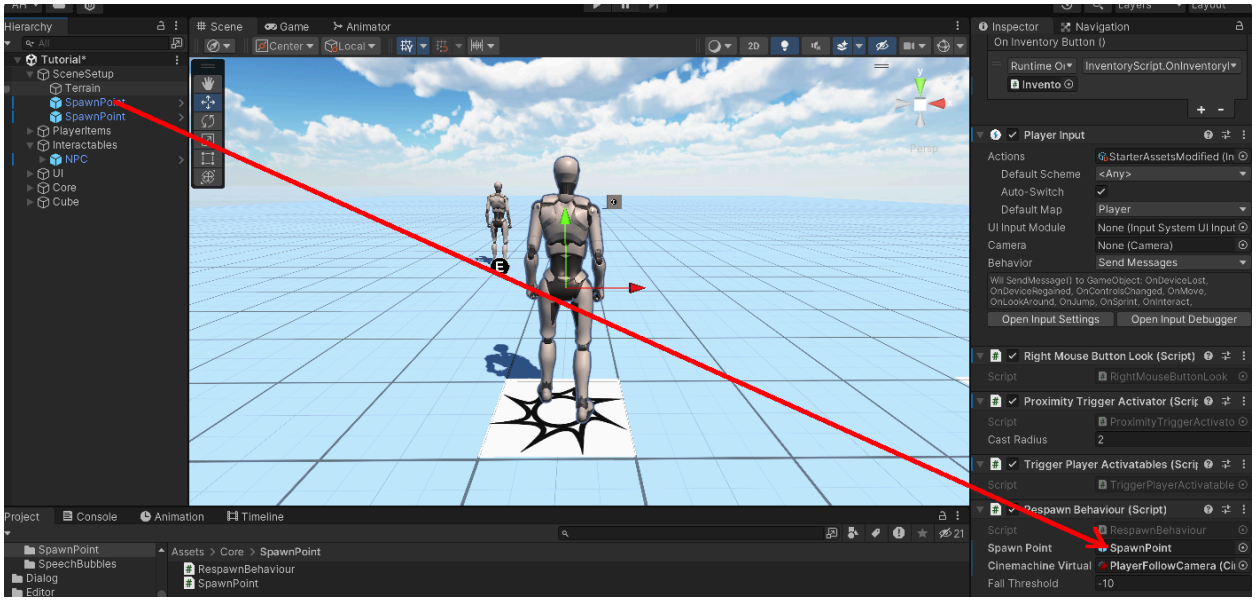
To set the spawn point for the player, select the player in the hierarchy:



In the inspector window, locate the RespawnBehaviour component (scroll down for it).



You can drag your newly created SpawnPoint on the property that is called Spawn Point. That will ensure the player will spawn at this SpawnPointat the start of the game, regardless of its initial position.

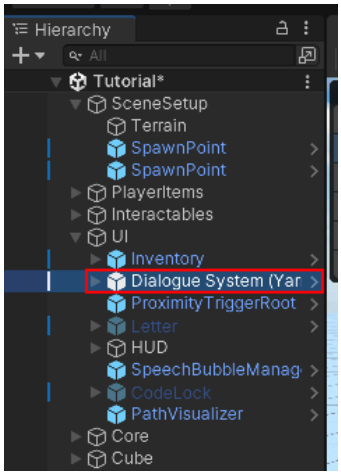


Starting a dialogue at start of the game

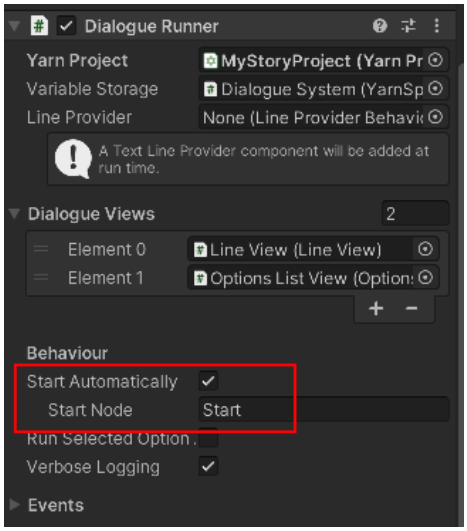
When the framework starts, it is set up in such a way that YarnSpinner will automatically play the node called “Start” in the YarnSpinner project, shown here:

```
//---------------------------------------------------------------
//                                Start
//---------------------------------------------------------------
0 references | Show in Graph View
title: Start
---
<<declare $mission = "none">>
<<set_enabled NPC false>>
Hello, welcome to the framework.
Would you like a demonstration of the features?
->Yes
    <<jump DemonstrateFeatures>>
->No
===
```

To change this behavior, locate the YarnSpinner GameObject in the hierarchy:

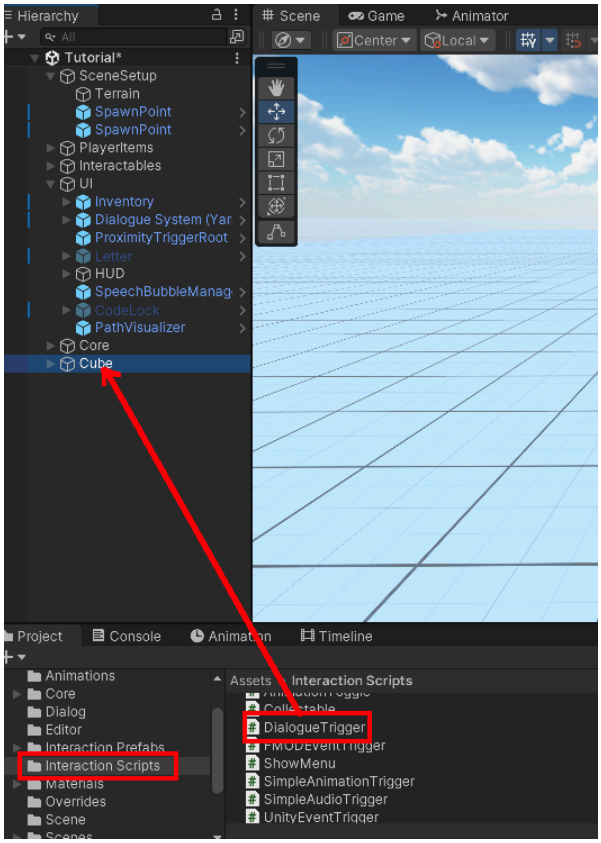


After selecting it, find the “Dialogue Runner” component in the inspector. Under *Behaviour* you will see a checkbox called “Start Automatically” and a text field called “Start Node” that currently says “Start”. Modify these properties to fit your needs.

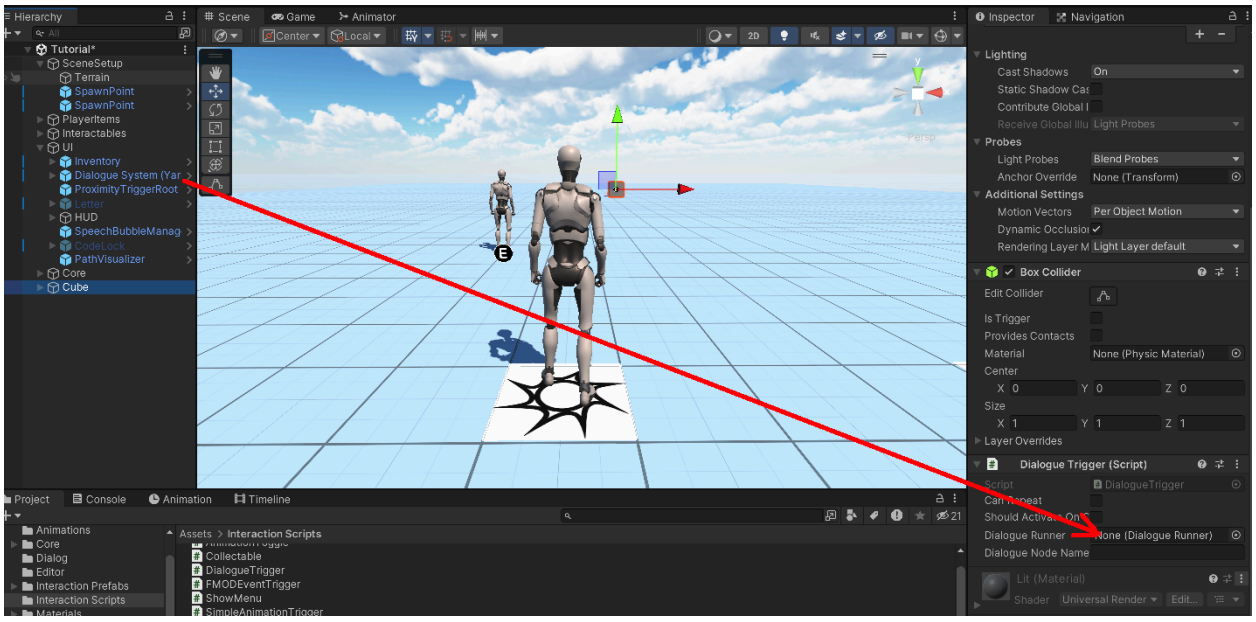


Starting a dialogue from trigger

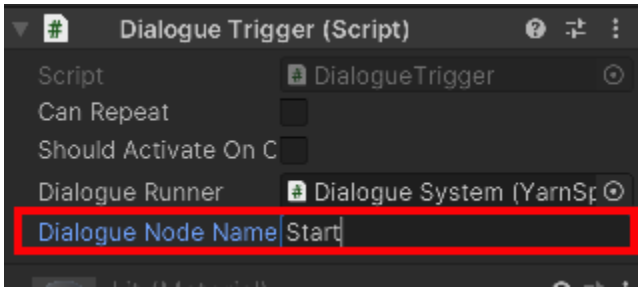
You can also use a PlayerInteractable to activate a dialogue. For that, you need to add the script DialogueTrigger to the GameObject.



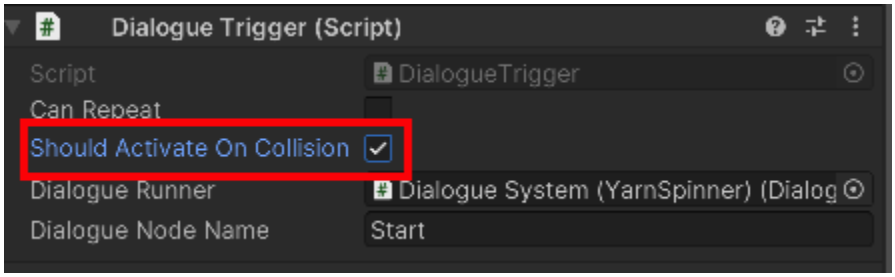
Next, locate the Dialogue Trigger component in the inspector and drag the GameObject Dialogue System (YarnSpinner) from the UI node onto the Dialogue Runner property of the component.



Finally, fill in the name of the YarnSpinner node you want to run from the interactable:

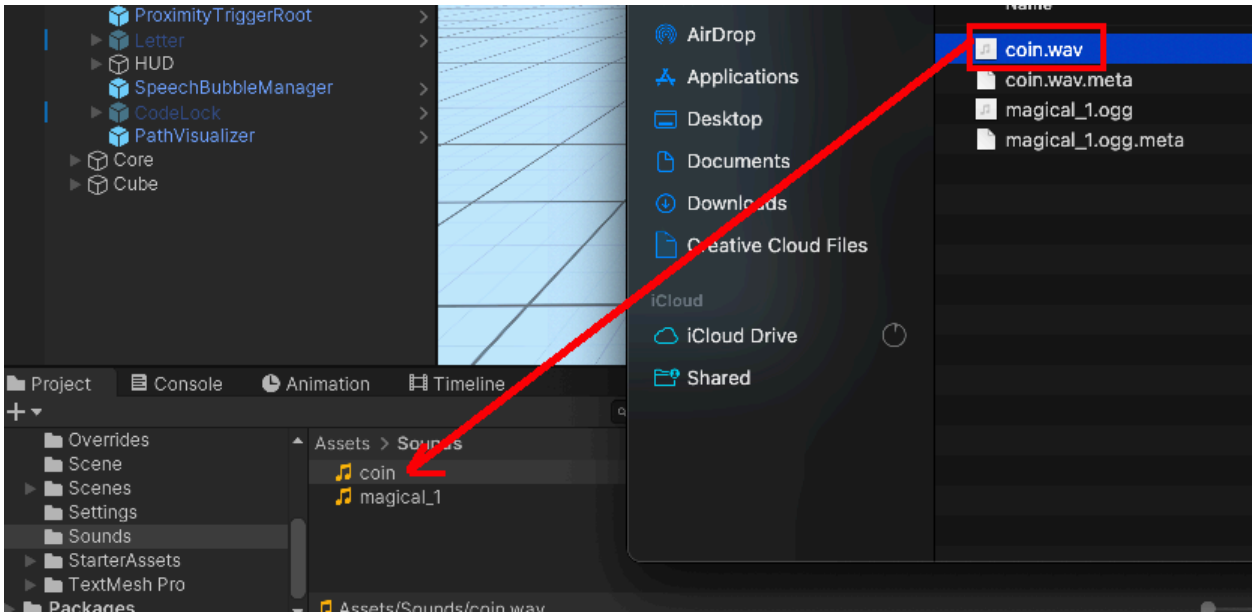


You can activate the PlayerInteractable using a ProximityTrigger (*see the section "Adding ProximityTriggers" above*) or by enabling the "Should Activate on Collision" property of the component.

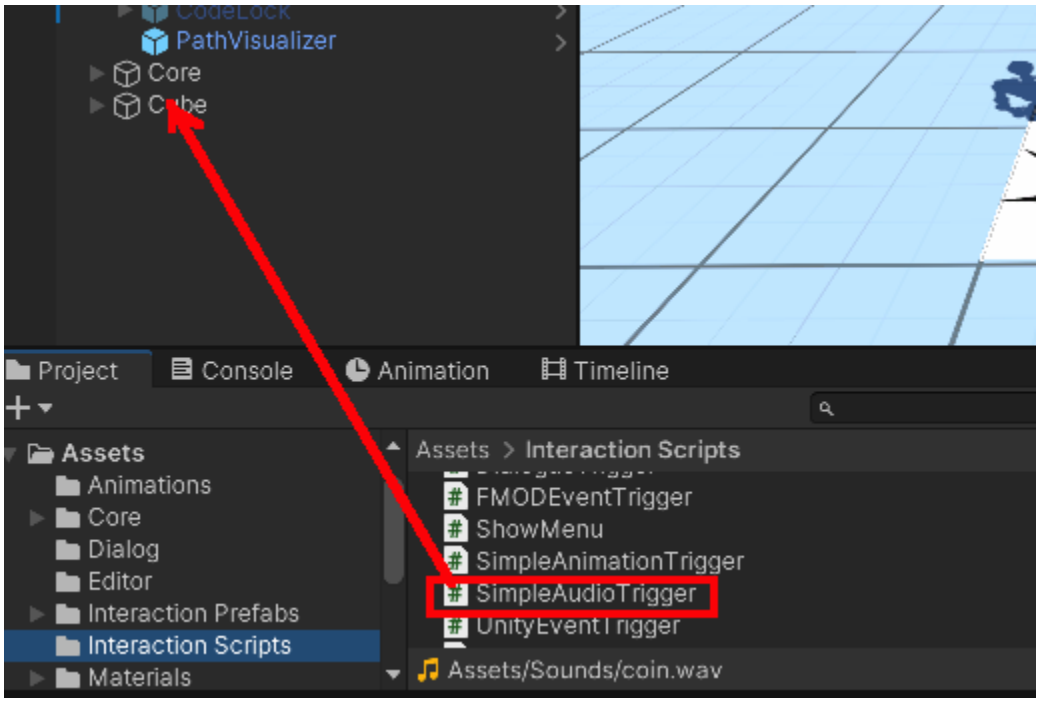


Playing a sound

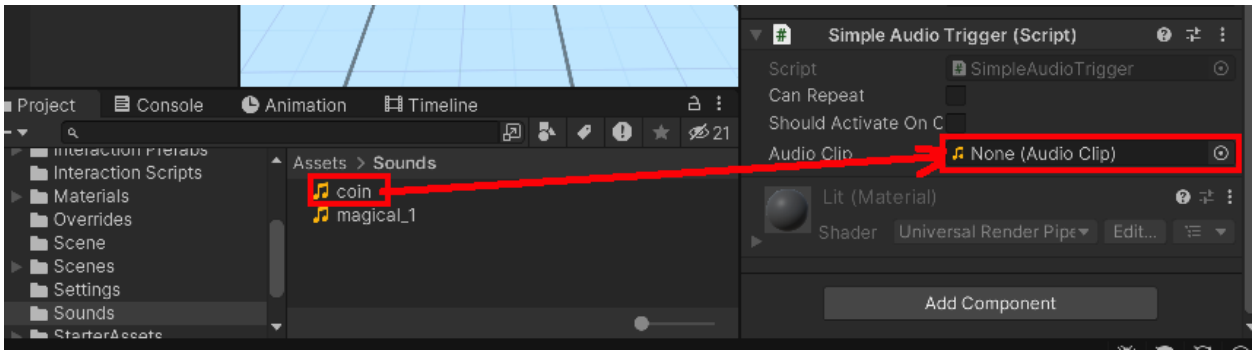
There is a PlayerInteractable that is able to play a sound when it is activated. You need to import a sound file into Unity. You can drag the sound file onto your project window.



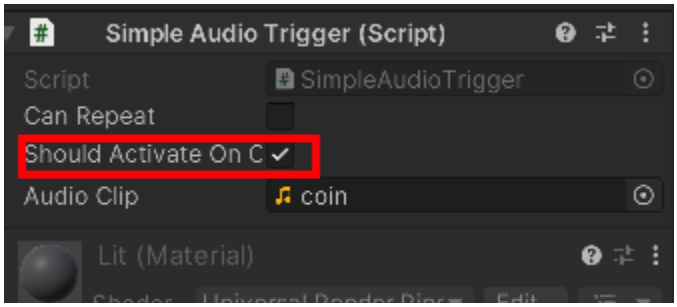
Locate the SimpleAudioTrigger script in the “Assets/Interaction Scripts” folder and drag it onto the GameObject that you want to apply it to.



Next, find the SimpleAudioTrigger component in the inspector and drag the sound from the project window onto the “Audio Clip” property of the component.



You can activate the SimpleAudioTrigger using a ProximityTrigger (*see the section “Adding ProximityTriggers” above*) or by enabling the “Should Activate on Collision” property of the component.



Working with variables in script

YarnSpinner allows you to set a variable from a script. This can be done to keep track of things that the player has done. For instance, you can use variables to keep track of items that were collected (hasKey) or places that the user has visited (hasVisited). You can also use it to count certain things. For instance, you can count how often the player visited the NPC.

Here is an example script that uses a variable $hasKey.
It can have two values: true or false. When it is “true”, the player has collected the key.

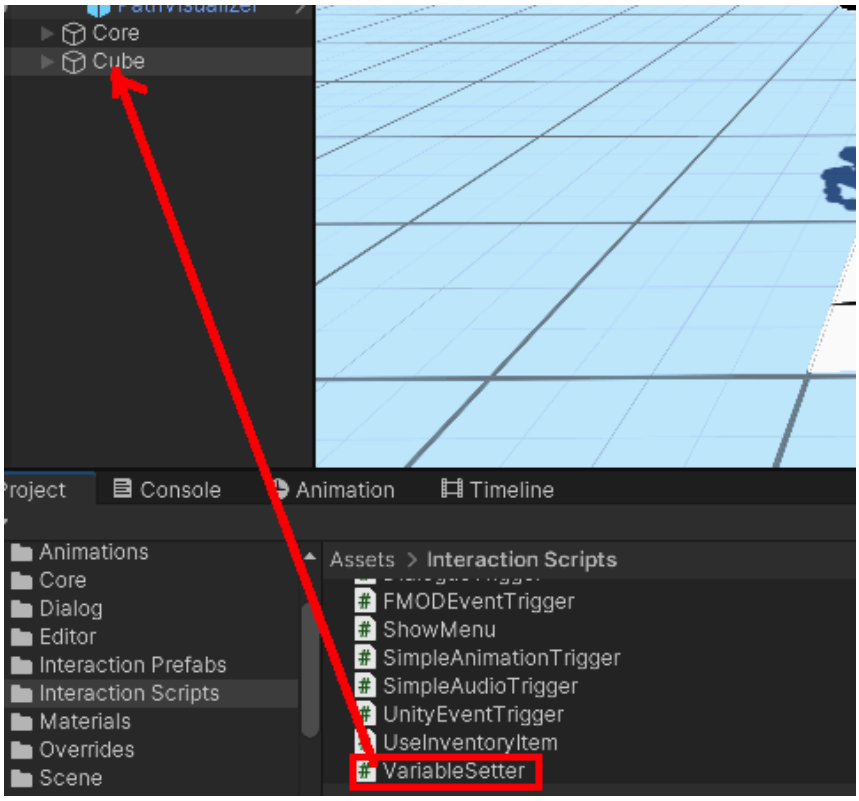
```
//---------------------------------------------------------
//                                  Start
//---------------------------------------------------------
0 references | Show in Graph View
title: Start
---
<<declare $hasKey = false>>
Hello, welcome to the framework.
Find the key!
===

//---------------------------------------------------------
//                                  GetKey
//---------------------------------------------------------
0 references | Show in Graph View
title: GetKey
---
You found a key!
<<set $hasKey to true>>
===

//---------------------------------------------------------
//                                  OpenDoor
//---------------------------------------------------------
0 references | Show in Graph View
title: OpenDoor
---
<<if $hasKey == true>>
    Opening the door!
<<else>>
    You need a key for this>>
<<endif>>
===
```

More information about YarnSpinner variables can be found on their website:
https://docs.yarnspinner.dev/getting-started/writing-in-yarn/logic-and-variables

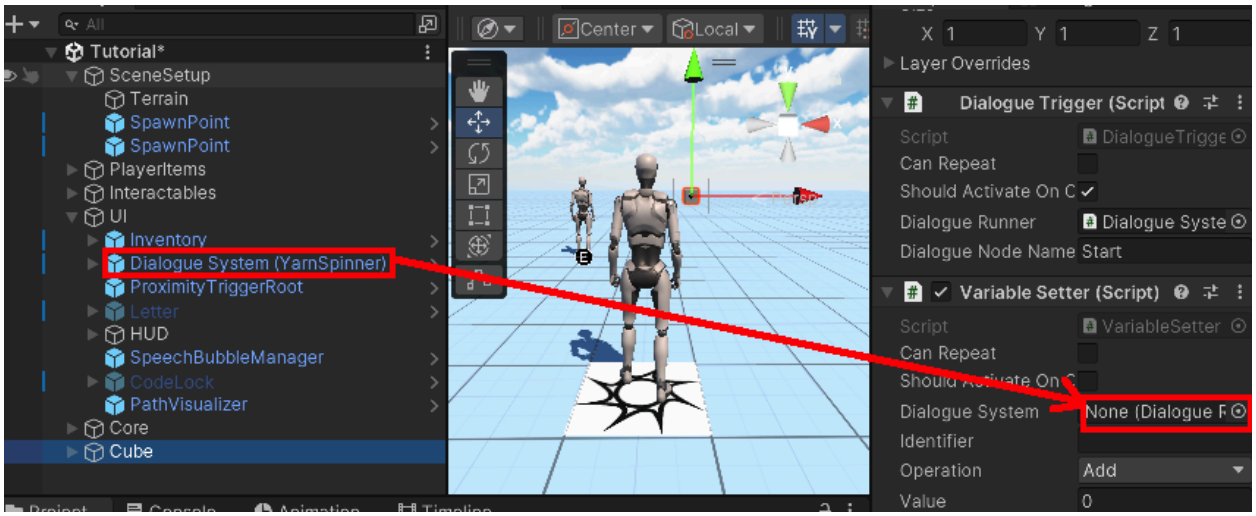
Setting variable in trigger

There is a PlayerInteractable that can set variables. When the component is activated it can alter a variable in the YarnSpinner script. You can set a variable to a certain value but also add or subtract an amount.

Locate the VariableSetter script in “Assets/Interaction Scripts” and drag it onto the GameObject you want to apply it to.

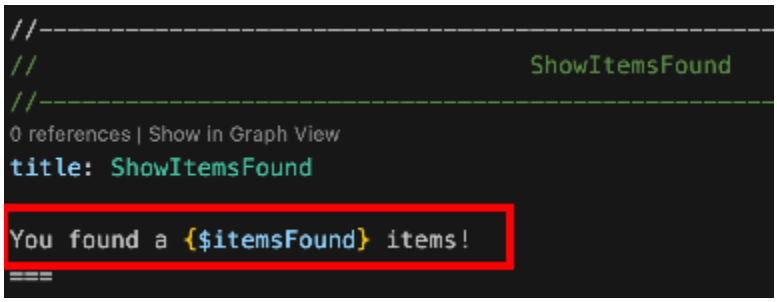


Drag the Dialogue Runner GameObject from the UI node in the hierarchy onto the VariableSetter component in the inspector:



In the YarnSpinner script, create a variable. This variable should be of the Number type. Currently that is the only type of variable supported by the VariableSetter component.

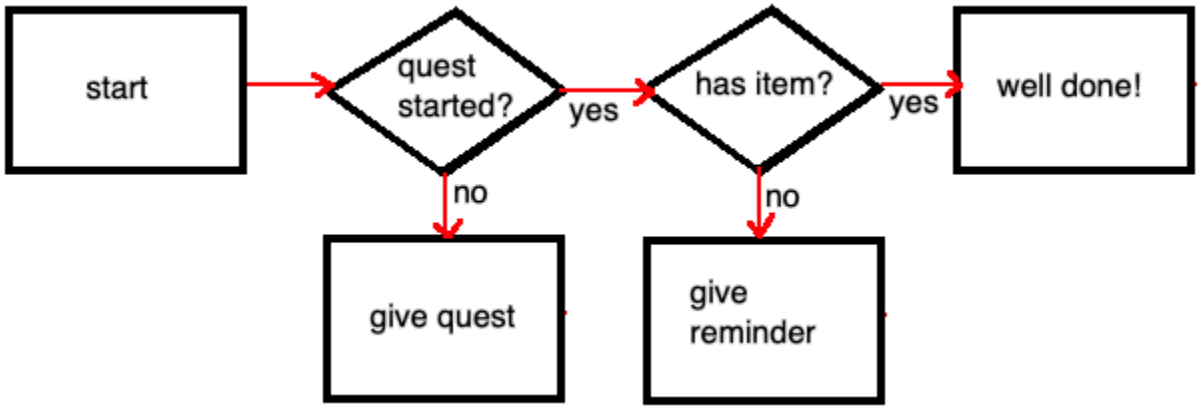
For testing purposes we can add a node that will display the variable in a dialogue.



Now when you run the dialogue node, it will display the amount of items.

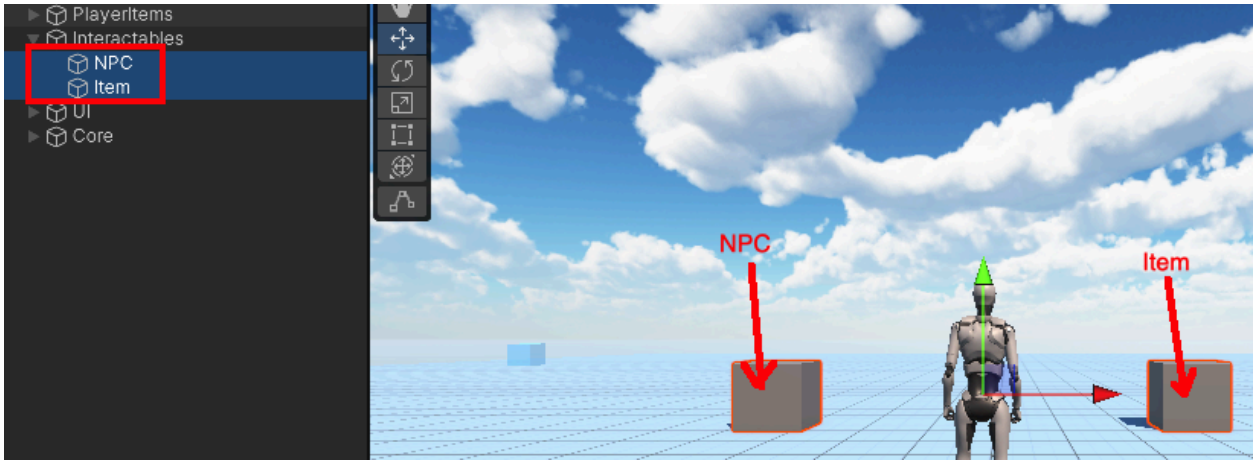
Setting up a fetch quest

If you want to create a quest for the player, you can use a variable in the YarnSpinner script to set this up. This variable can keep track of the status of the quest. For instance, these states can be “unassigned”, “in progress” and “done”.
This tutorial will tell you how to set this up. It will show how to create the variable, how to add an item to the inventory from a pickup and check for it in the YarnSpinner script.

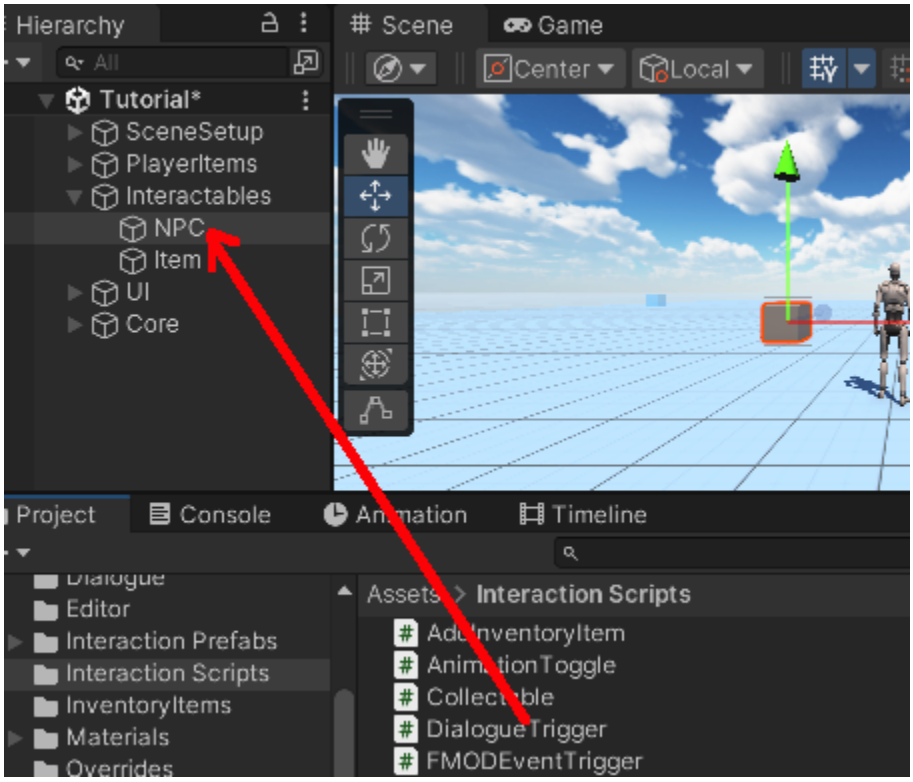


When the user interacts with an NPC, it will prompt the user to complete the first quest:
If the user is already doing the quest, talking to the NPC will give a friendly reminder.
When the user completes the quest, the NPC should reward the player.

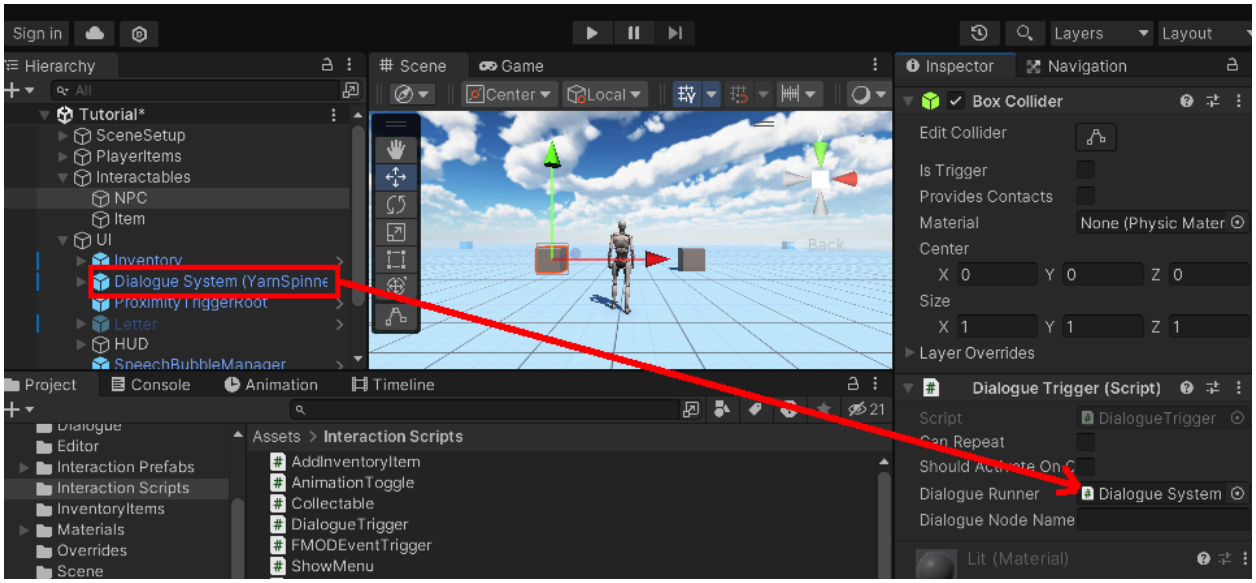
Add two cubes to the game, name one of them “NPC” and the other one “Item”.



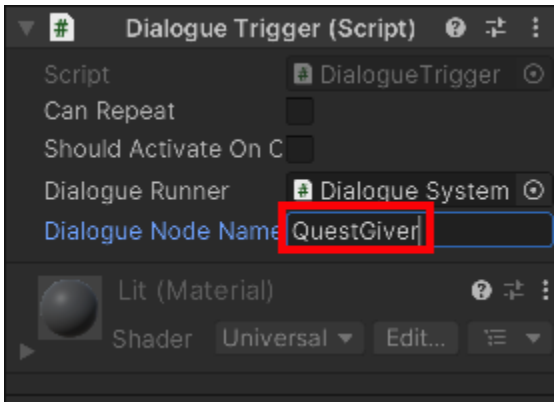
Add a DialogueTrigger to the NPC item.



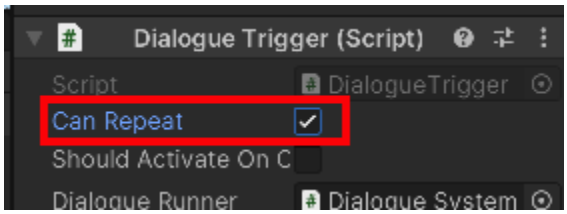
From the UI node in the scene, drag the Dialogue System onto the DialogueTrigger.



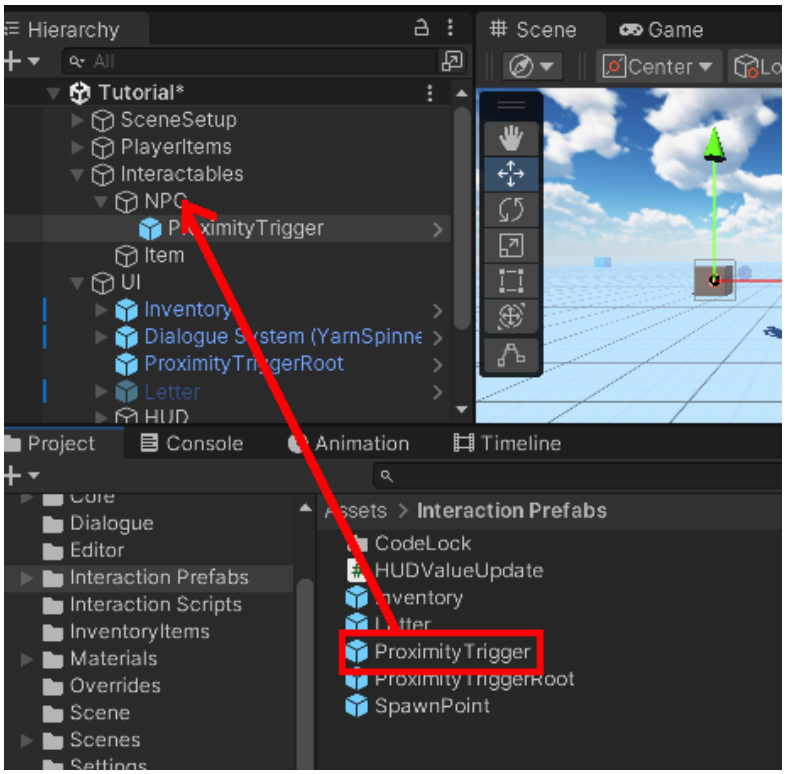
In the properties of the DialogueTrigger component, add “QuestGiver” to the Dialog Node Name.



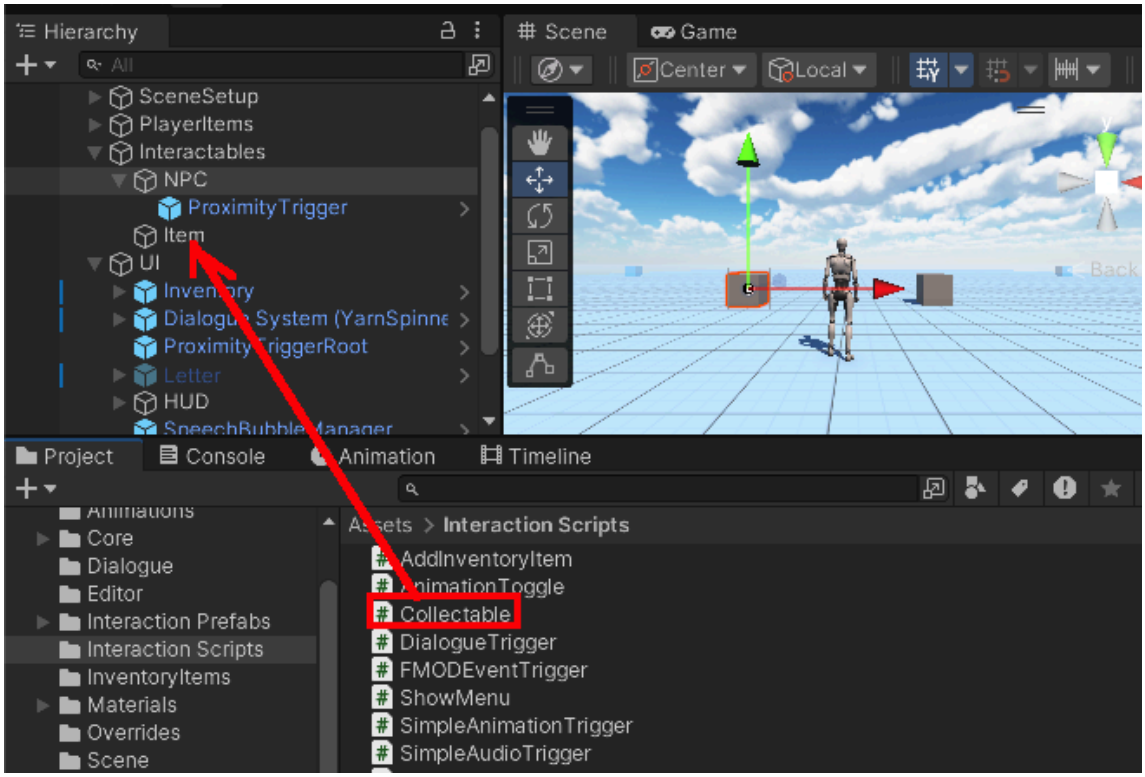
Check the box that says “Can Repeat”



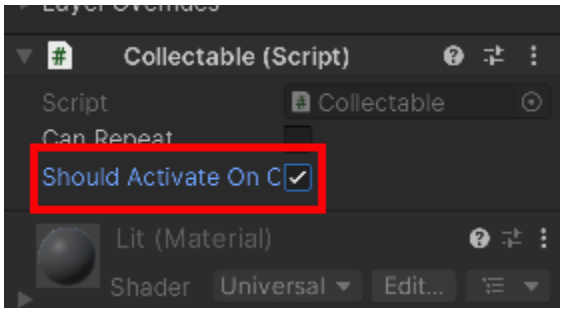
This will make the YarnSpinner script play the node called QuestGiver.
Drag a ProximityTrigger from the to folder “Assets/Interaction Prefabs” onto the NPC.



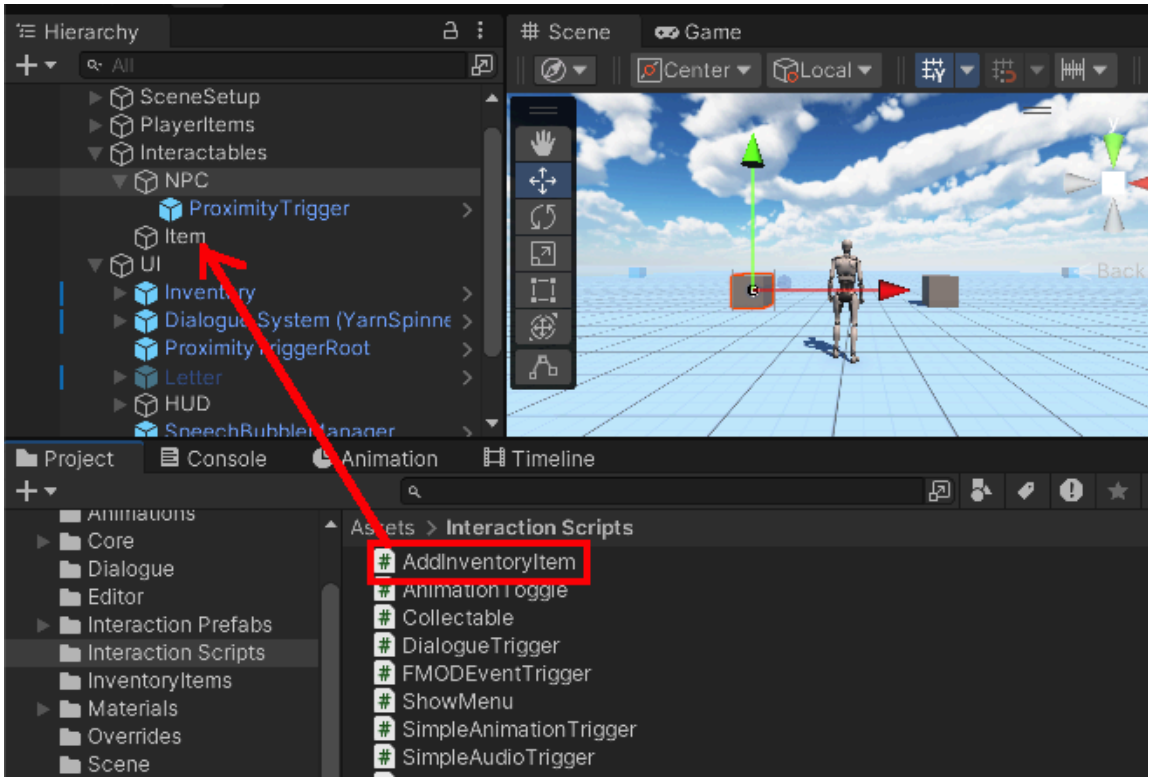
Next, drag the Collectable script onto the cube that is called Item.



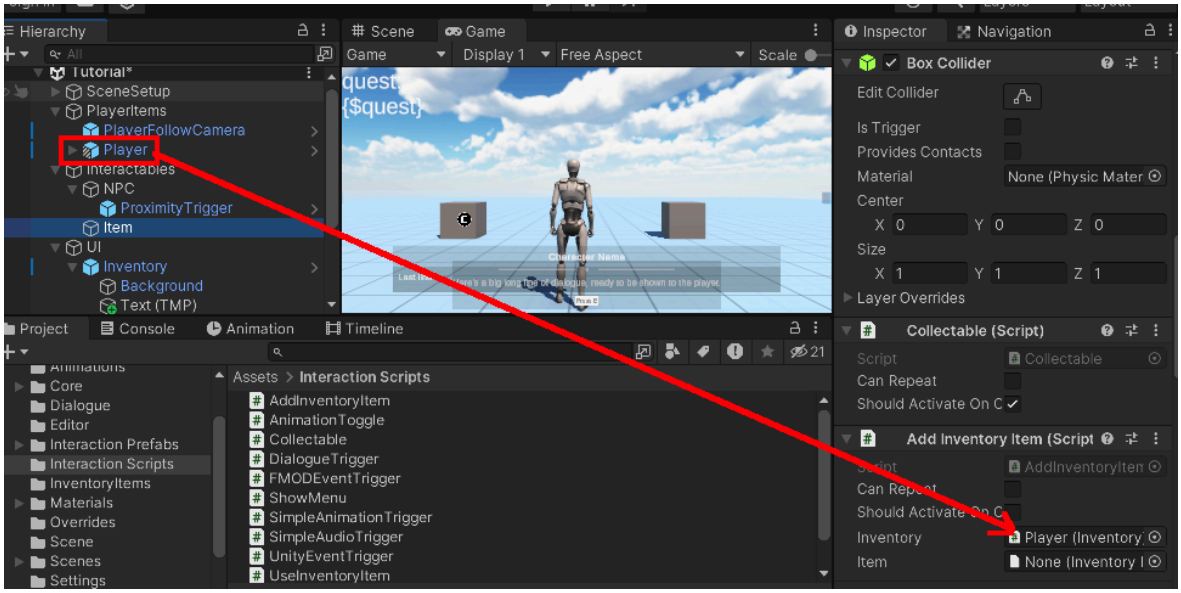
Check the “Should Activate On Collision” checkbox on the Collectable component.



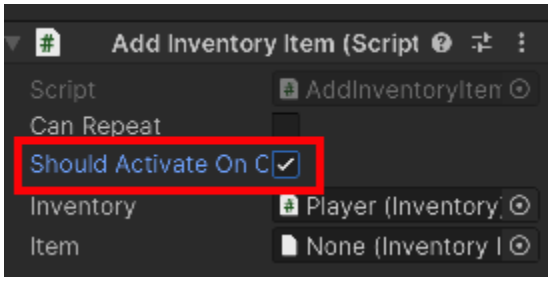
Add the AddInventoryItem script to the Item as well.



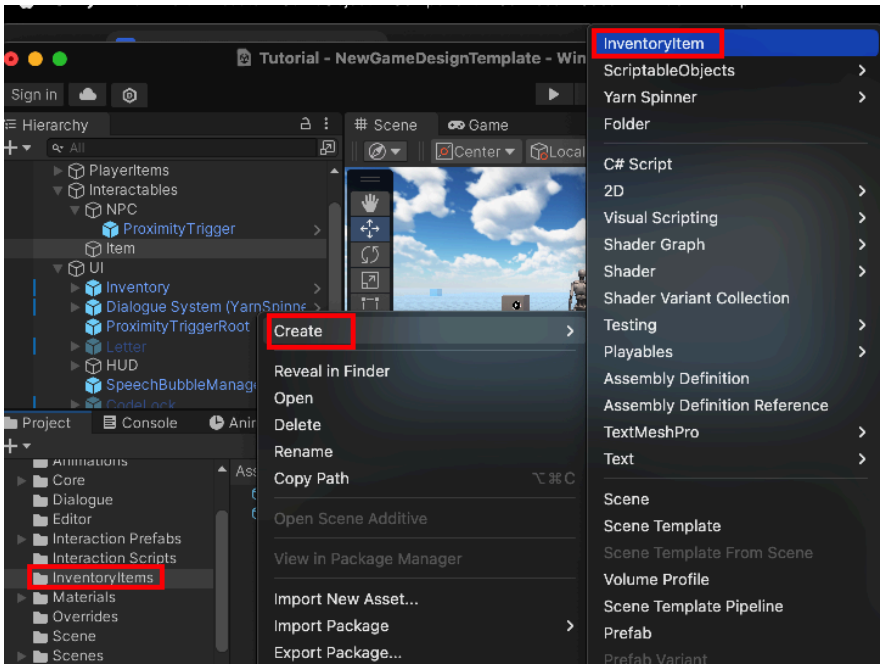
Drag the Player object from the PlayerItems node onto the “Inventory” property of the AddInventory component. This will ensure the items will end up in the Player’s inventory.



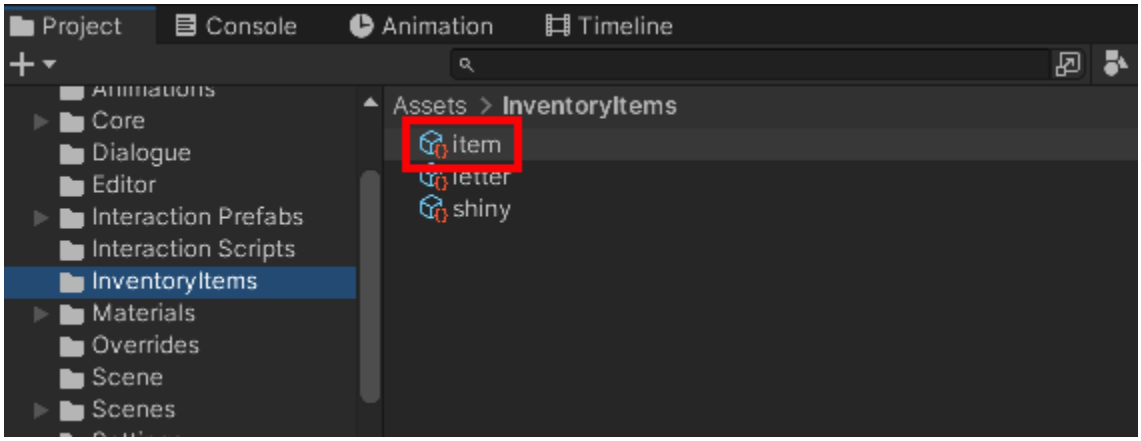
Check the “Should Activate On Collision” checkbox on the component.



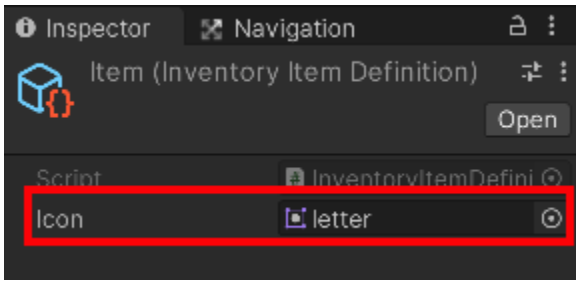
Open the folder Assets/InventoryItems and right-click the folder to create an InventoryItem. InventoryItems are used to define the name of the items and they define what icon should be shown in the inventory in the HUD.



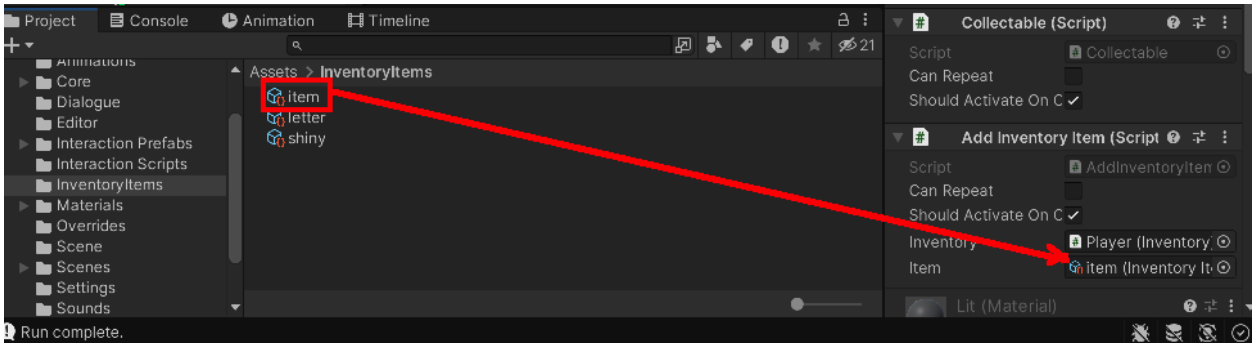
Name the InventoryItem “item” and select it. The name will be used in the YarnSpinner script to reference the item.



In the inspector, assign a sprite to the Icon property. This is the sprite that will be shown in the inventory view in the HUD.



Drag the newly created item onto the “item” property of the AddInventoryItem component.



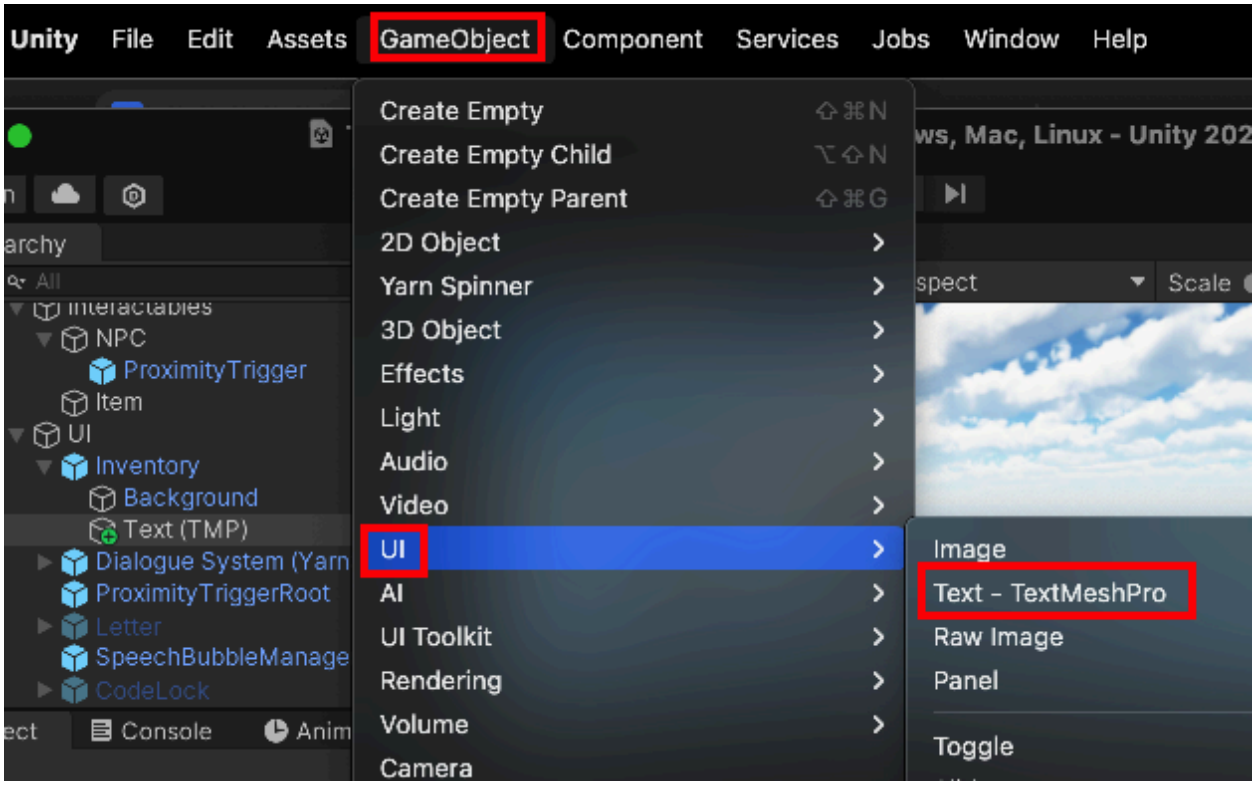
Now, when the player collects the item we created, it will appear in the inventory.

Now let’s set up the script. Locate the script in the “Assets/Dialogue” folder and open it.
In the first node of the project, we’ll add a variable called $quest.

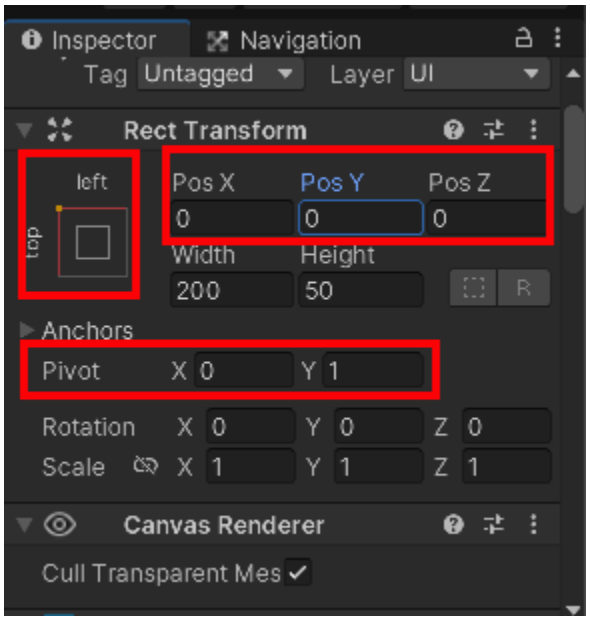
```
//---------------------------------------------------------------------------
//                                Start
//---------------------------------------------------------------------------
0 references | Show in Graph View
title: Start
---
<<declare $quest = "unassigned">>
===
```

The first value for the variable is “unassigned”. This will indicate the quest has not started yet. We will change it to “in progress” and “done” later on.

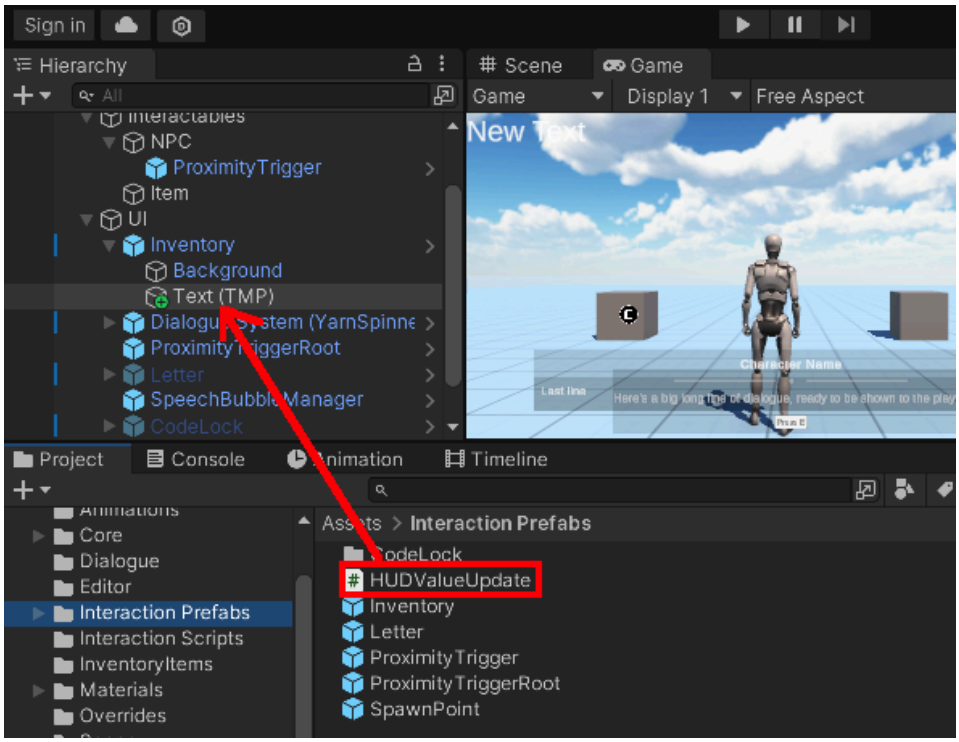
Add a TextMeshPro text object to the scene. This will be used to display the $quest variable.



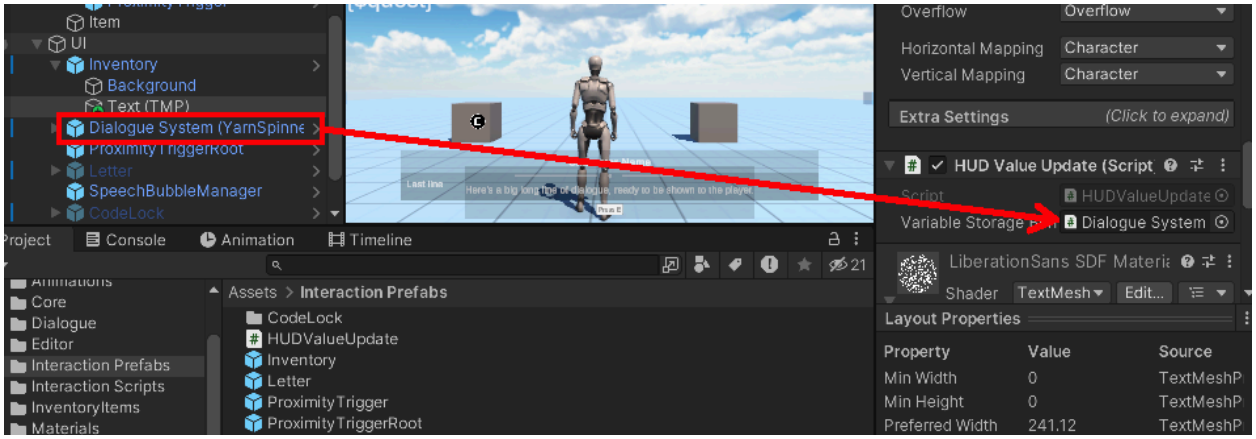
For this, go to the Unity's menu bar, then press GameObject->UI->Text - TextMeshPro.
In the inspector, change the origin of the object to 0, 1. Change the Anchor Preset to the top-left setup. Finally, change the position of the object to (0, 0). These settings are highlighted in the screenshot.



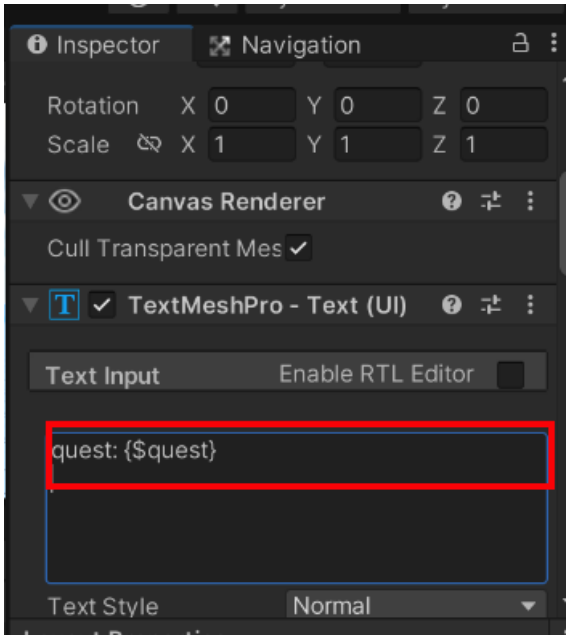
Drag the HUDUpdateValue script from the folder “Assets/Interaction Prefabs” onto the newly created text object.



Drag the Dialogue System node from the UI node in the hierarchy onto the HUDUpdateValue component’s field “variable storage behavior”.



In the inspector, change the text field of the TextMeshPro Text component into: Quest: {$quest}
The part that reads {$quest} will be replaced with the content of the variable from the YarnSpinner script. This will show the quest onto the screen for the player.

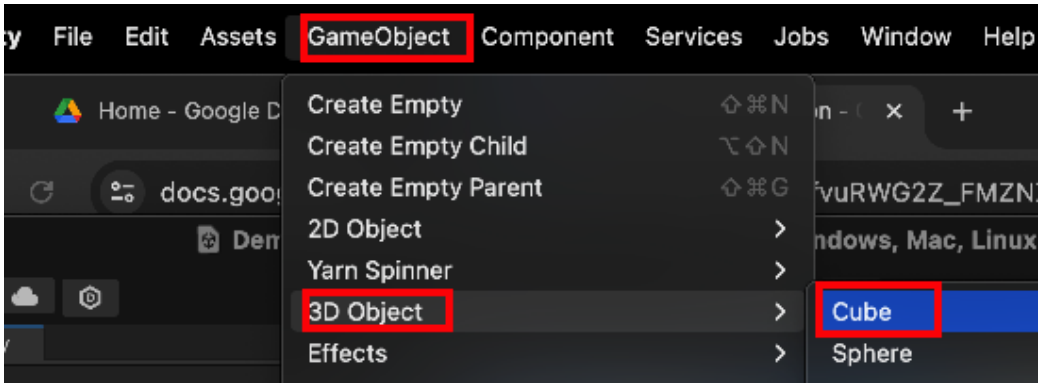


Open up the script again and add a new node called QuestGiver. When the user interacts with the NPC, the DialogTrigger will trigger this node.
In the QuestGiver node, we will check if the user has the mission. If not, we will give the quest. If they have the quest, but not the Item in their inventory, we will give them a reminder.

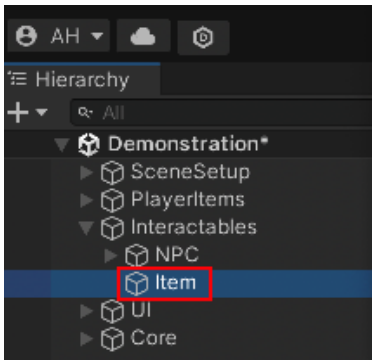
```
//---------------------------------------------------------------------------
//                                QuestGiver
//---------------------------------------------------------------------------
0 references | Show in Graph View
title: QuestGiver
---
<<if $quest == "unassigned">>
    NPC: Hello, traveler. Could you bring me the item?
    <<set $quest to "in progress">>
<<elseif $quest == "in progress">>
    <<if not has_item("Player", "item")>>
        NPC: Ah, did you fetch the item yet?
    <<else>>
        NPC: Thank you for doing the quest!
        <<set $quest to "done">>
    <<endif>>
<<elseif $quest == "done">>
    NPC: This quest is over.
<<endif>>
===
```

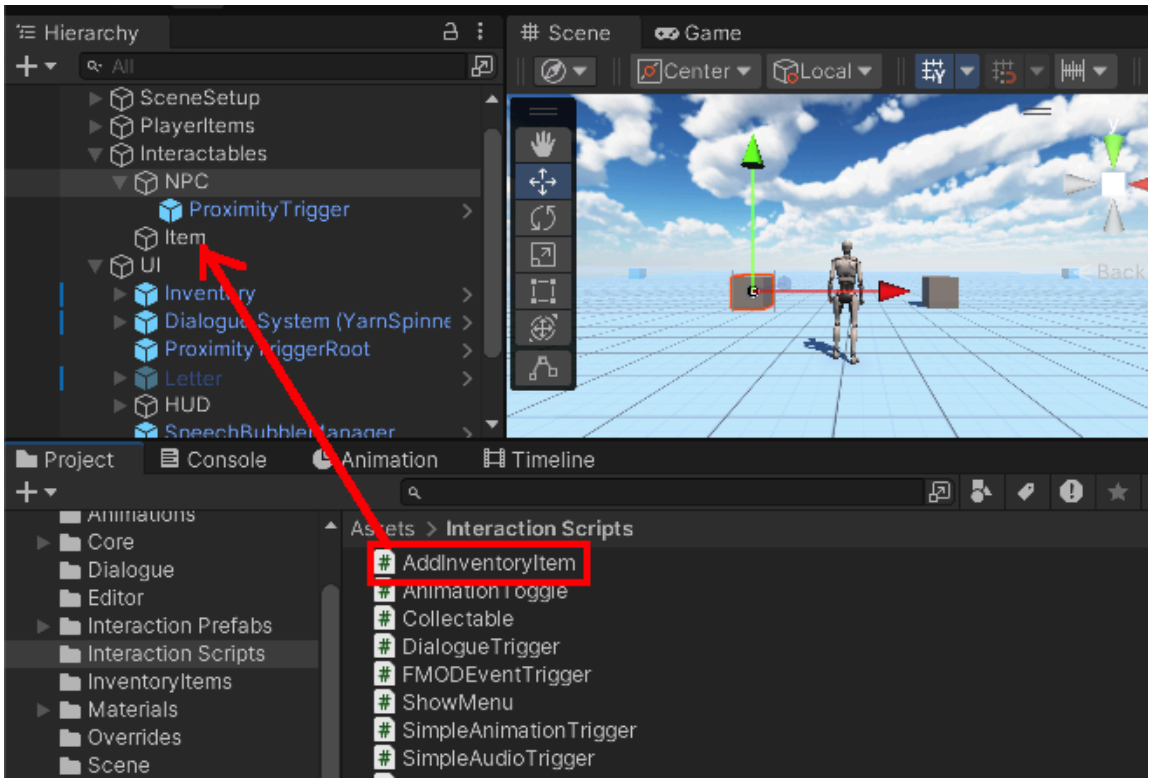
Adding item to inventory from trigger

Add a cube to the scene and name it “Item”.
For that, go to GameObject->3D Object->Cube

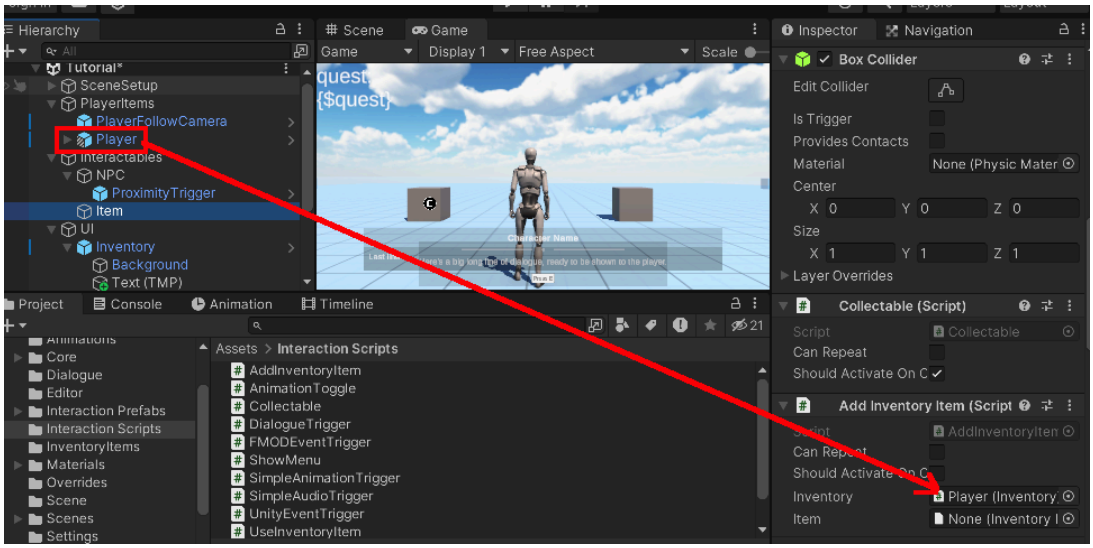


Rename the cube to “Item”.

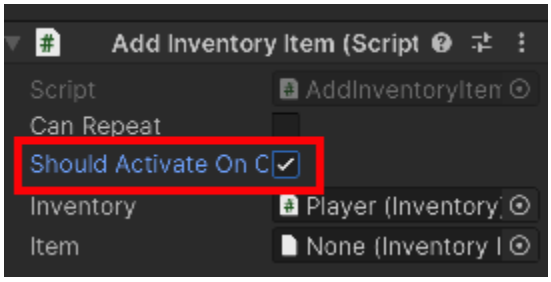




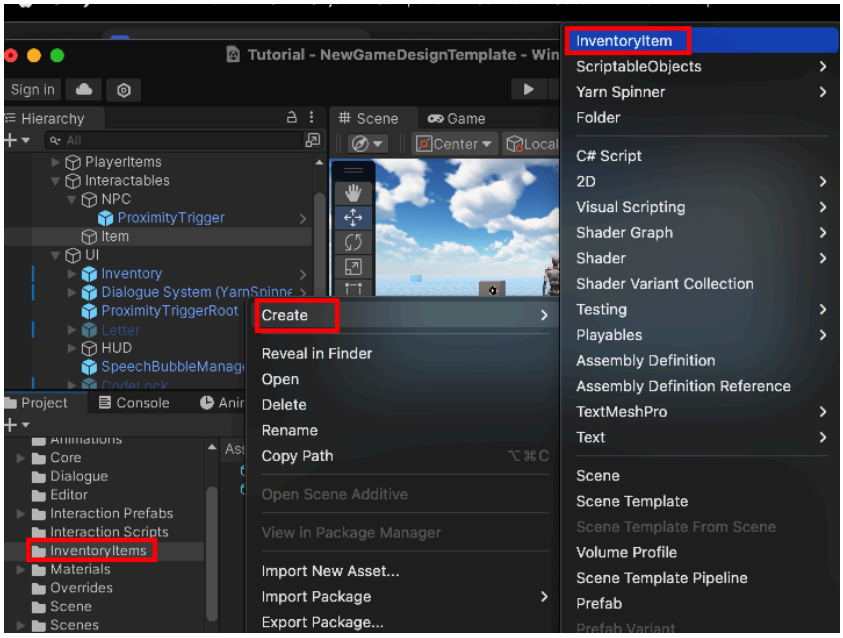
Drag the Player object from the PlayerItems node onto the “Inventory” property of the AddInventory component. This will ensure the items will end up in the Player’s inventory.



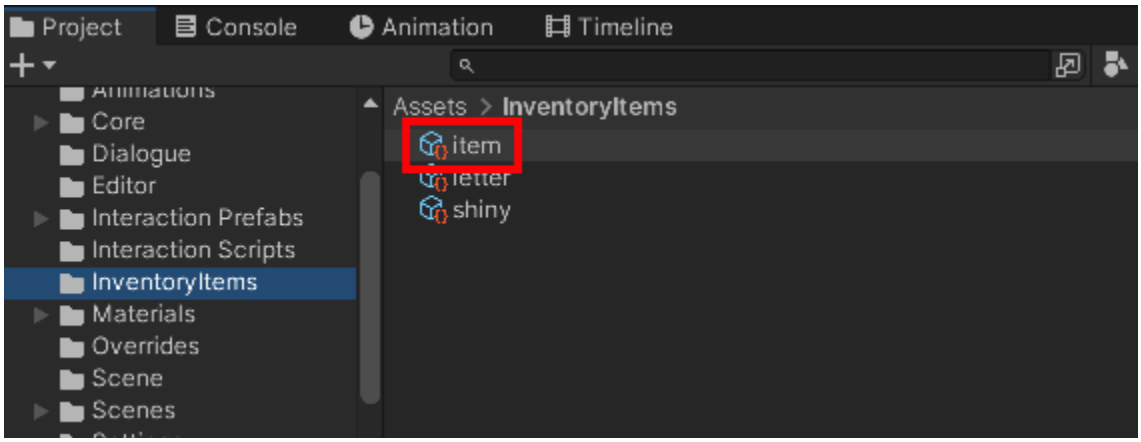
Check the “Should Activate On Collision” checkbox on the component.



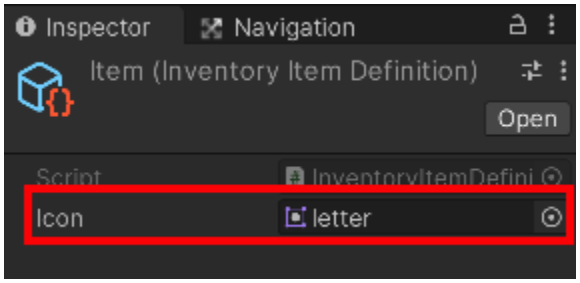
Open the folder Assets/InventoryItems and right-click the folder to create an InventoryItem. InventoryItems are used to define the name of the items and they define what icon should be shown in the inventory in the HUD.



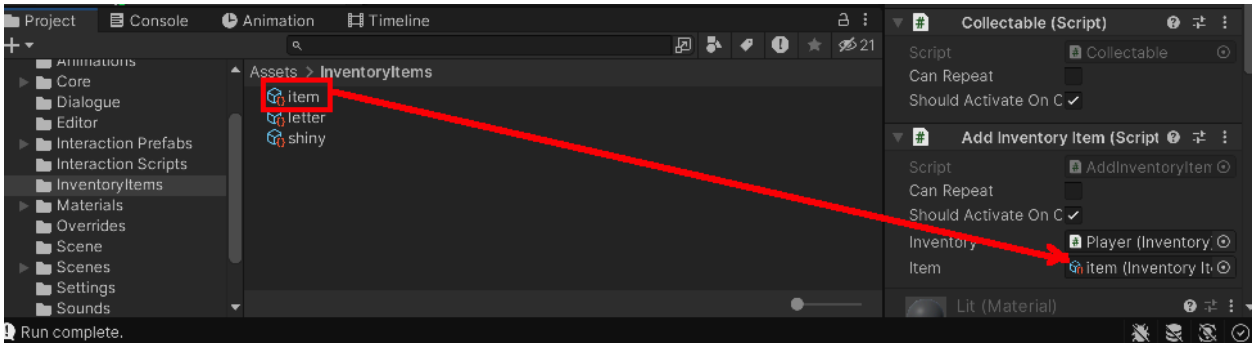
Name the InventoryItem “item” and select it. The name will be used in the YarnSpinner script to reference the item.



In the inspector, assign a sprite to the Icon property. This is the sprite that will be shown in the inventory view in the HUD.



Drag the newly created item onto the “item” property of the AddInventoryItem component.



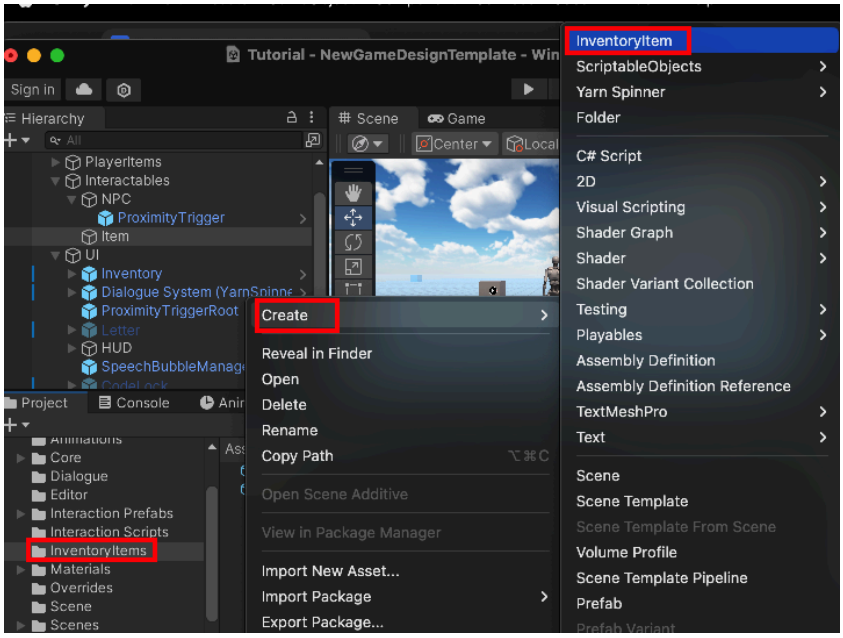
Now, when the player collects the item we created, it will appear in the inventory.

If you want the item to disappear after being collected, then add the Collectable component to it and check the “Should Activate On Collision” checkbox on the Collectable component.

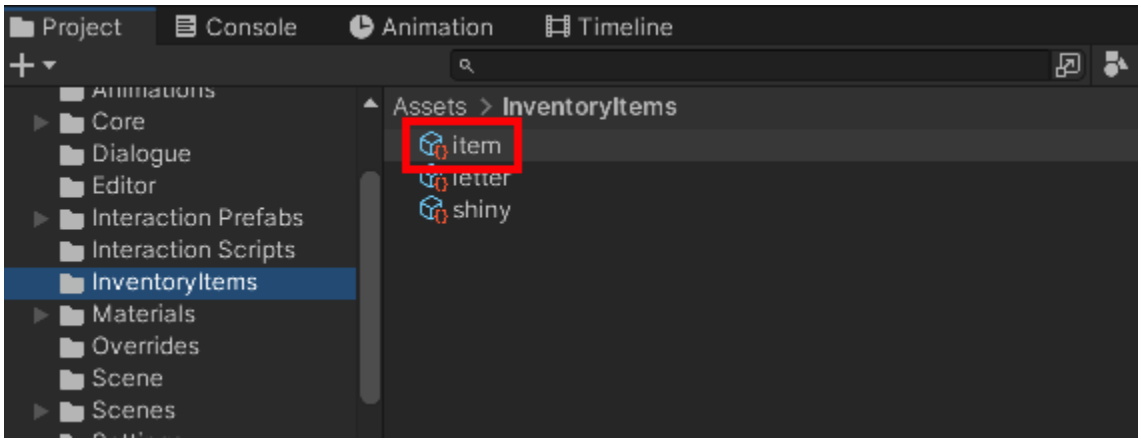
Adding item to inventory from script

Adding an inventory item from a script can be done with the give_item command. The system will show the item in the inventory using an icon. You need to define the icon using a ScriptableObject in the InventoryItems folder.

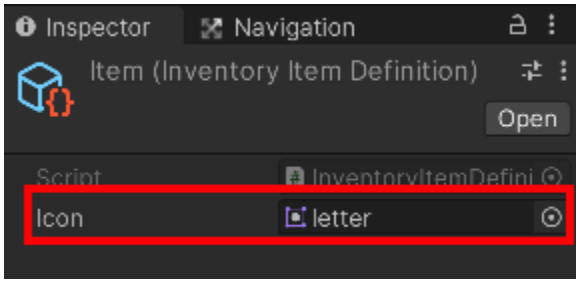
Open the folder Assets/InventoryItems and right-click the folder to create an InventoryItem. InventoryItems are used to define the name of the items and they define what icon should be shown in the inventory in the HUD.



Name the InventoryItem “item” and select it. The name will be used in the YarnSpinner script to reference the item.



In the inspector, assign a sprite to the Icon property. This is the sprite that will be shown in the inventory view in the HUD.



In the node where you want to give the item to the player, add the following script:

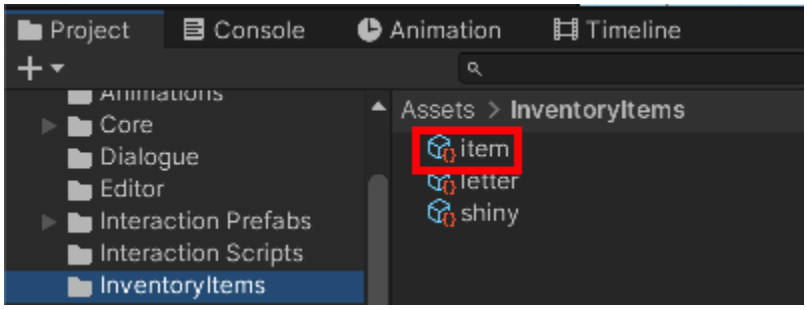
```
//---------------------------------------------------
//                                        Start
//---------------------------------------------------
0 references | Show in Graph View
title: Start
---
<<give_item "Player" "item">>
===
```

Removing item from inventory from script

In the node where you want to take an item from the inventory, add the following script:

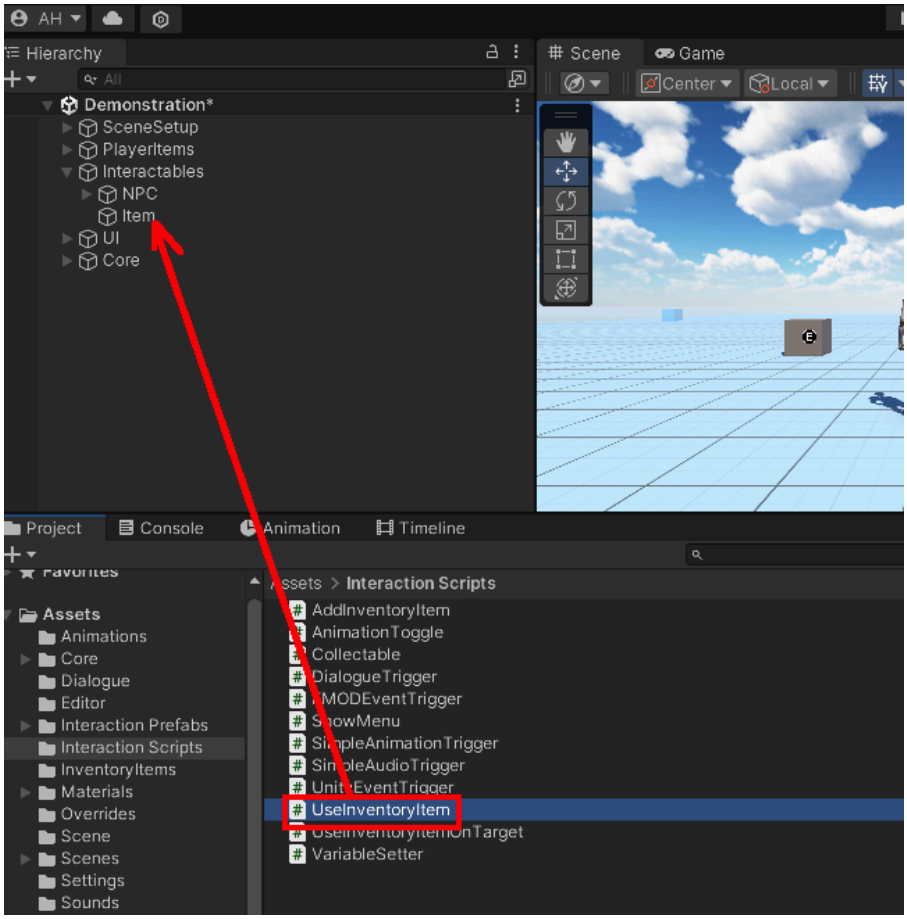
```
//---------------------------------------------------
//                                      QuestGiver
//---------------------------------------------------
0 references | Show in Graph View
title: QuestGiver
---
<<take_item "Player" "item">>
===
```

Make sure the name of the item matches the one in the “Assets/InventoryItems” folder:

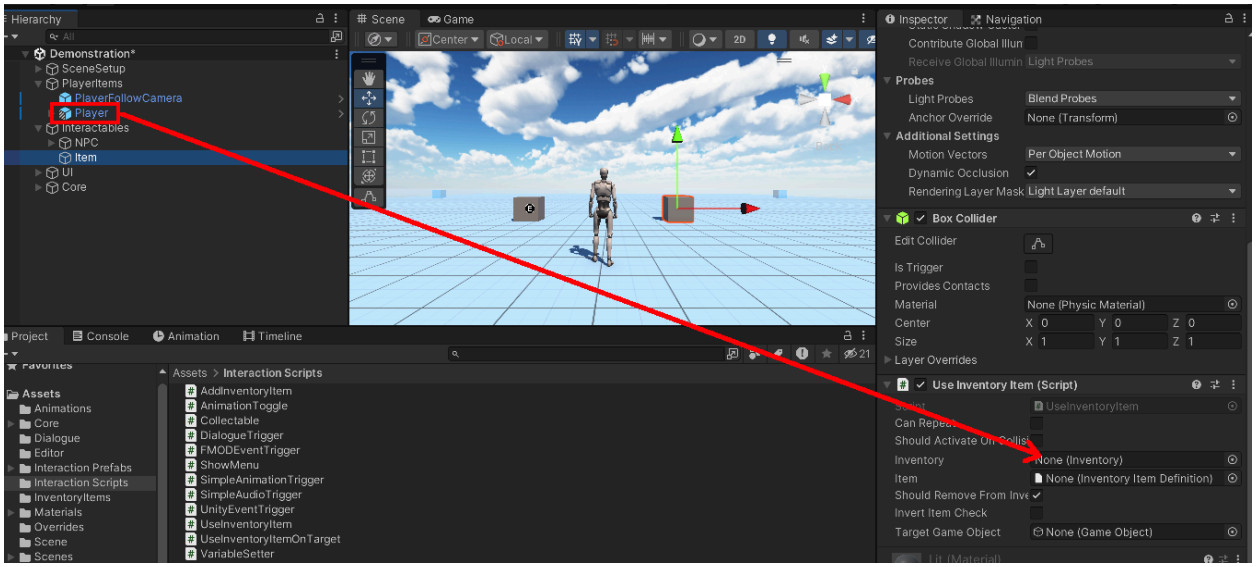


Checking and removing item from inventory from trigger

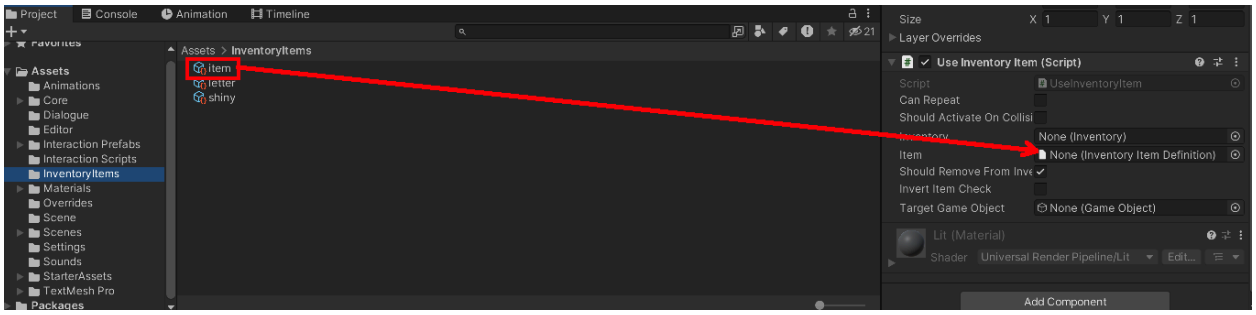
Locate the script UseInventoryItem in the folder “Assets/Interaction Scripts” and drag it into the item that you want to use for checking the item in the inventory.



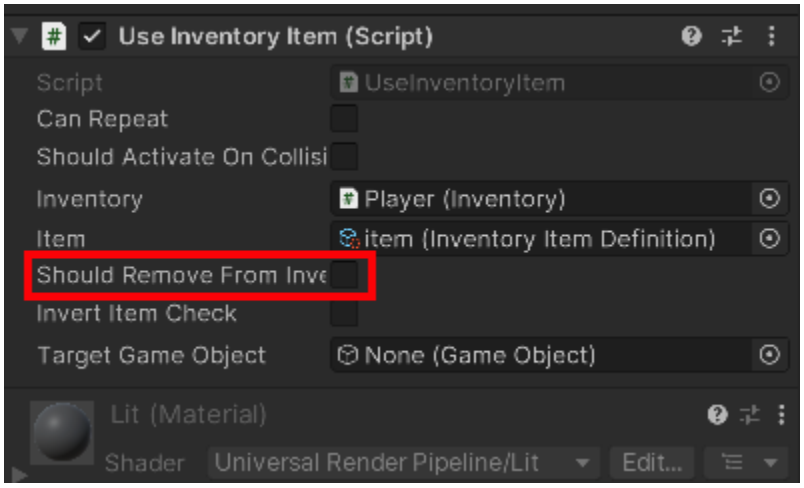
Drag the player GameObject from the PlayerItems node in the inventory onto the “Inventory” property of the UseInventoryItem component.



Locate the item that you want to check for in the “Assets/InventoryItems” folder and drag it onto the “Item” property of the component.



The UseInventoryItem component will disable all other PlayerInteractables until the user has the specified object in their inventory. Optionally, you can uncheck the box that says “Should remove from inventory”. This will keep the item in the inventory after usage.



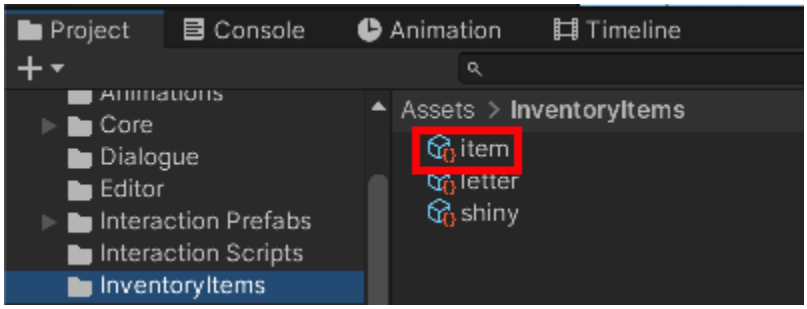
The “Invert Item Check” Checkbox will make the component check the inventory for *not* having the item.

Checking for inventory item from script

To check for an item in the script, you can use the “has_item” function.

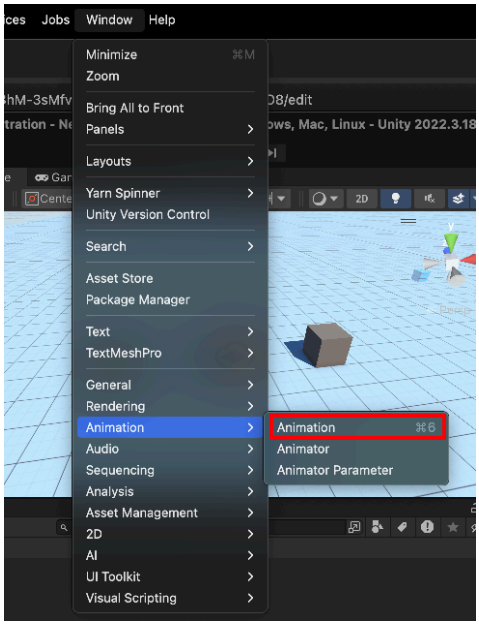
```
//---------------------------------------------------------
//                                          QuestGiver
//---------------------------------------------------------
0 references | Show in Graph View
title: QuestGiver
---
<<if has_item("Player", "item")>>
    NPC: You have the item!
<<else>>
    NPC: You don't have the item
<<endif>>
===
```

The name of the item should match the name of the one in the “Assets/InventoryItems” folder:

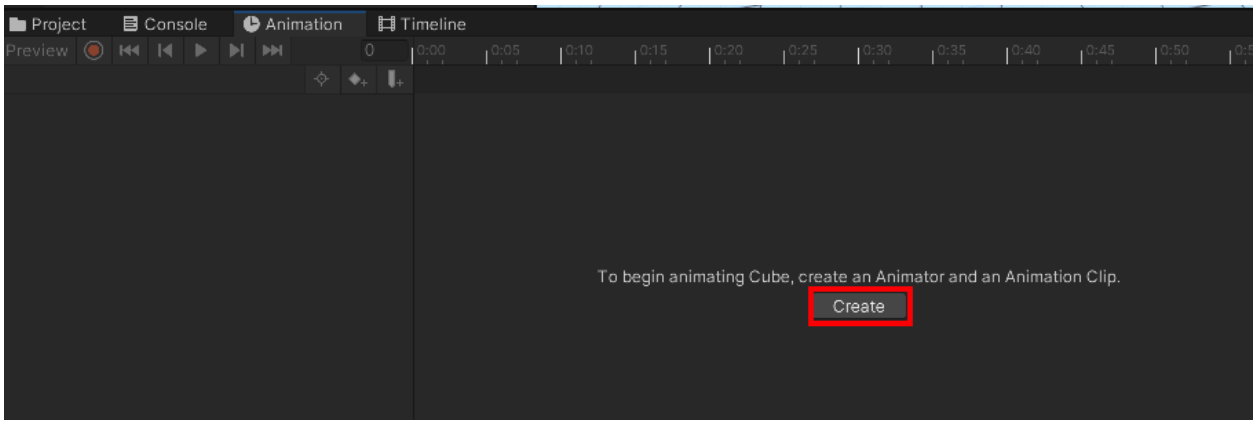


Creating and triggering animation from trigger

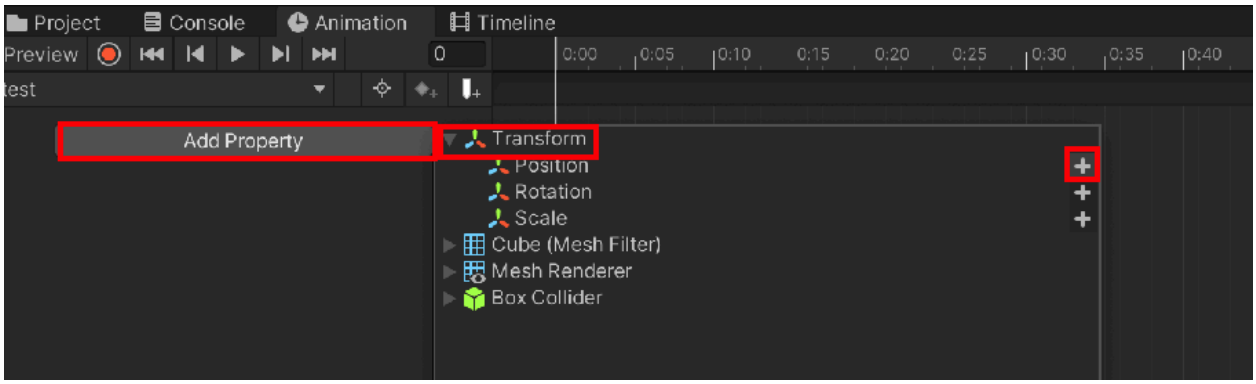
Select the gameobject that you want to animate in the hierarchy and open the Animation window. (CTRL/CMD+6)



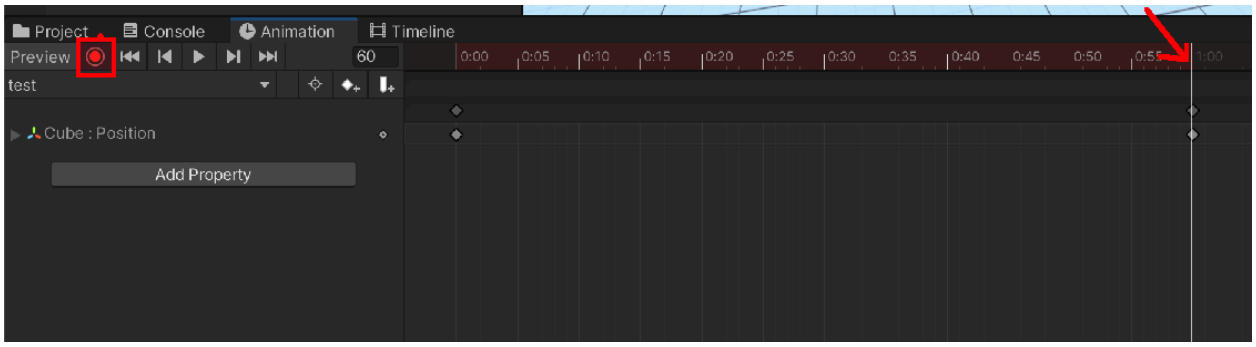
In the Animation window, press the button that says “Create” and save the animation. Remember where you save the animation, because you need to locate it later on.



Press “Add Property”, select “Transform” and press the plus-sign that is on the right, next to position.

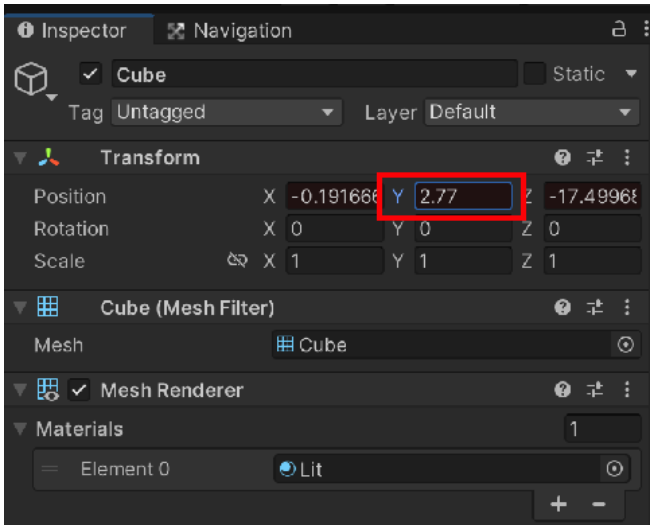


This will allow you to animate the object’s position.

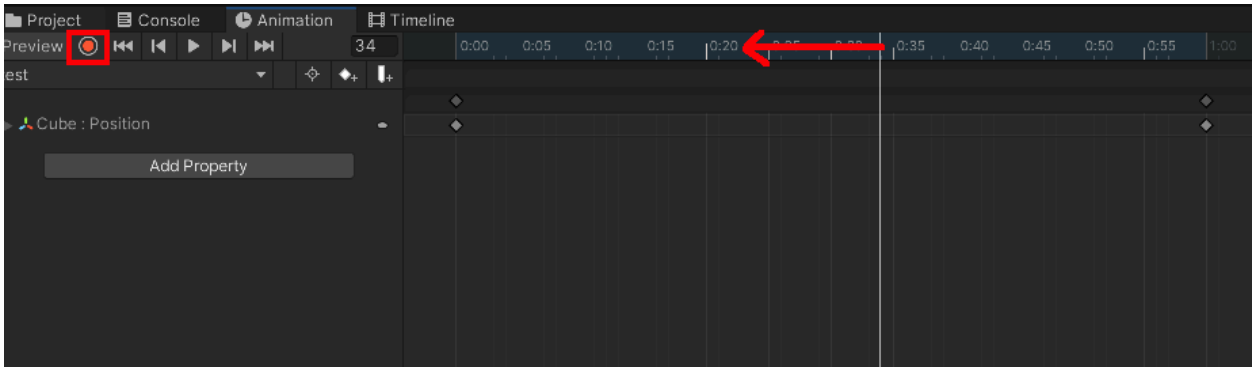


Press the record button and move the play head to the second keyframe.

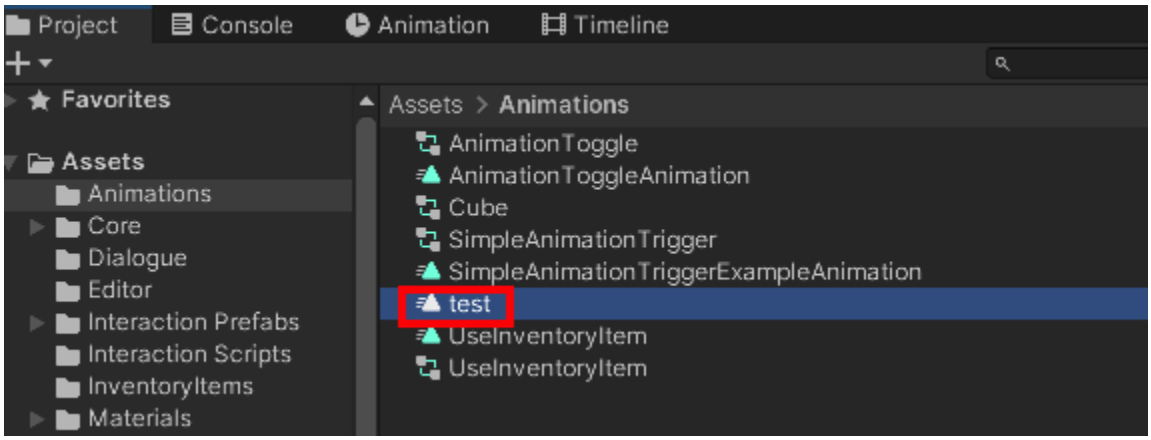
In the inspector, change the y-position of the GameObject.



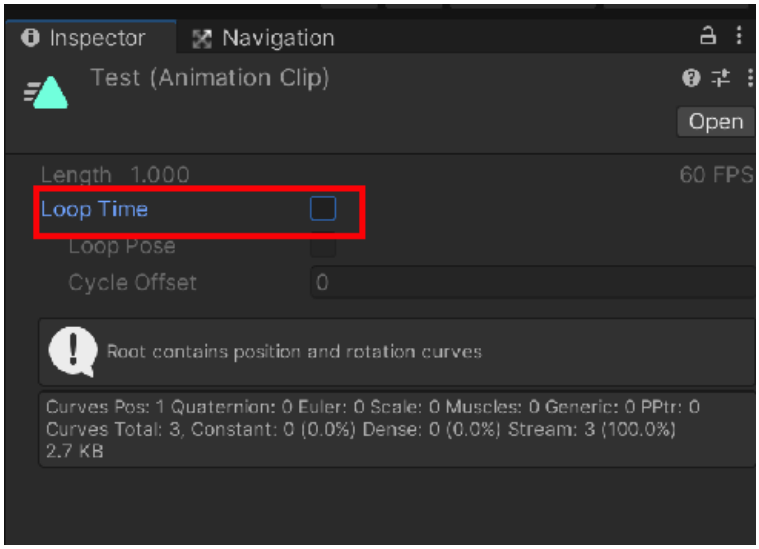
Toggle the record button to stop recording and scrub the playhead to test the animation.



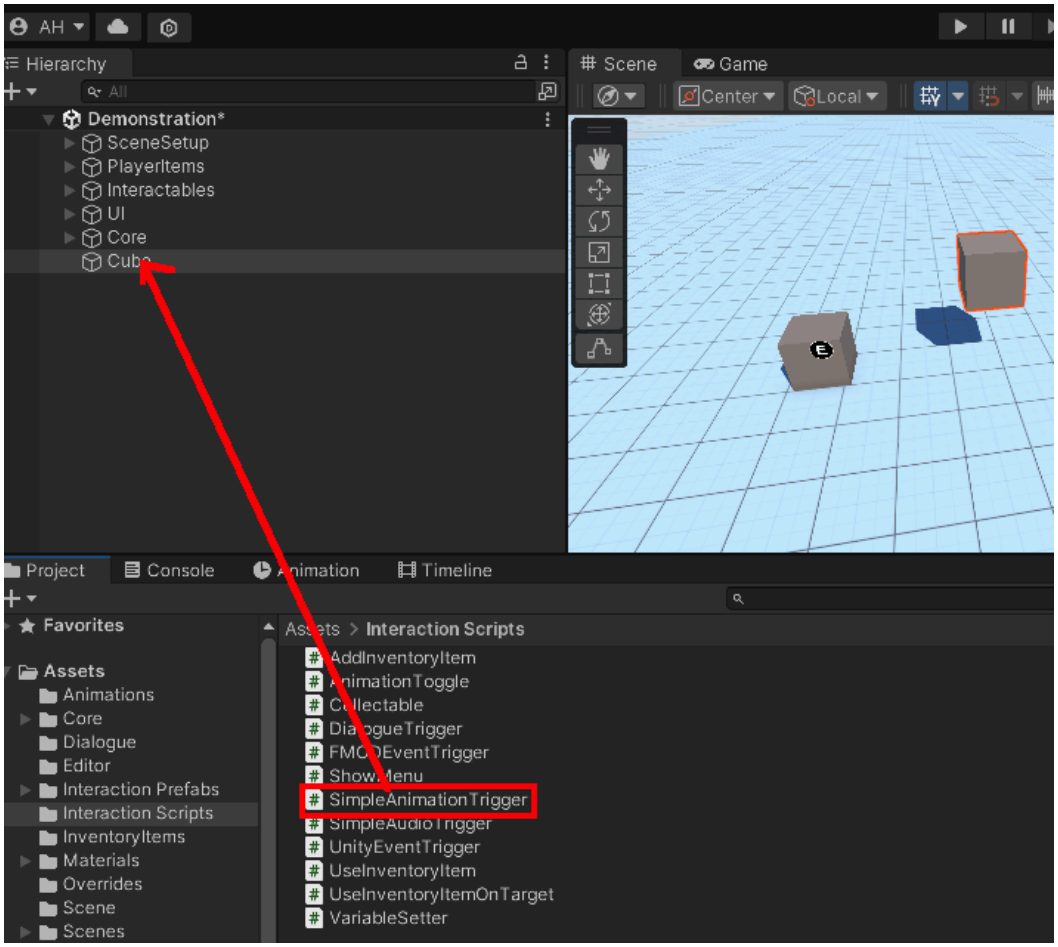
The GameObject should move between the old and new position as you scrub the playhead. Find the animation that you saved in the Assets folder and select it.



In the Inspector window, uncheck “Loop Time”. This will prevent the animation from looping.



Locate the SimpleAnimationTrigger in the folder “Assets/Interaction Script” and drag it onto the GameObject.

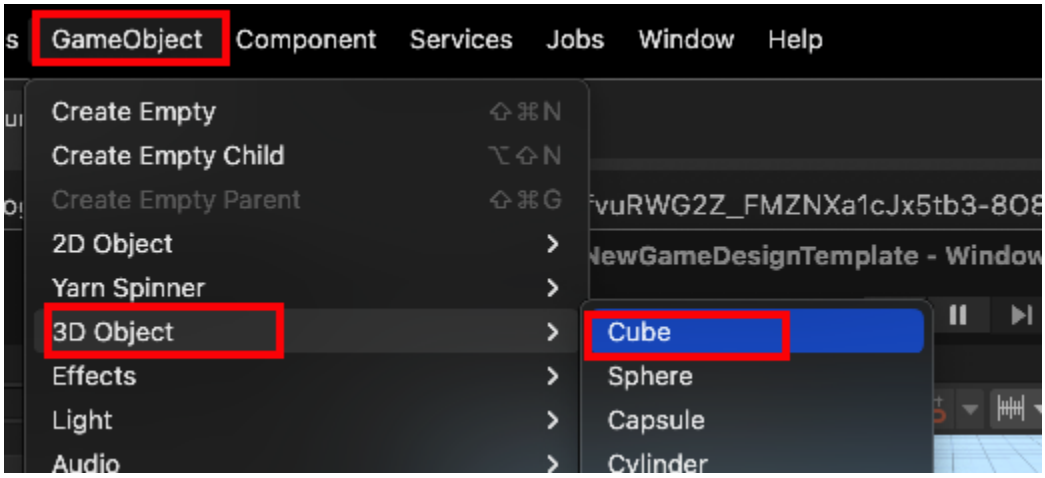


Drag the ProximityTrigger from the “Assets/Interaction Prefabs” onto the GameObject. Now when the player activates the item, the animation should play.

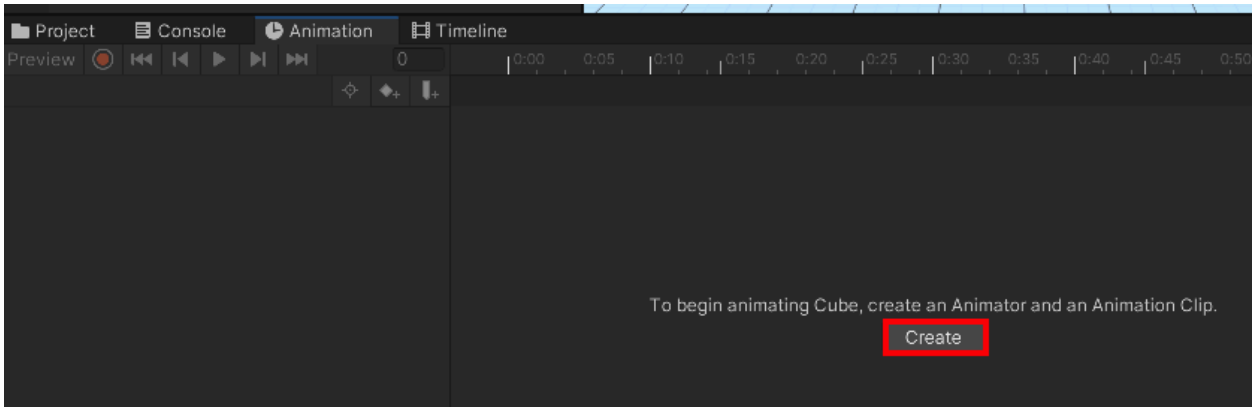
Toggle animations

Toggling animations allows for multiple animations to be sequenced and repeated. For instance opening and closing a drawer: each time the player interacts with the drawer, it toggles between the open and close animation.

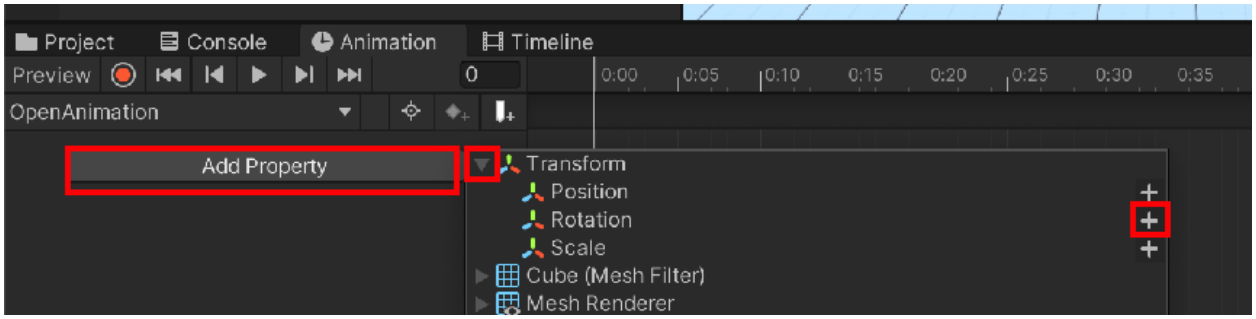
Create a new GameObject in the scene.



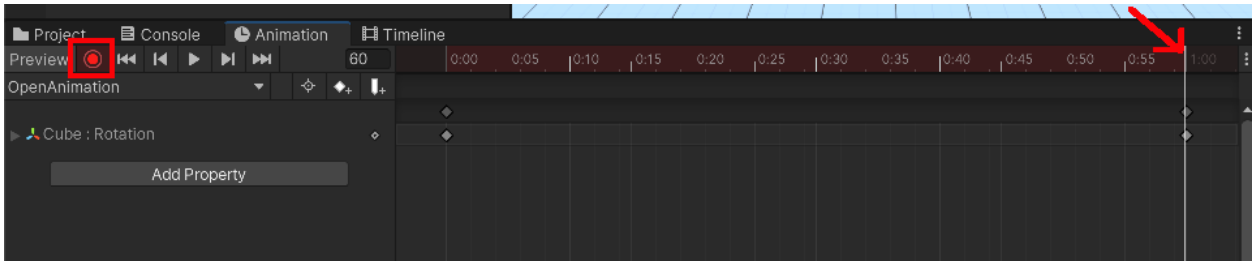
Select the item in the hierarchy and open the Animation window (CTRL/CMD+6)
Press the Create button to create an animation for the item and save it as OpenAnimation.
This animation should depict opening the door.



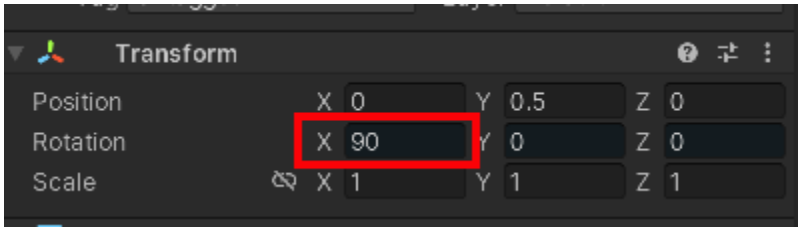
Press Add Property and add the rotation property.



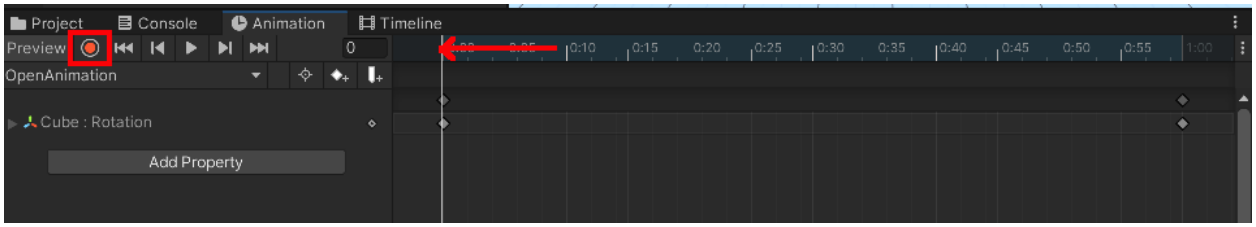
Press the record button and move the playhead to the second keyframe.



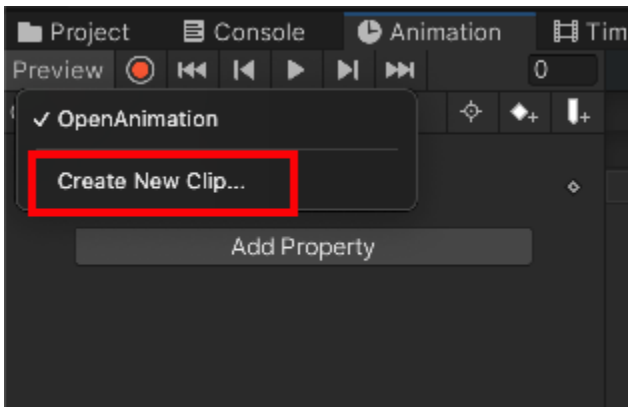
In the inspector, rotate the GameObject 90°. For instance, this could be a door that opens.



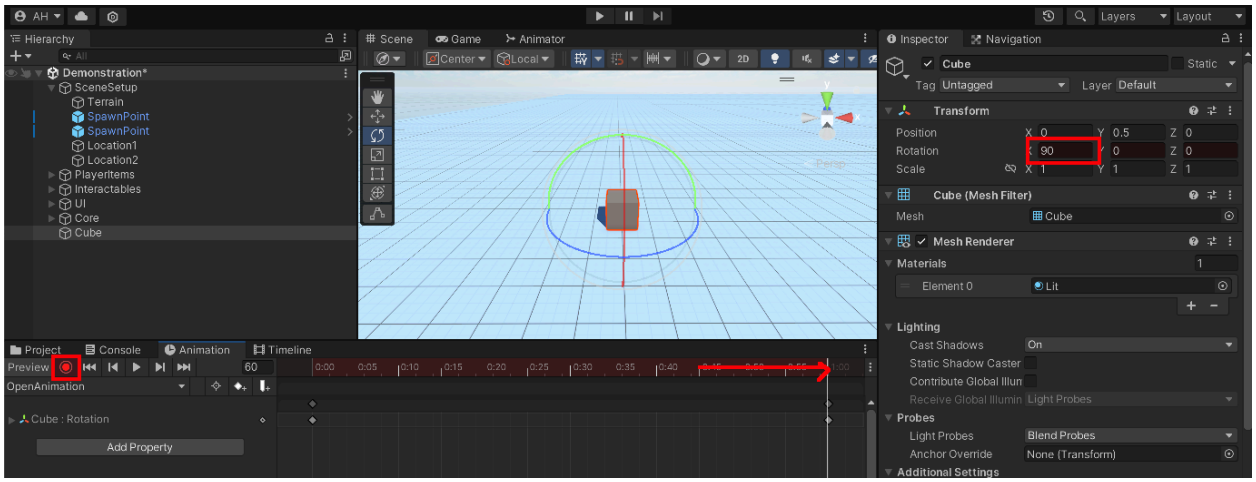
Press the record button again to stop recording and scrub the playhead back to the start to verify the animation. The object should turn back into its original position.



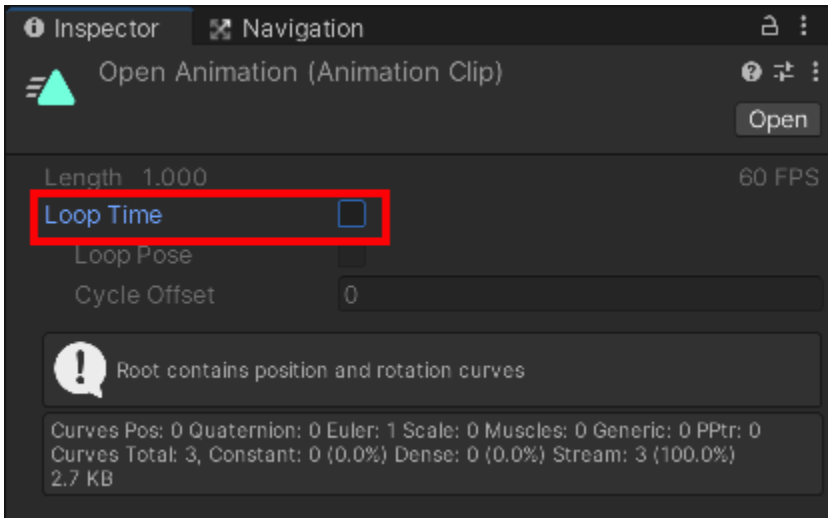
In the animation window, click the name of the current animation and in the dropdown, press “Create New Clip”. Save this animation as CloseAnimation.
This animation should depict closing the door.



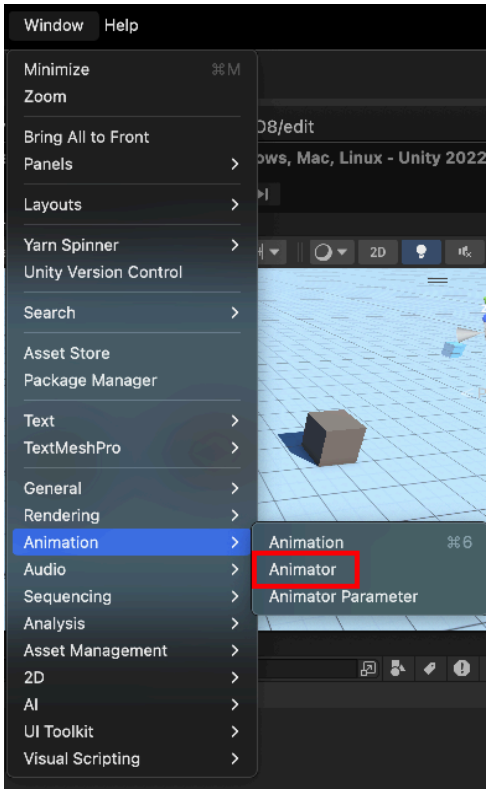
Repeat the same steps as before: press the record button, move the playhead to the second keyframe and change the rotation in the inspector.



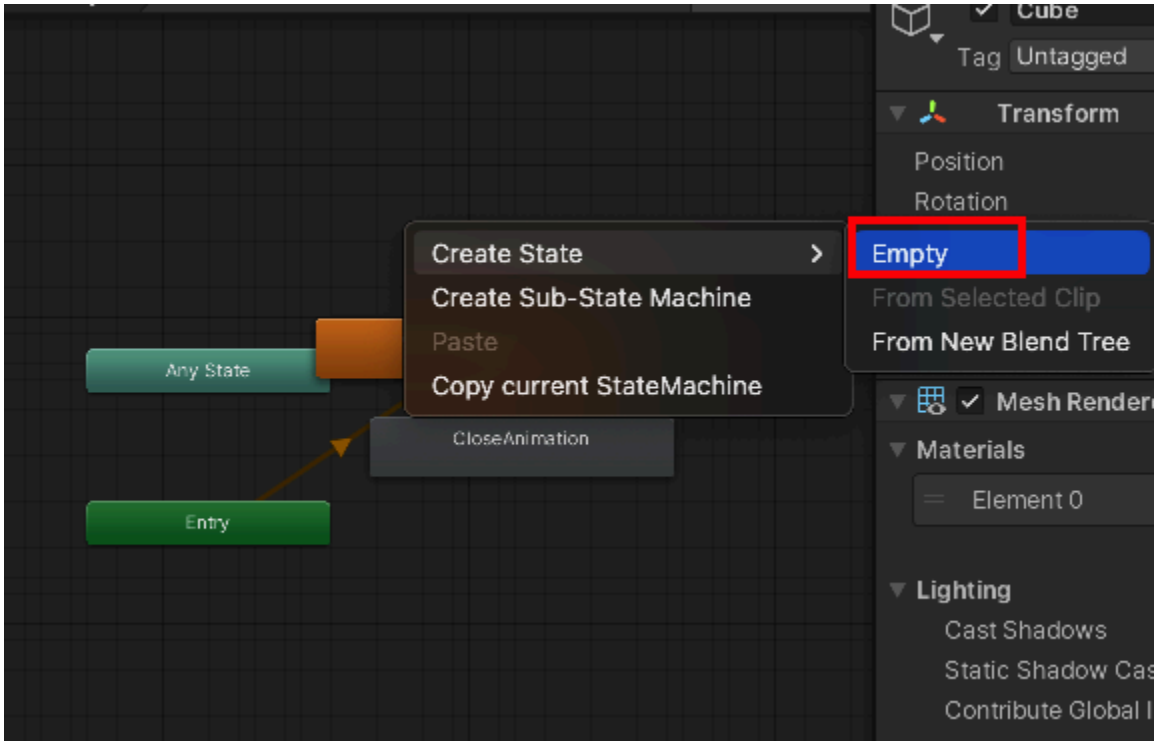
Press the record button again to end recording the animation.
Locate both animation clips in the Assets folder and select them. In the inspector, disable “Loop time” for both of them.



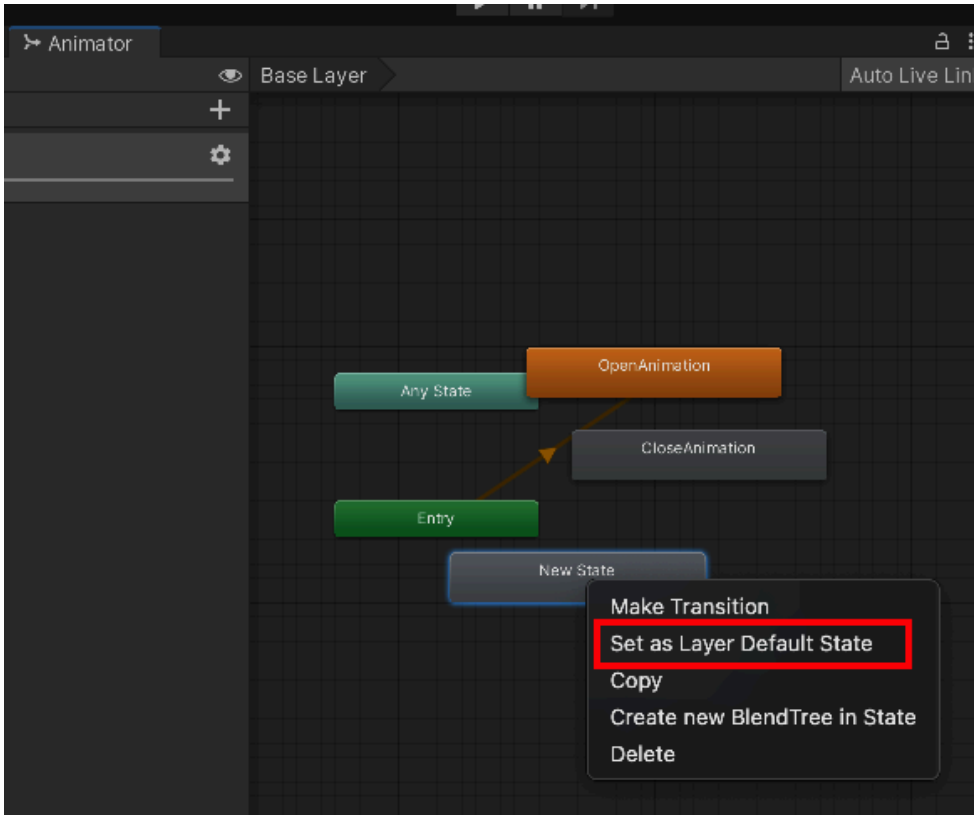
Now go to Window->Animator.



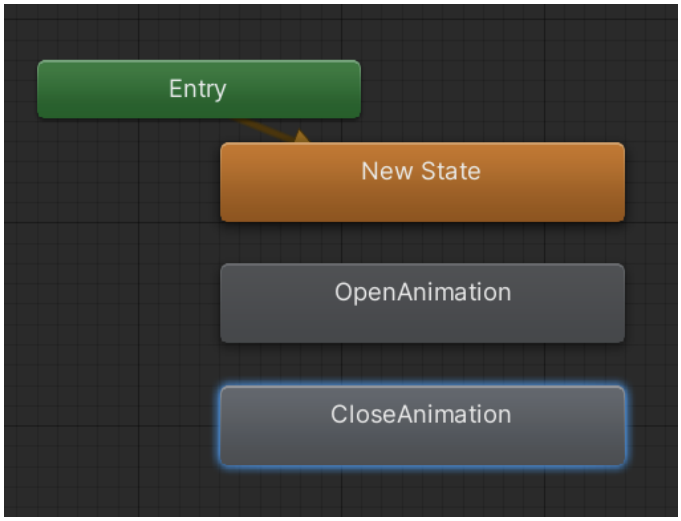
In the Animator window, right-click the open area to create a new, empty animation state.



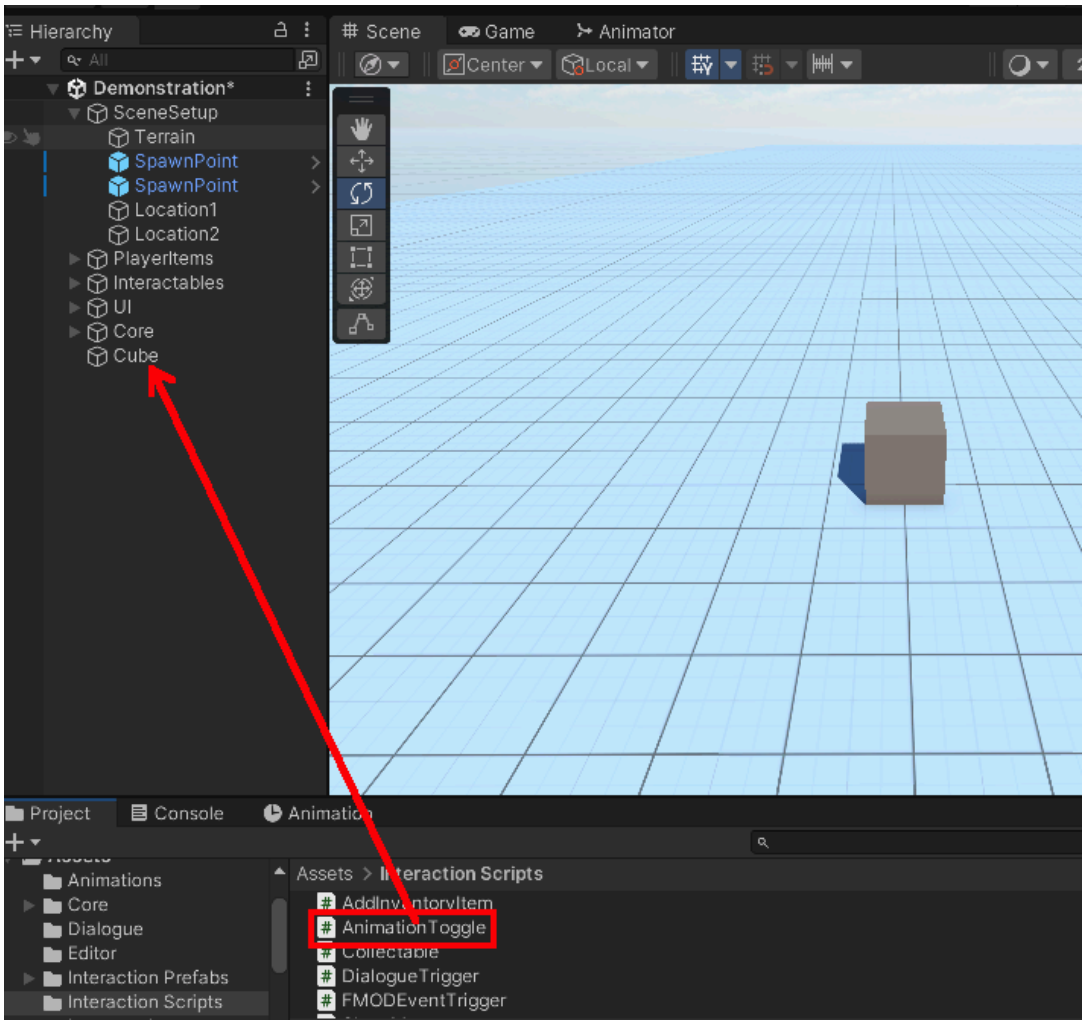
Right-click the newly created state and choose “Set as Layer Default State”



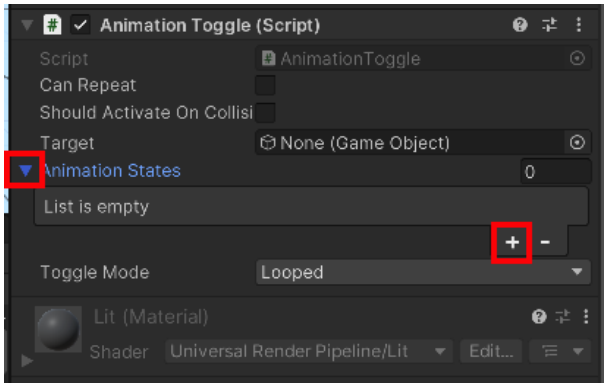
You can rearrange the layout a bit for clarity:



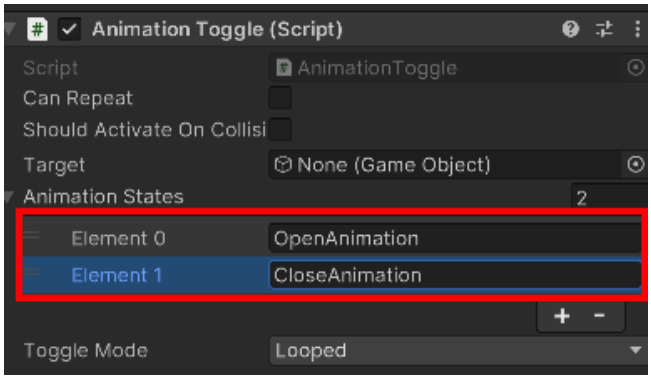
In the “Assets/Interaction Scripts” folder, locate the AnimationToggle script and drag it onto the GameObject in the hierarchy:



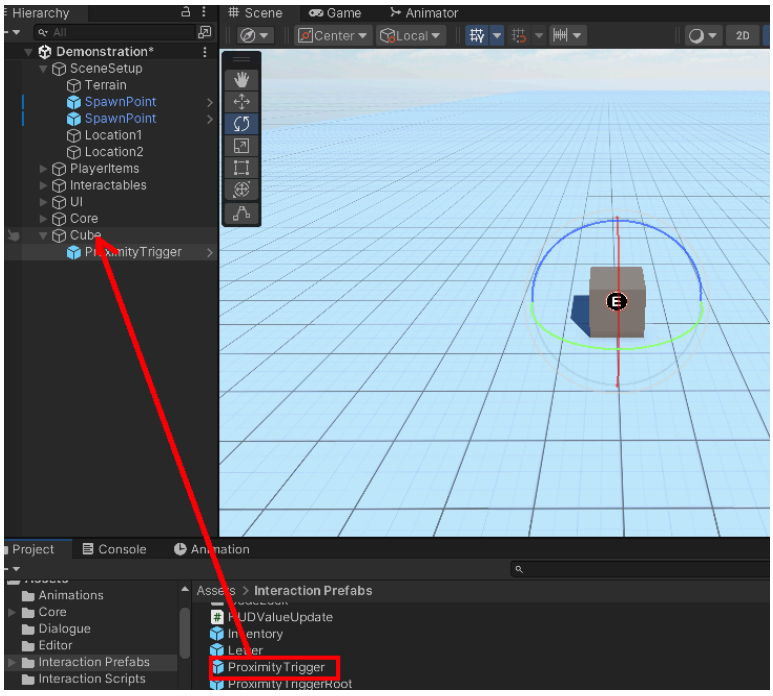
In the inspector, locate the AnimationToggle component and open the Animation States list. Use the plus sign on the right to add slots.



Fill in the names of the two animations you created before



Finally, locate a ProximityTrigger in the “Assets/Interaction Prefabs” folder and drag it onto the GameObject.



The object should now toggle between the two animation states.

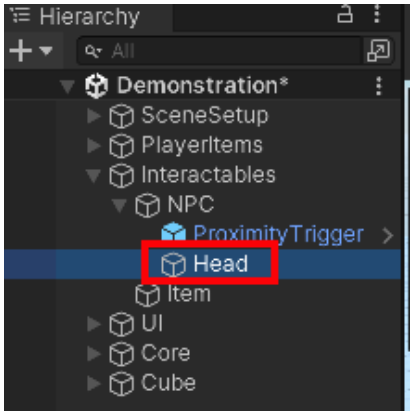
Freezing/unfreezing the player

It could be useful to temporarily disable the player movement. For instance if you want to show a cut scene. For that, two commands can be utilized in the script. Here is a short example how to use it:

```
//---------------------------------------------
//                                      Start
//---------------------------------------------
0 references | Show in Graph View
title: Start
---
You will be stunned for 5 seconds.
<<disable_controls Player>>
<<wait  5>>
<<enable_controls Player>>
You can now move again.
===
```

Adding a speech bubble

To add a speech bubble, you can use the “say” command. For completion: speech bubbles are processed by the SpeechBubbleManager GameObject that must be placed in the scene. This is the default in the current project setup.
The speech bubble will be placed directly above the GameObject. If that is an NPC, the speech bubble would look as if it is placed at the center of its head. A typical setup would be to add a GameObject as a child to the NPC and use that for placing the speech bubbles.



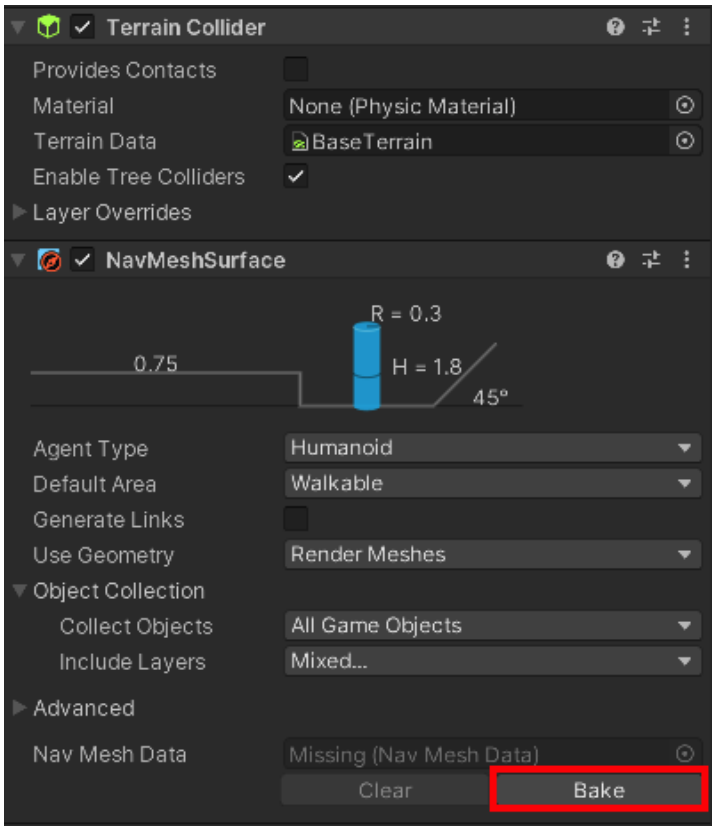
In code, this item can be referred to by name:

```
//---------------------------------------------
//                                      Start
//---------------------------------------------
0 references | Show in Graph View
title: Start

<<say "Head" "Hello I am a speech bubble!">>
===
```

Making NPC walk somewhere

In the current setup, there is an NPC. The NPC is a humanoid GameObject that has a “WalkToBehaviour” added to it. You can control this behavior from a script.
In order for this behavior to work properly you need to bake a NavMesh. That is data the Unity uses to navigate through the game world without running into obstacles.

Baking the NavMeshSurface
In the hierarchy, locate the node “Terrain” in “SceneSetup”. In the inspector, locate the NavMeshSurface component and press the Bake button.



To make the NPC walk to a location, you can use this code:

```
//---------------------------------------------------
//                                   Start
//---------------------------------------------------
0 references | Show in Graph View
title: Start
---
<<walk_to "NPC" "Location1">>
===
```

This will make the NPC walk towards the gameobject called “Location1”.

By default, the script will wait for it to finish. You can add an extra property to immediately continue your YarnSpinner script.

```
title: Start
---
<<walk_to "NPC" "Location1" 0 false>>
This message will show immediately
===
```

The number 0 in this script determines at what distance the NPC will stop walking when approaching the target.

Highlighting a path to somewhere

It can be useful to show a path through the game world to guide the player. The show_path helper function is made for that. You can show a path between two objects.
This command uses the NavMesh data, so you must first bake the NavMeshSurface as shown in the previous topic.
To show a path from one GameObject to another, you can use the following code:

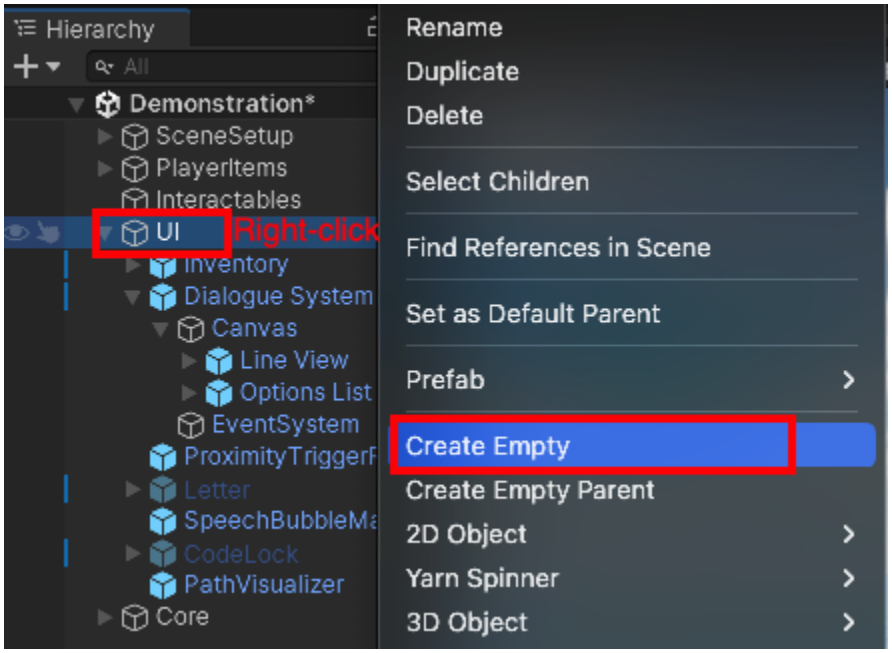
```
//---------------------------------------------------
//                                  Start
//---------------------------------------------------
0 references | Show in Graph View
title: Start
---
<<show_path "Player" "Location1">>
===
```

By default, the path will disappear when the first object approaches the second one.
However, if you want to explicitly hide the path, you can use the hide_path command.
If you want to use that, you should also set the “shouldWait” property to “false”.

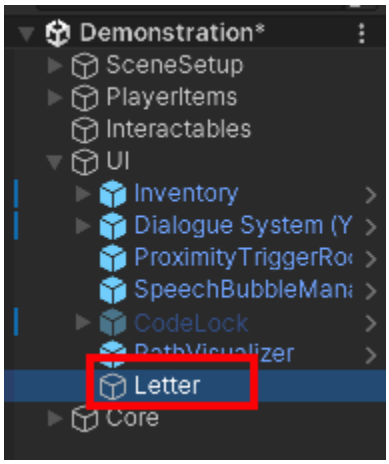
```
//---------------------------------------------------
//                                  Start
//---------------------------------------------------
0 references | Show in Graph View
title: Start
---
Reach the location in 4 seconds!
<<show_path "Player" "Location1" false>>
<<wait  4>>
<<hide_path>>
<<if get_distance("Player", "Location1") < 1>>
    You are close!
<<else>>
    You are not close!
<<endif>>
```

Showing a UI screen

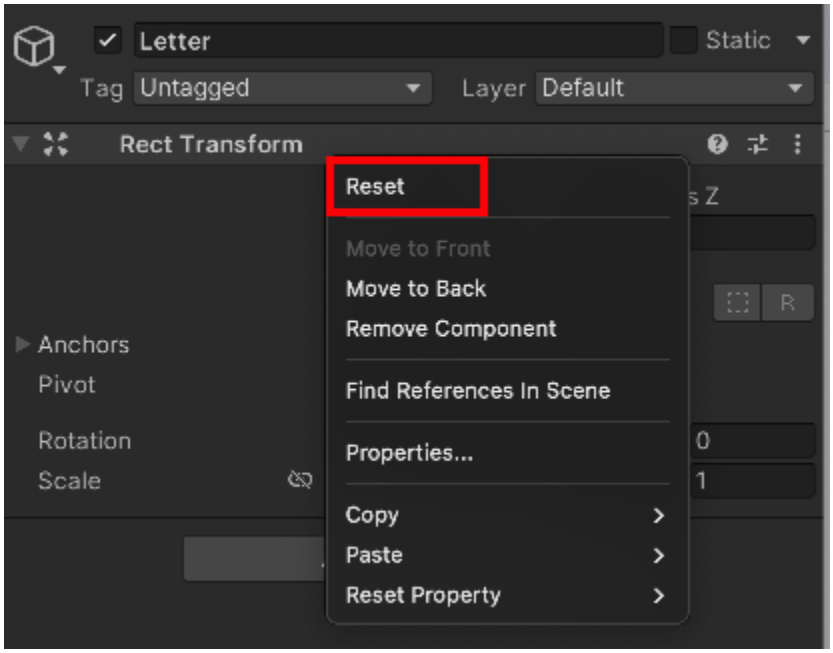
It can be useful to show a UI screen when the user activates a trigger. This could for instance be a piece of paper that they pick up. The UI screen could show the item in close up.
First, we create a GameObject to hold this screen. Right-click the hierarchy and add a new GameObject. You can add it to the UI node if you prefer that.



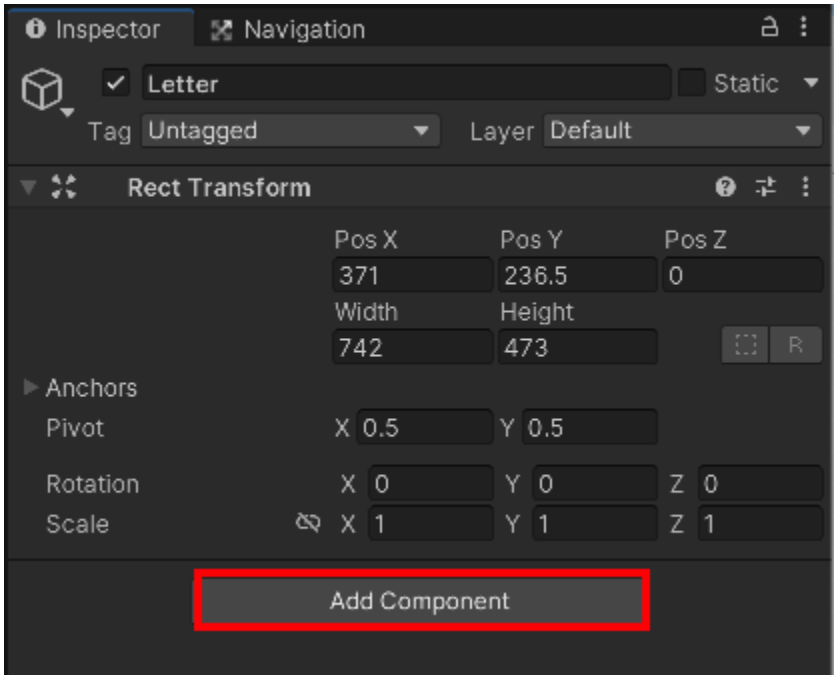
Rename the newly created item to “Letter”.

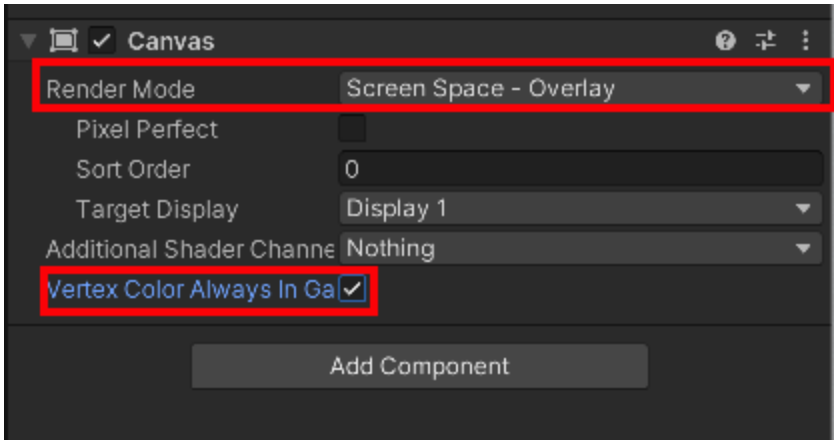


In the inspector, right-click the Rect Transform component and reset its values.

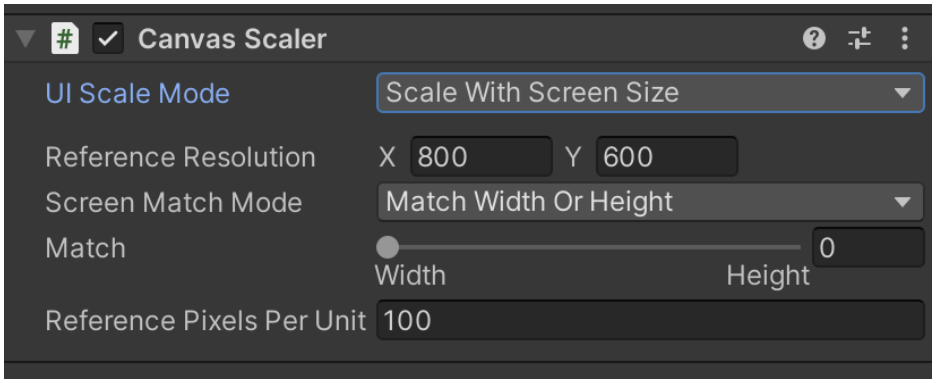


Next, we add a Canvas component to it. Press the “Add Component” button in the inspector and add a Canvas component.

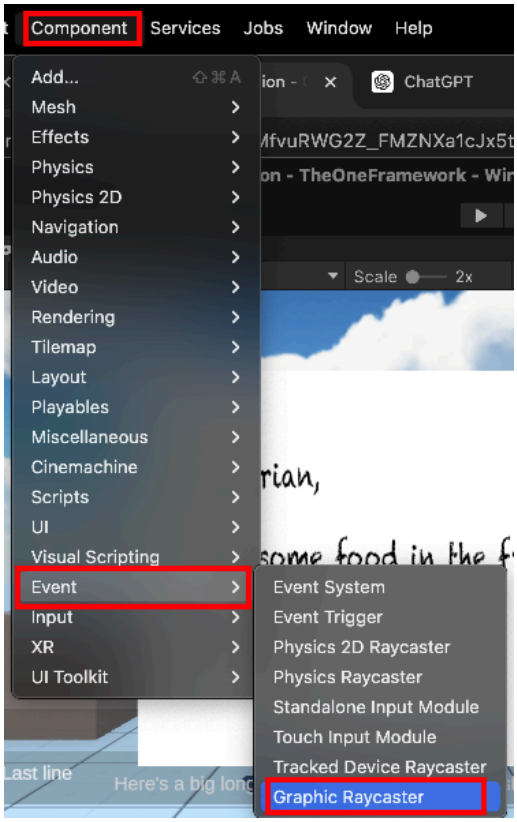




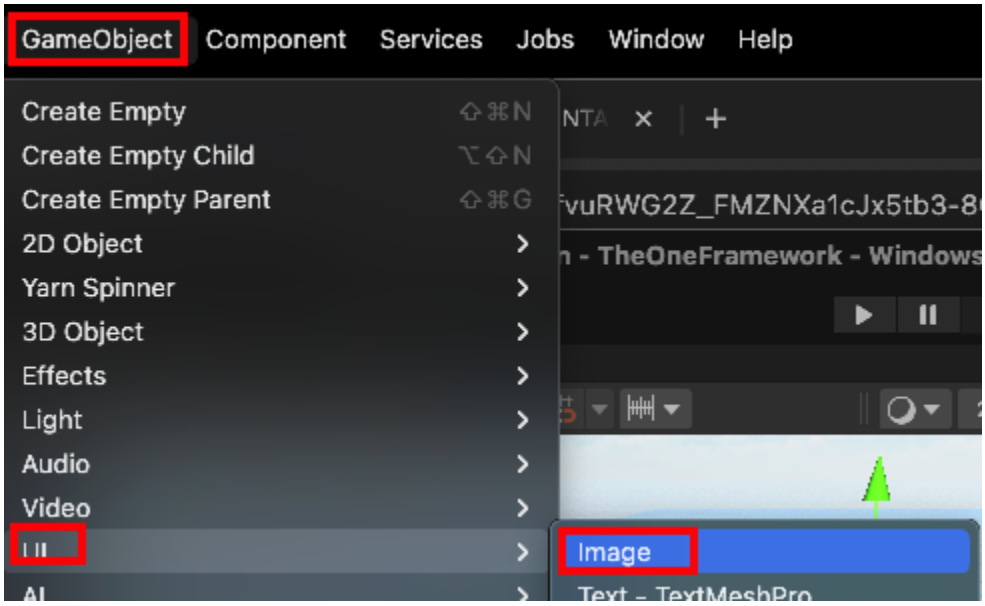
Change the Canvas Render Mode to “Screen Space - Overlay”. Optionally, check the “Vertex Color Always in Gamma Color Space” option to prevent the warning.
Then next, add a Canvas Scaler component the same way. Click the “Add Component” button again and add a Canvas Scaler.



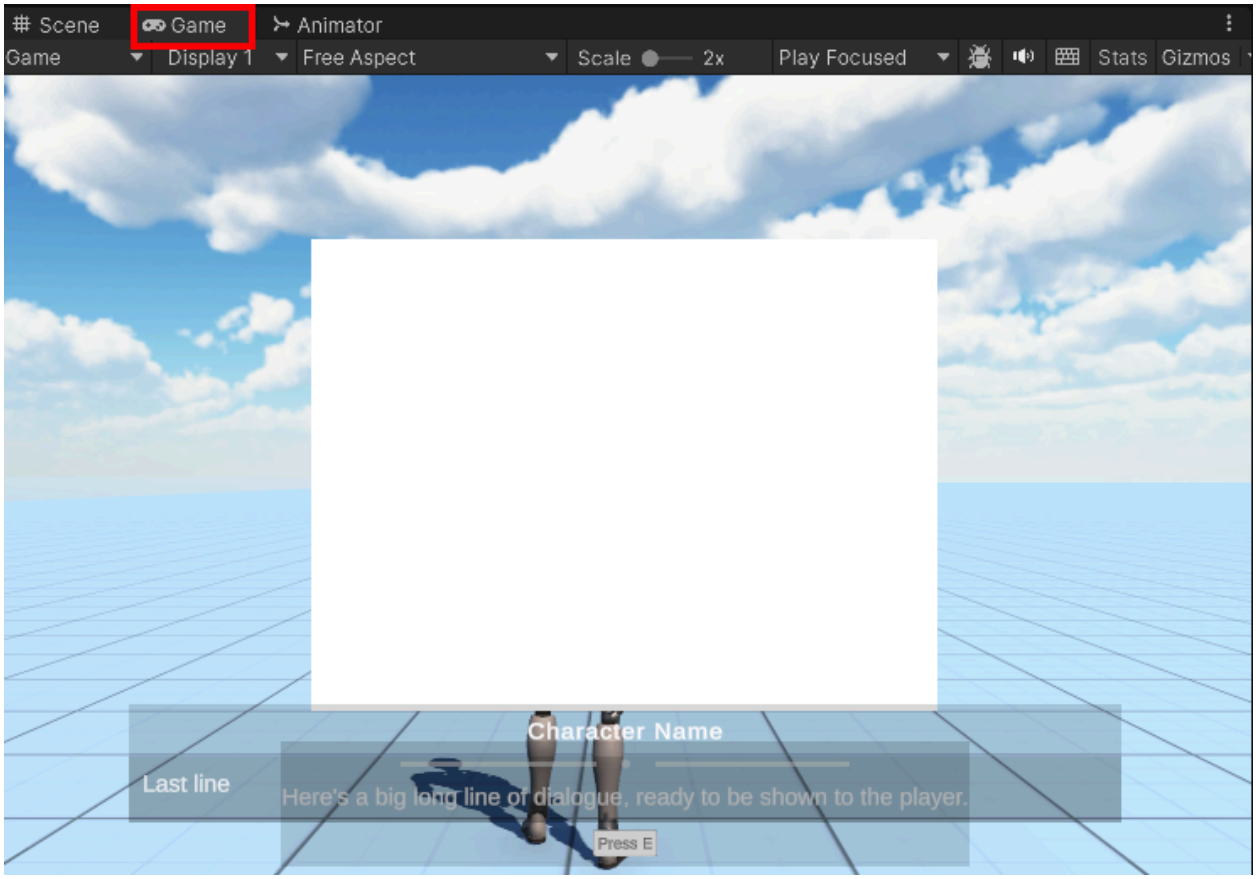
Change the UI Scale Mode setting to “Scale With Screen Size”
Finally, add the “Graphic Raycaster” Component.



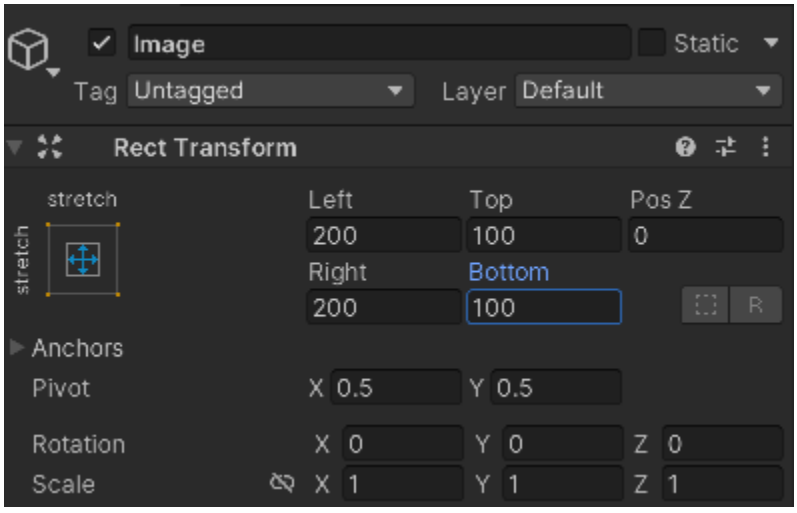
Now we add a UI Image. For this, go to GameObject->UI->Image.



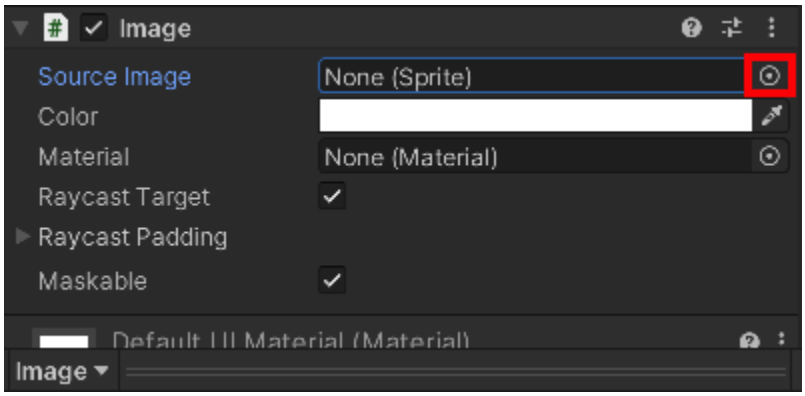
Use the tab above the scene view to switch to Game view.



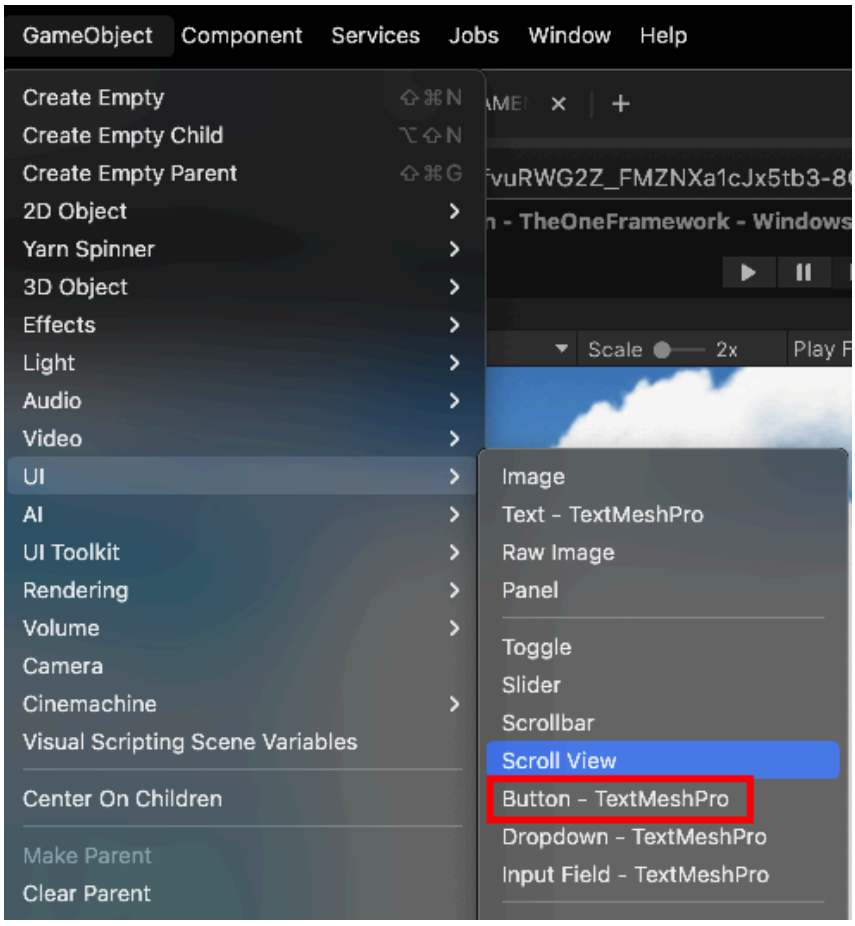
This will allow you to preview the image on screen.
You can adjust the scaling and anchoring in the inspector.



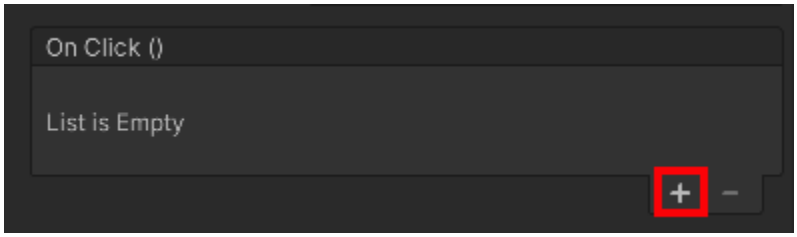
In the inspector, locate the Image component and press the small round icon on the right next to Source Image to select a new image.



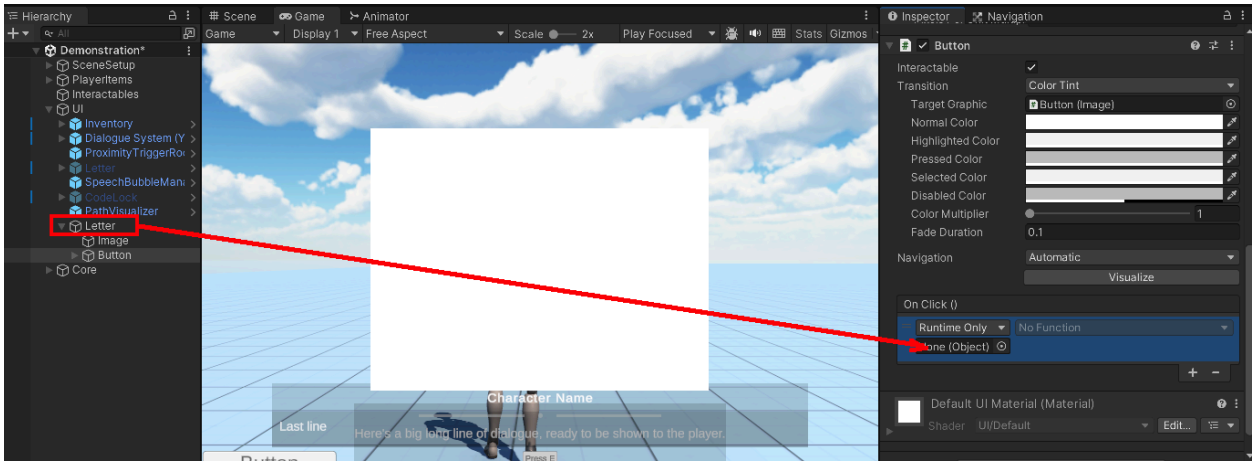
Now, let's add a button to the screen. Press GameObject->UI->Button - TextMeshPro. We will set this button up so that it will close the UI once it is on screen.



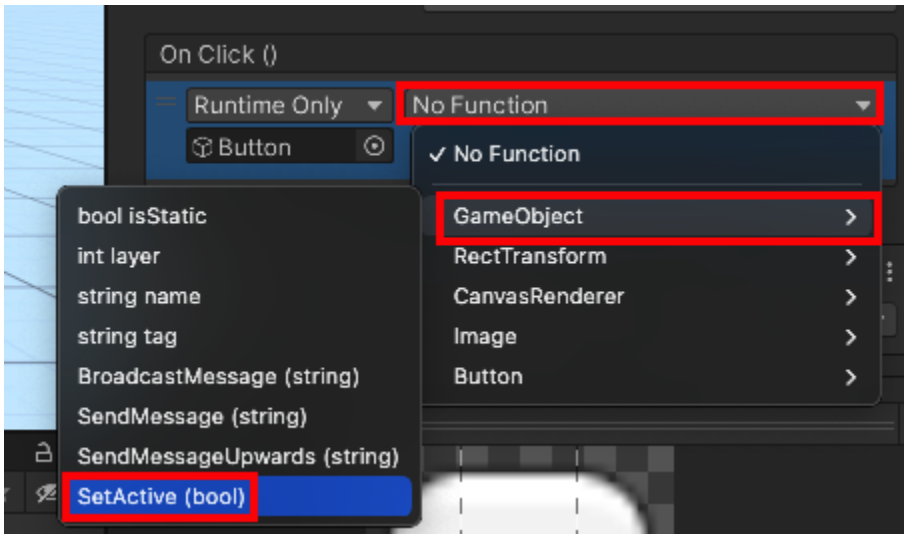
Select the newly created button in the hierarchy. In the Inspector window, locate the Button component. A bit down there is the OnClick property. Press the plus sign on the right to add an Unity Event listener.



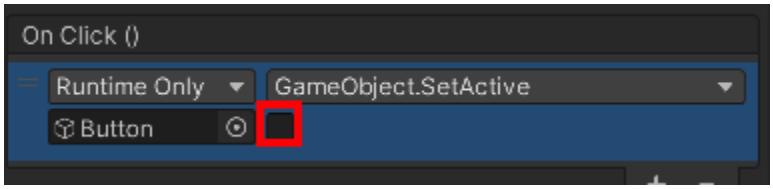
Next, drag the “Letter” GameObject from the hierarchy window onto the event listeners Object property.



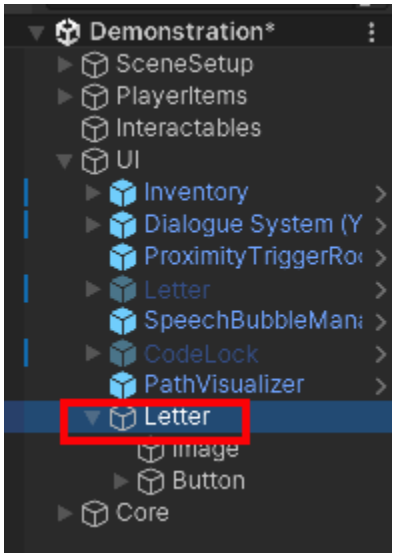
Next, in the “Function” dropdown, select GameObject->SetActive (bool).



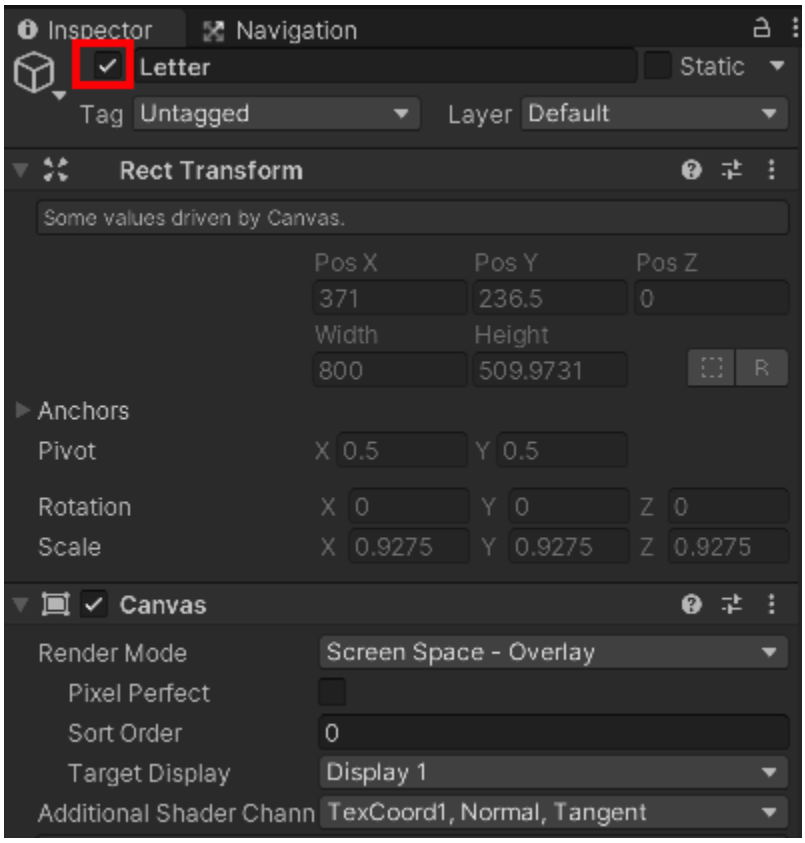
Leave the checkmark unchecked.



Select “Letter” in the hierarchy window.

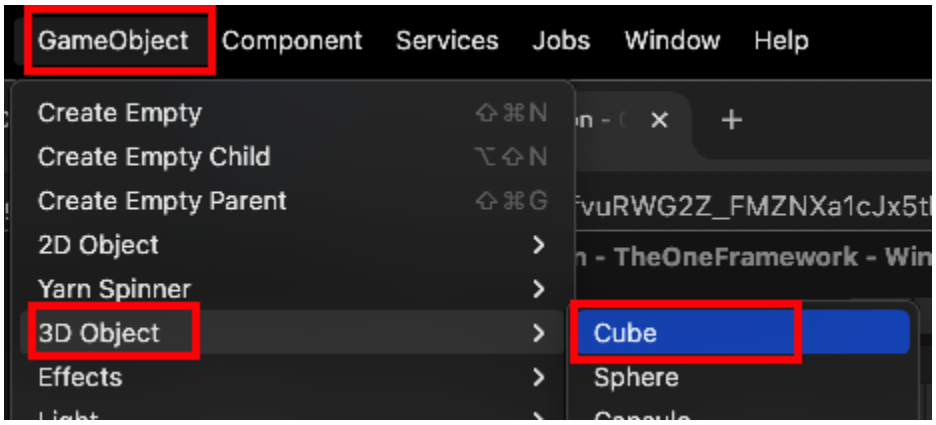


Go to Inspector and uncheck the box that is on the left of the object's name at the top of the window.

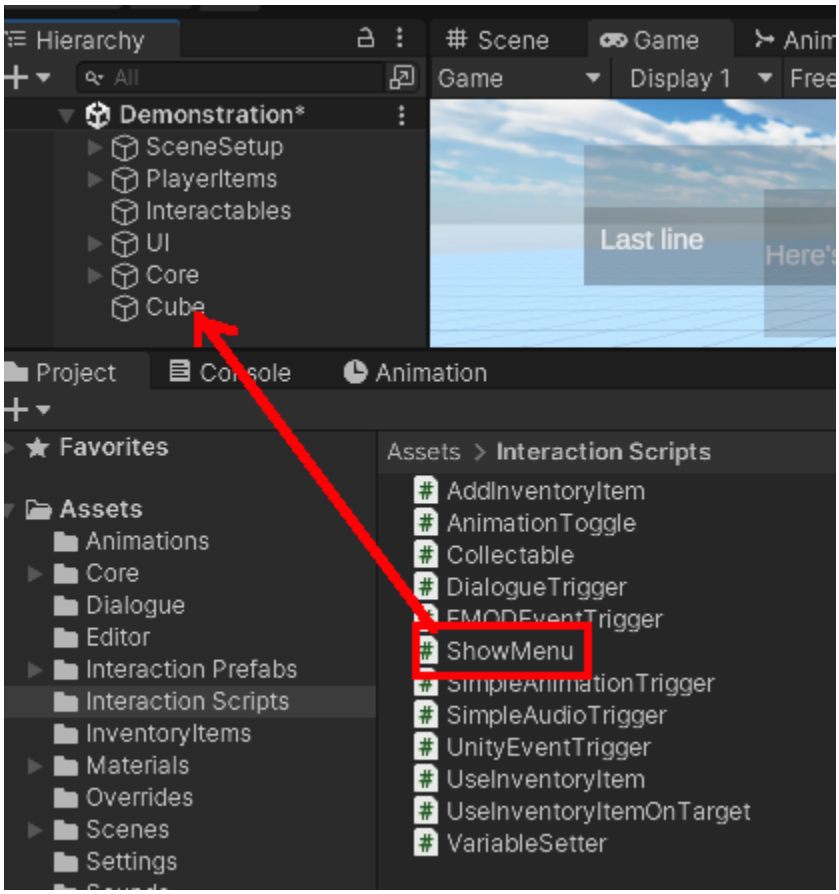


Now we made a UI node that has a close button and that is hidden at the start of the game.

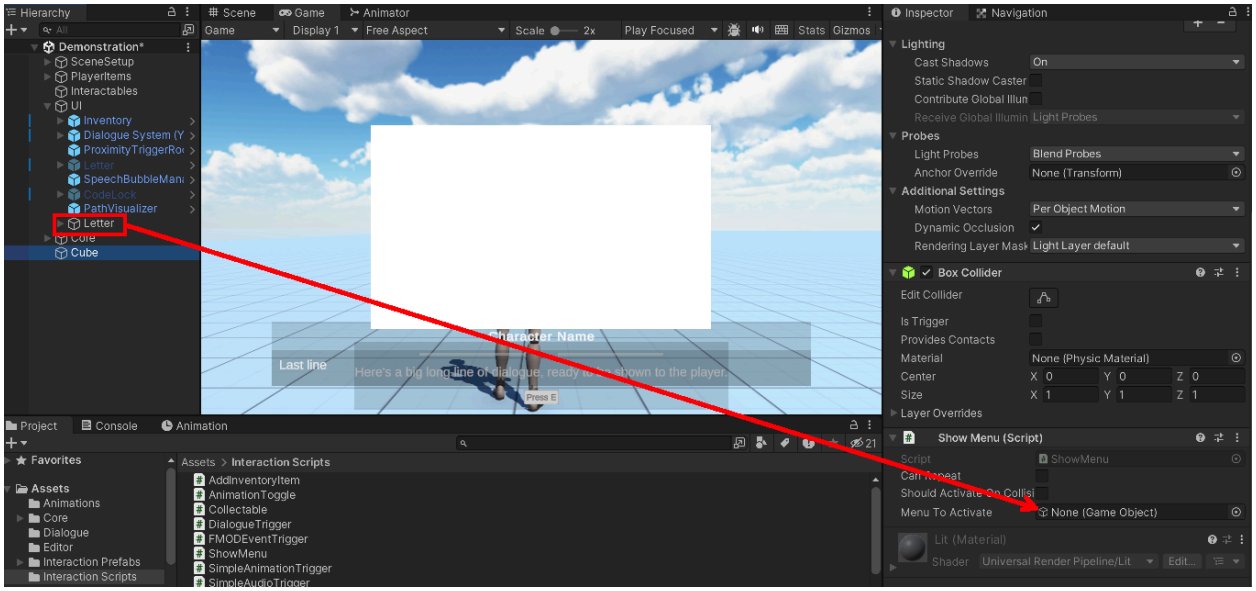
Let’s now set up a GameObject that will activate the UI node.
First, create a cube. Go the GameObject->3D Object->Cube



In the “Assets/Interaction Scripts” folder, locate the “ShowMenu” script and drag it onto the cube in the hierarchy.

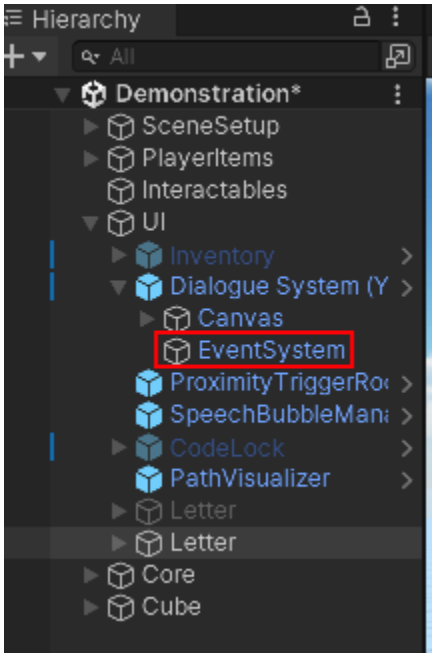


In the Inspector window, locate the ShowMenu component. Drag the Letter GameObject from the hierarchy onto the “Menu to activate” property of the component.



Finally, open the “Assets/Interaction Prefabs” folder and drag a ProximityTrigger onto the cube.

As a note: ensure that EventSystem is in the hierarchy and it is enabled. It is needed for buttons to interact.



If it is not there, you can add one by going to GameObject->UI->EventSystem.

Updating a variable in the HUD

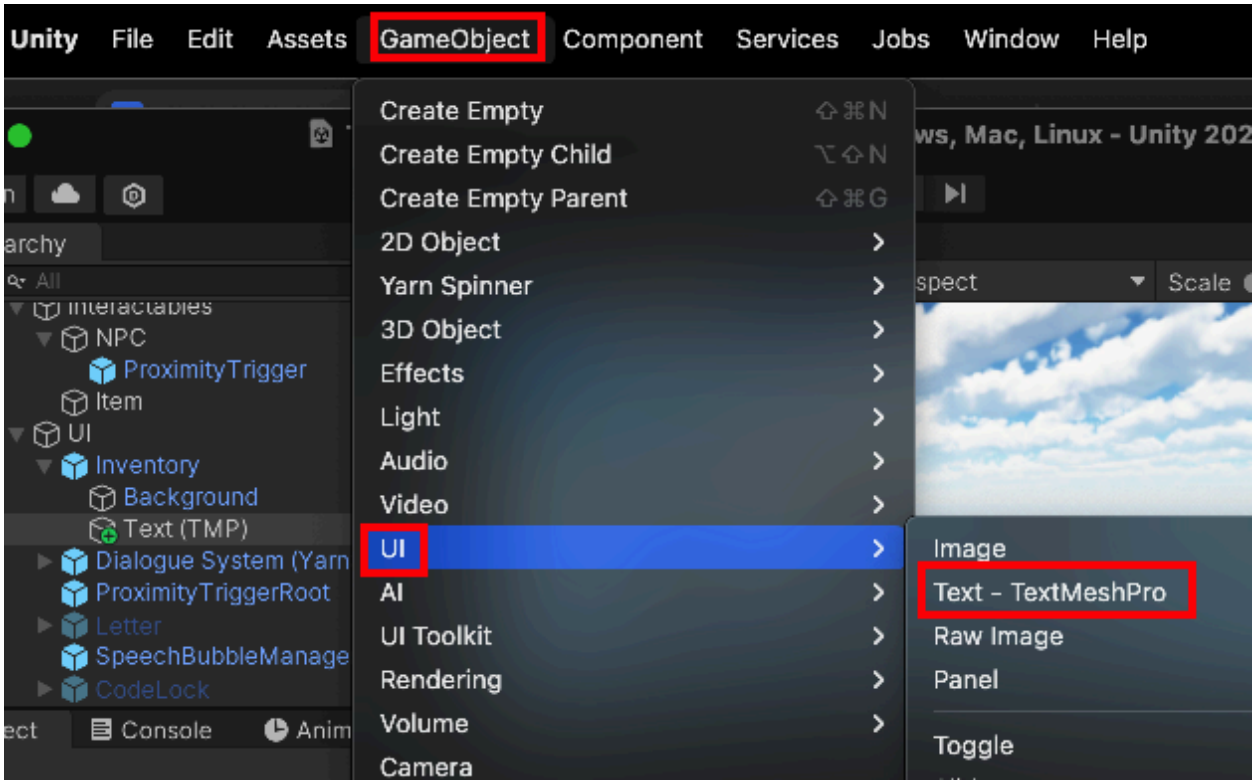
You can display a YarnSpinner variable in the HUD. This can be useful to show score or money. Variables can hold a number value or a string value. A string is a text between quotes.

In this example, we will make a variable in YarnSpinner, give it a value and display this value in the HUD. This is an edited excerpt from the *"Making a fetch quest"* tutorial above.

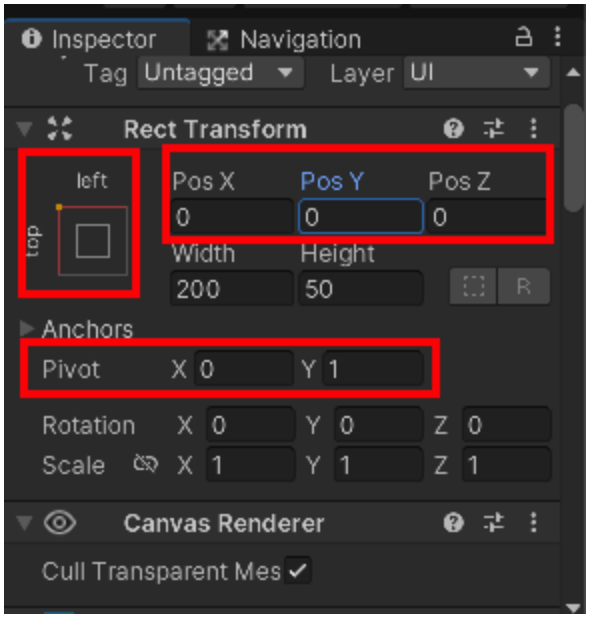
First, locate the script in the "Assets/Dialogue" folder and open it.
In the first node of the project, we'll add a variable called $score.

```
title: Start
---
<<declare $score = 0>>
===
```

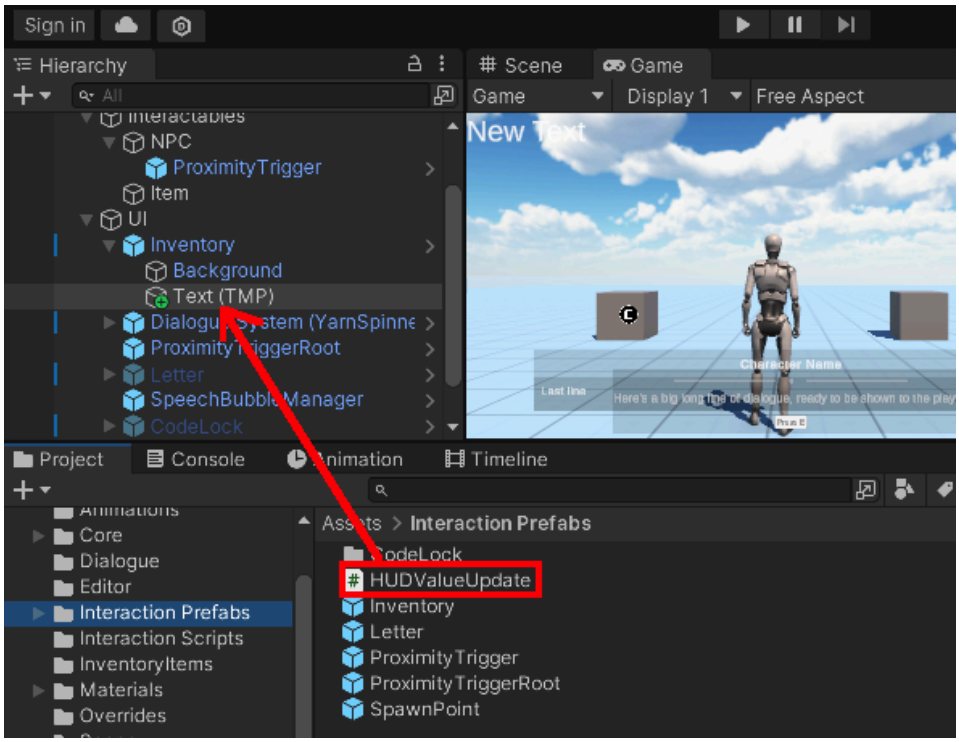
The value for the variable is 0. We can change it later on.
Add a TextMeshPro text object to the scene. This will be used to display the $quest variable.



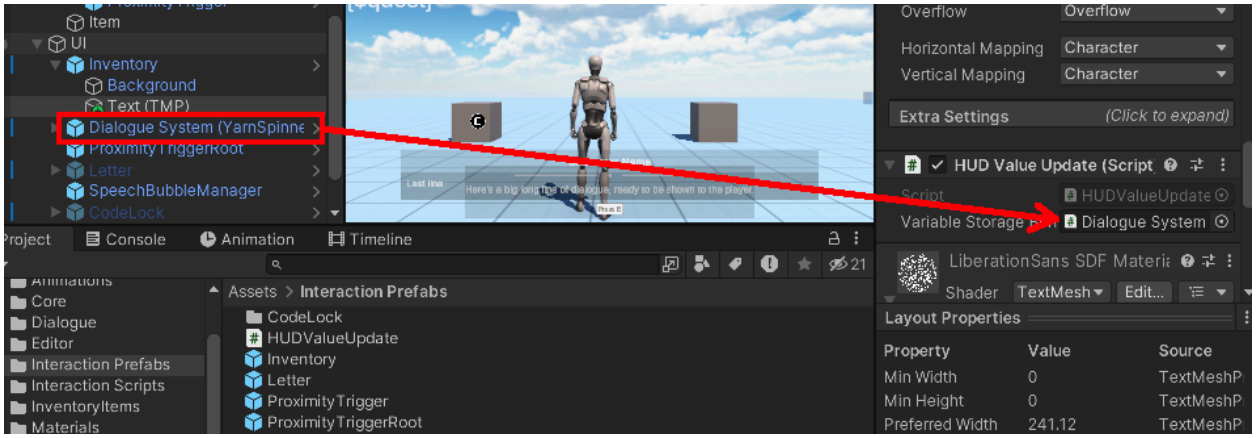
For this, go to the Unity's menu bar, then press GameObject->UI->Text - TextMeshPro.
In the inspector, change the origin of the object to 0, 1. Change the Anchor Preset to the top-left setup. Finally, change the position of the object to (0, 0). These settings are highlighted in the screenshot.



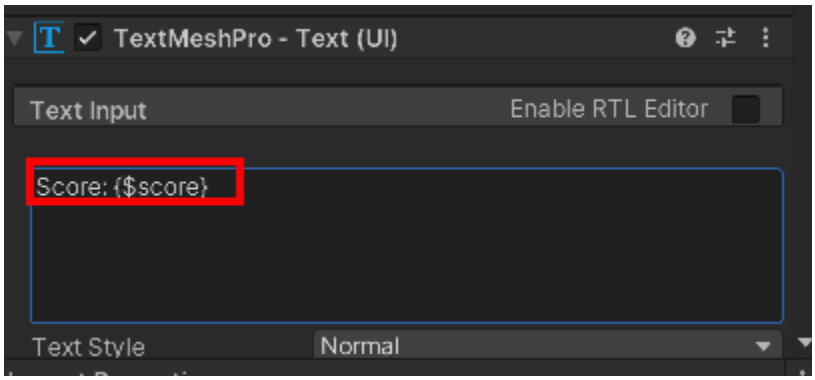
Drag the HUDUpdateValue script from the folder “Assets/Interaction Prefabs” onto the newly created text object.



Drag the Dialogue System node from the UI node in the hierarchy onto the HUDUpdateValue component’s field “variable storage behavior”.



In the inspector, change the text field of the TextMeshPro Text component into: Score: {$score}



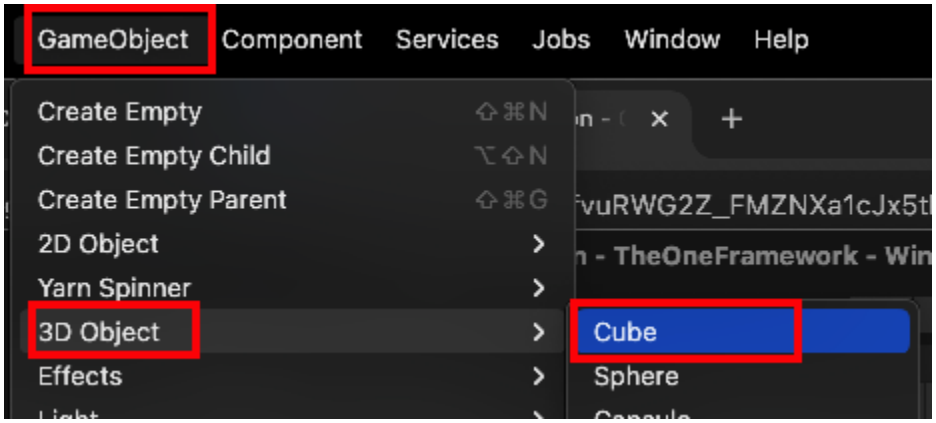
The part that reads {$score} will be replaced with the content of the variable from the YarnSpinner script. This will show the score onto the screen for the player.

For more information about YarnSpinner variables, see this article:
https://docs.yarnspinner.dev/beginners-guide/syntax-basics#logic-and-variables

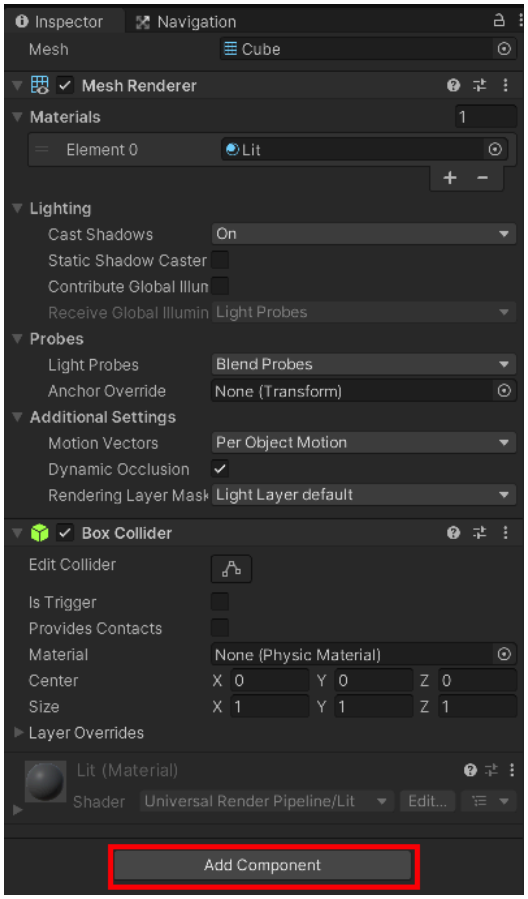
Starting a UnityEvent (Particle Effect)

The UnityEventTrigger component is a PlayerActivable that can trigger a Unity event. This gives low-level access to Unity's functionality. You can use it to trigger a wide range of things. You can enable/disable items, change their color, start/stop animations, sounds, music. Most of the functionality of the other PlayerActivatables can be achieved with only this script, and more. In this demonstration, we will make a particle effect and activate it.

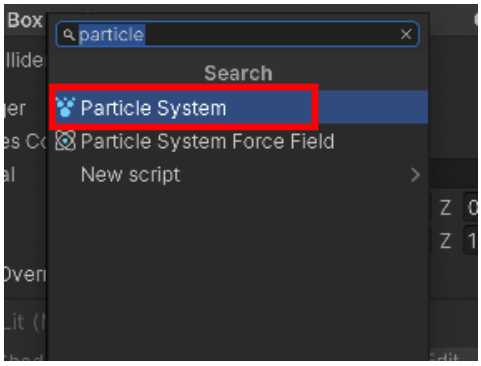
Go to GameObject->3D Object->Cube to add a cube to your scene. Make sure to select it.



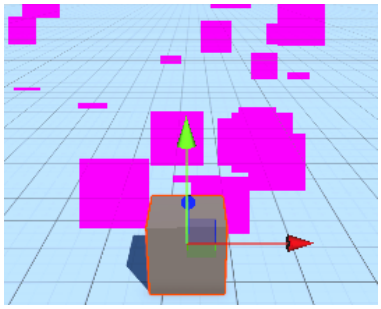
In the Inspector, press "Add Component"



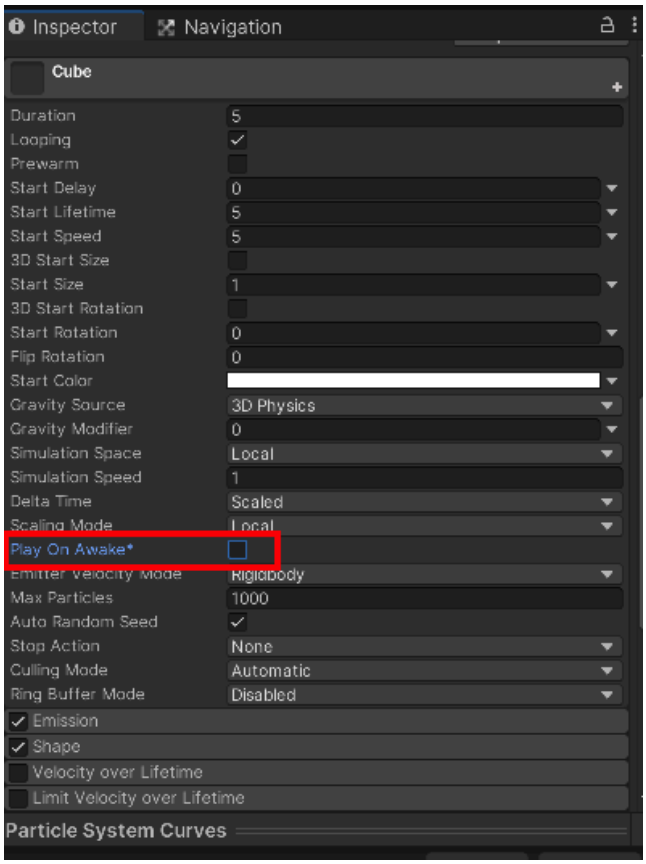
In the box that appears, search for “Particle System”.



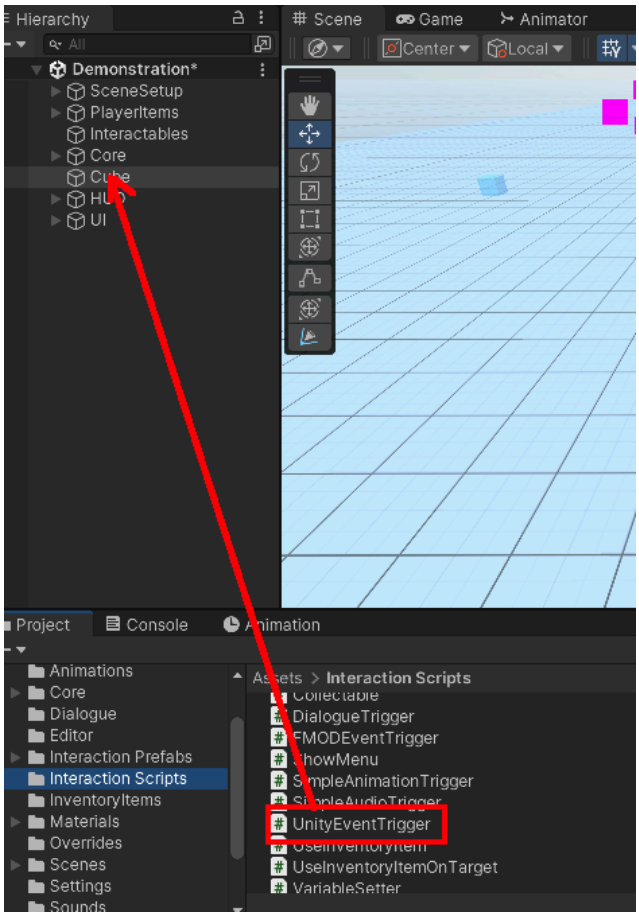
You can change the properties of the particle system. By default, the material is not compatible with the current pipeline, so it will look purple.



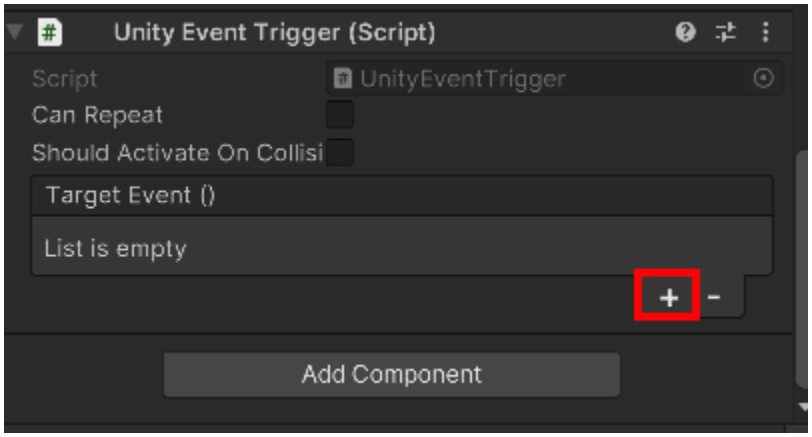
In the Inspector for the Particle System, find the property “Play on Awake” and disable it.



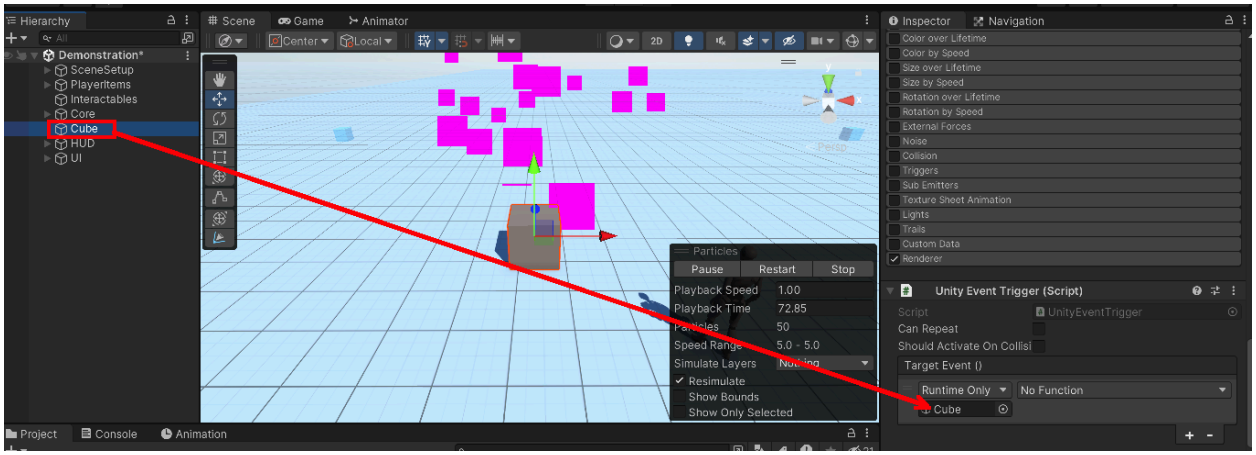
Locate the UnityEventTrigger script in the folder “Assets/Interaction Scripts” and drag it onto the cube.



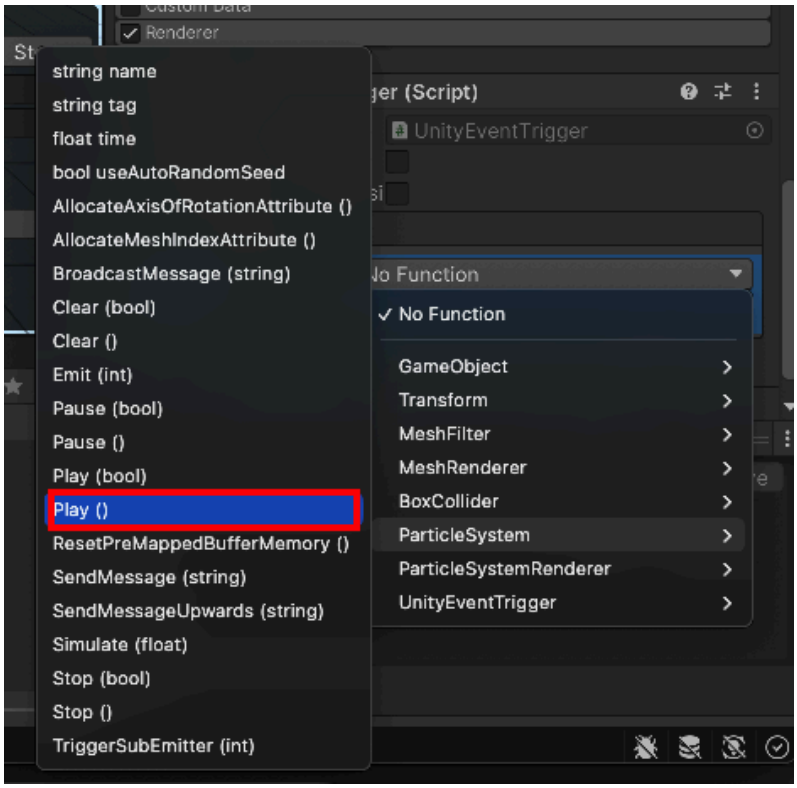
In the Inspector for the UnityEventTrigger, locate the “Target Event()” property and add a slot using the “+” button on the bottom right of it.



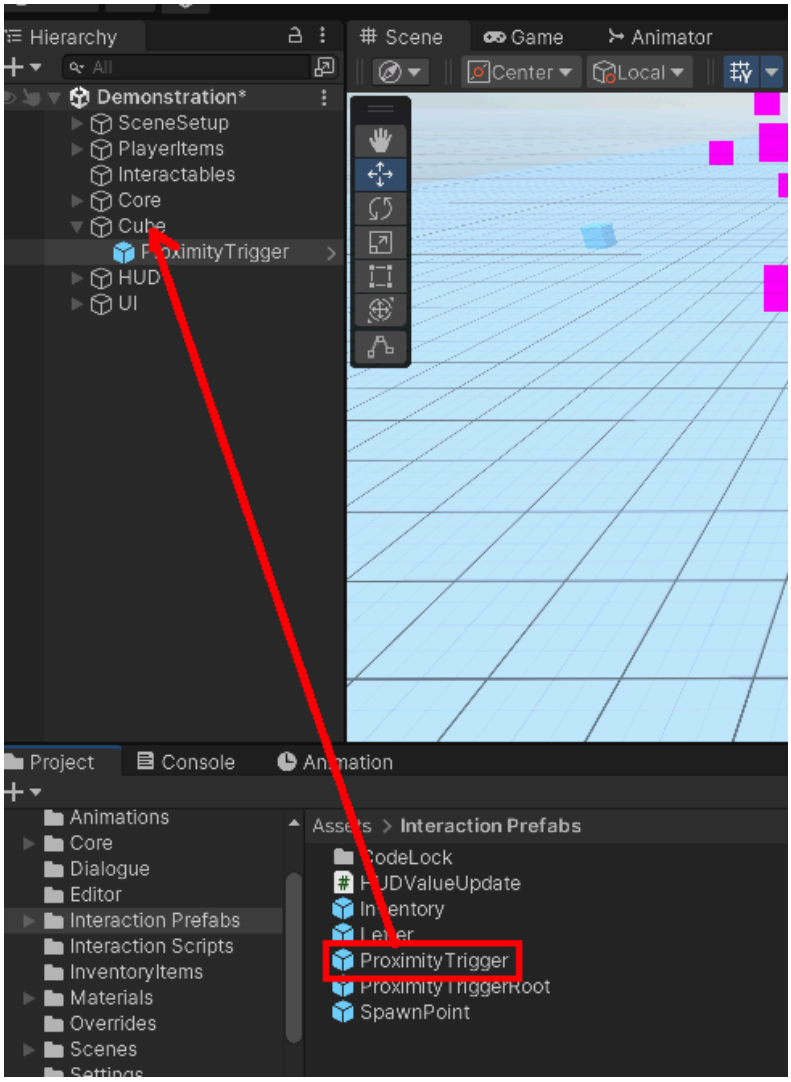
Drag the cube onto the target property (that currently says “None”)



In the drop-down box on the right (the one that says “No Function”), select the Particle System script and select the “Play” method.

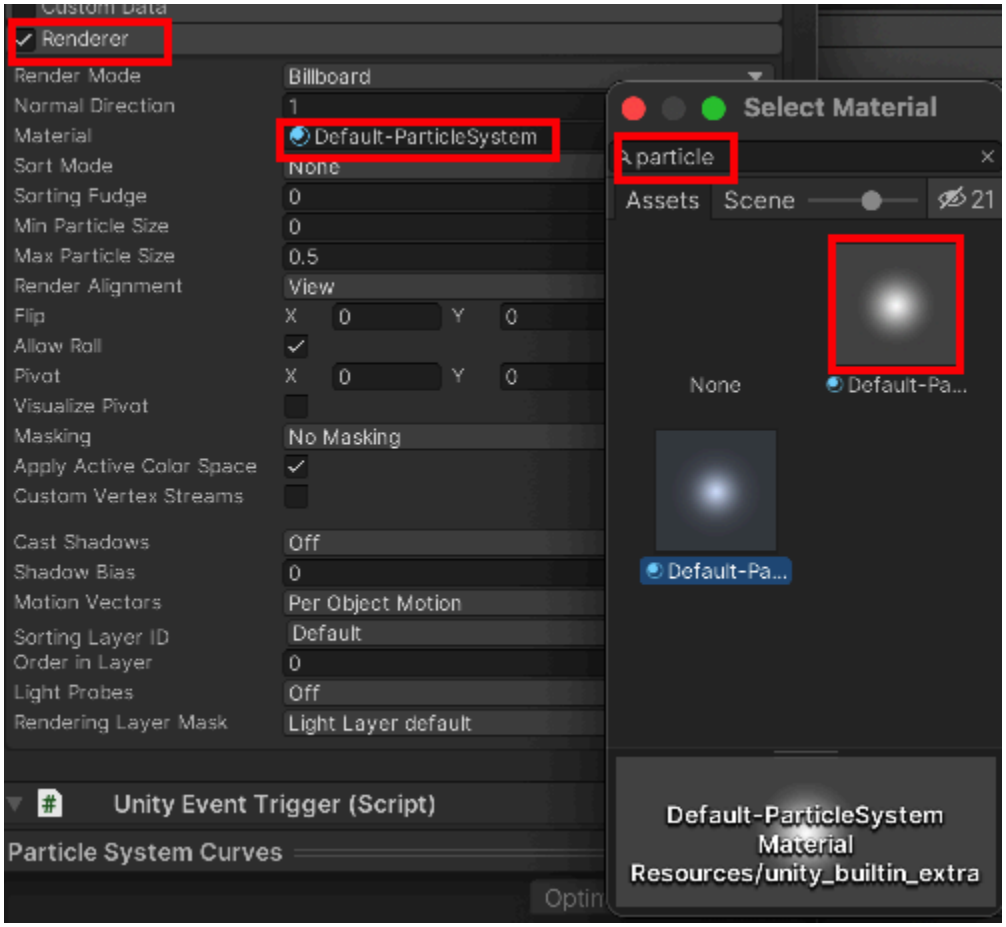


Locate the ProximityTrigger in the “Assets/Interaction Prefabs” folder and drag it onto the Cube.

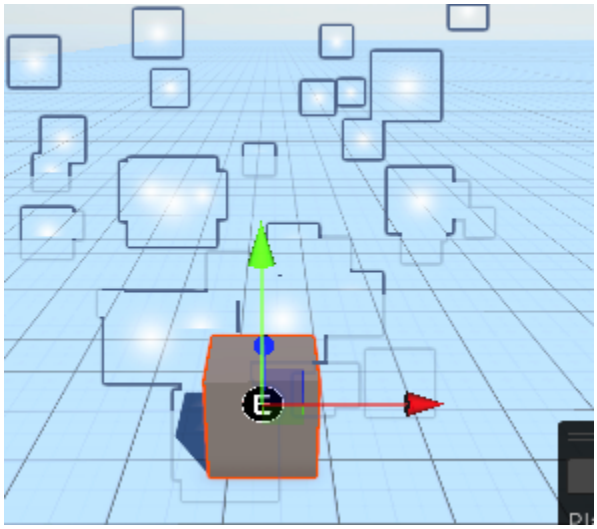


Now when you activate the cube, the particle effect should show.

To change the material, you could look at the “Renderer” property of the ParticleSystem.



Click on Renderer in the inspector, edit the Material property and search for a material that contains the name “Particle”.

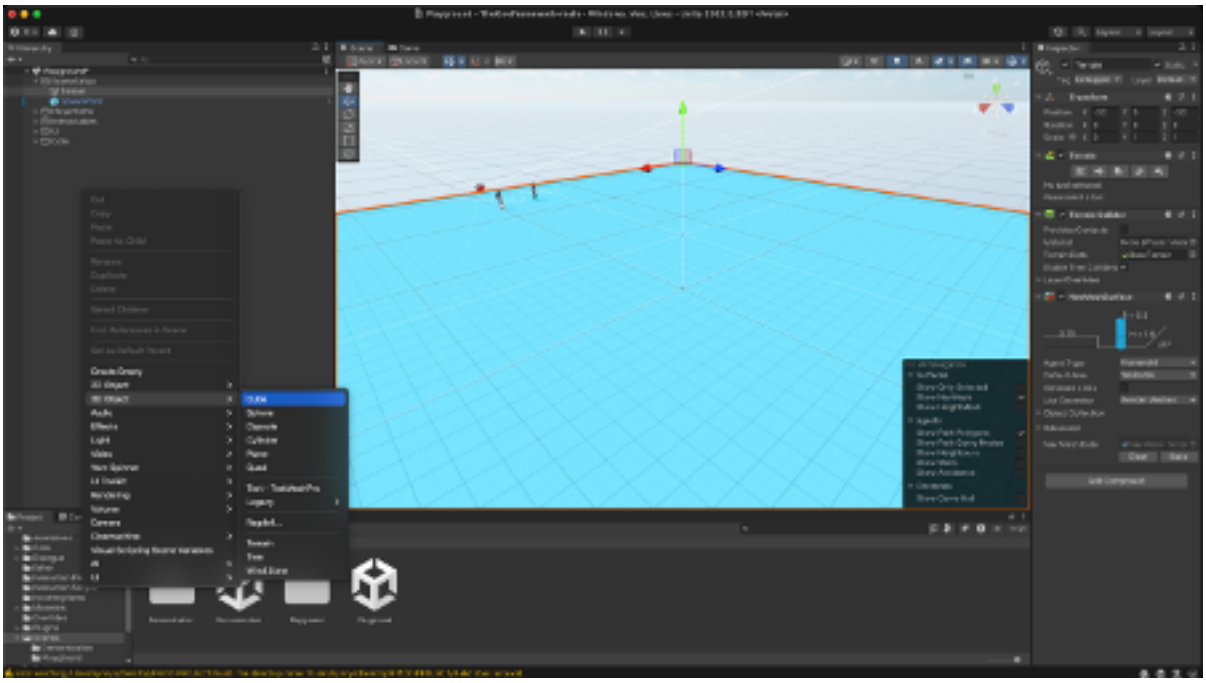


For more information about editing particle effects, please refer to the Unity documentation: https://learn.unity.com/tutorial/introduction-to-particle-systems#

Setting up interaction with FMOD event

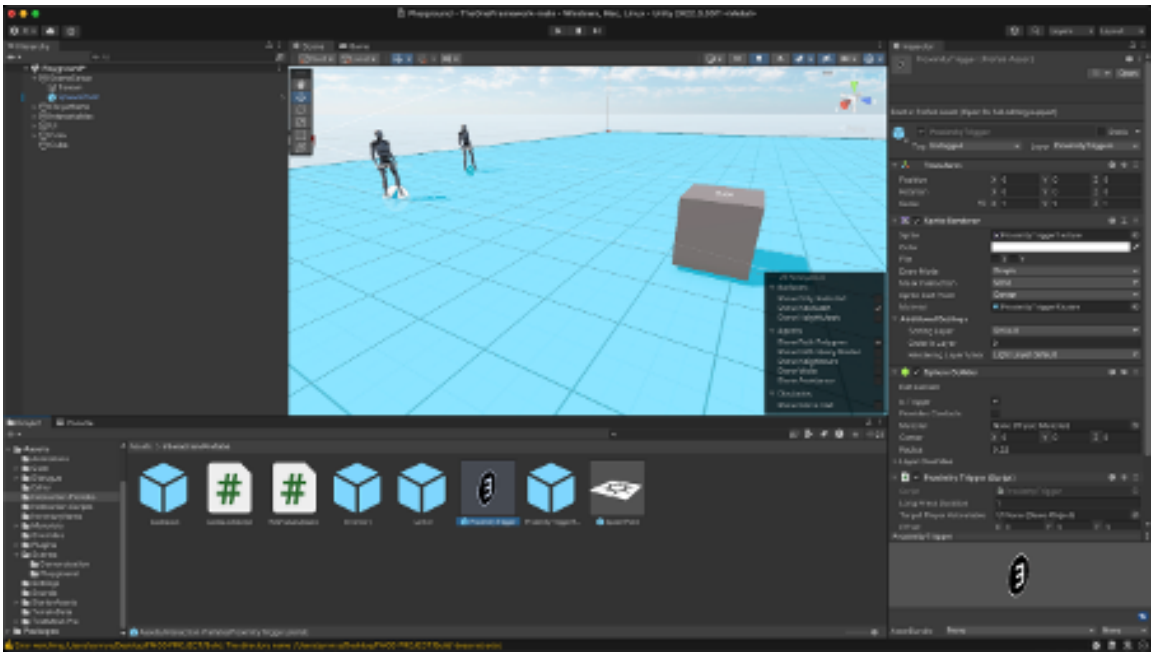
Before you start, please follow the FMOD Integration Setup for Unity Project at the end of this document.

1. Create a Cube:
    - In Unity, create a new cube by going to GameObject > 3D Object > Cube.

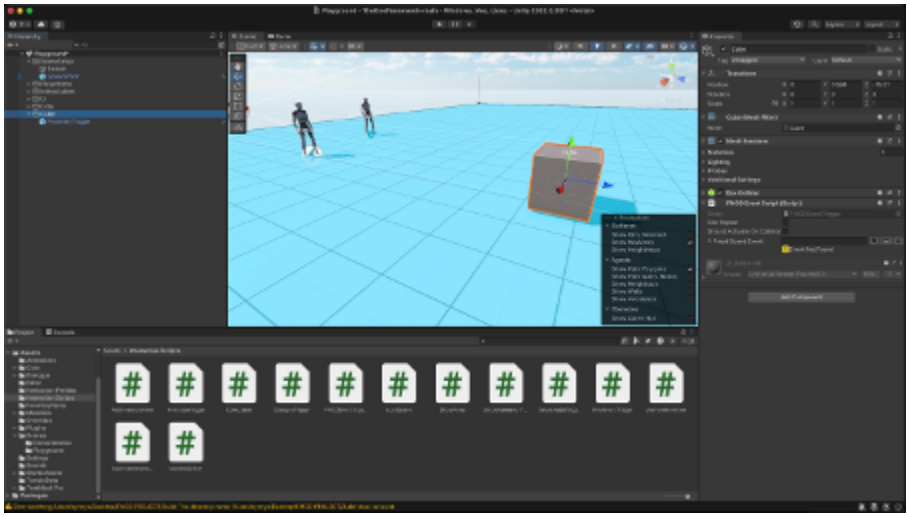


2. Add Proximity Trigger:

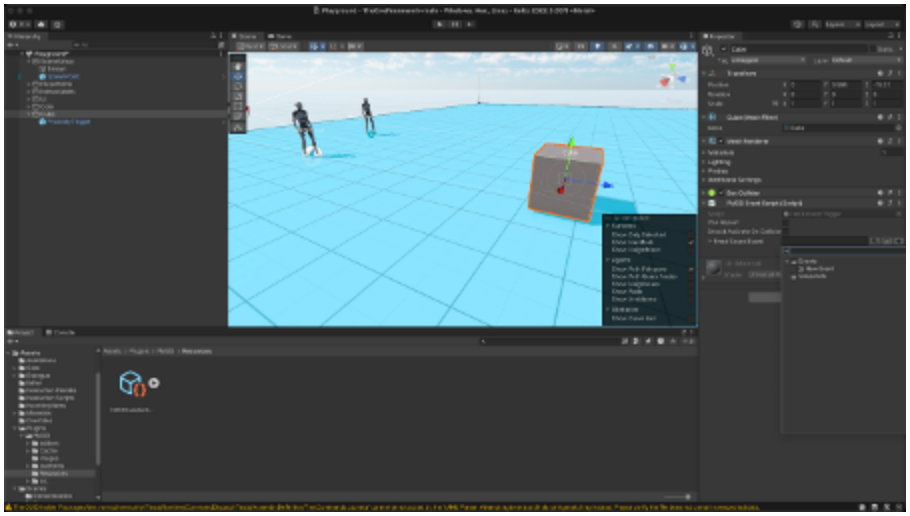
- Drag the Proximity Trigger prefab from the interaction prefabs folder onto the cube.



3. Attach FMODEventTrigger Script:
    - Drag the FMODEventTrigger script from the interaction scripts folder onto the cube.



4. Configure FMODEventTrigger:
    - Select the cube and go to the Inspector window.
    - In the FMODEventTrigger component, browse for the event you created in FMOD Studio and select it.

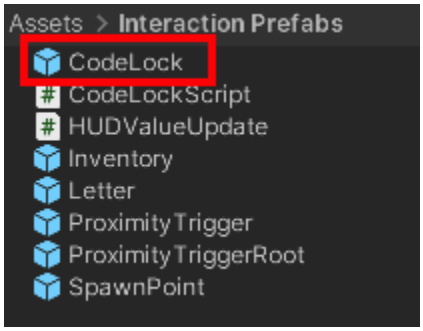


5. Test the Setup:
    - Play the game in Unity.
    - Walk up to the cube until an "E" displays on the screen.
    - Hold the "E" key for more than 2 seconds, and your sound should start playing.

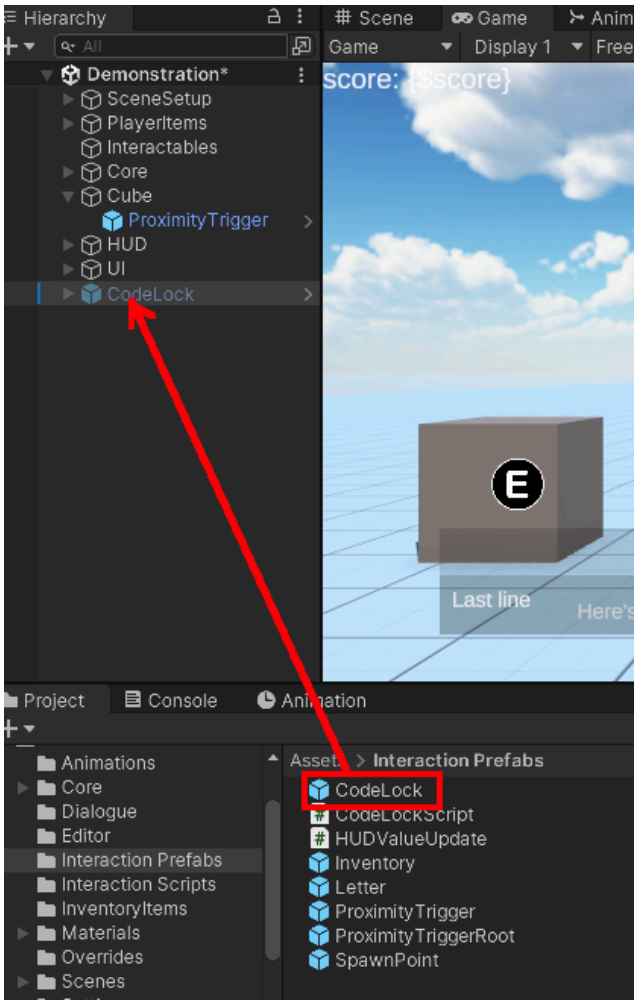
Following these steps will ensure that your FMOD integration is complete, and you'll be able to trigger FMOD events through interactions in your Unity project.

Showing a code lock and responding to it in a dialogue

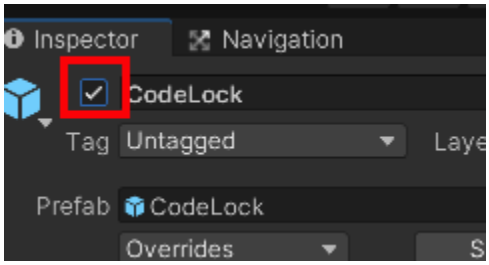
Locate the CodeLock prefab under “Assets/Interaction Prefabs”



Drag it into the hierarchy.

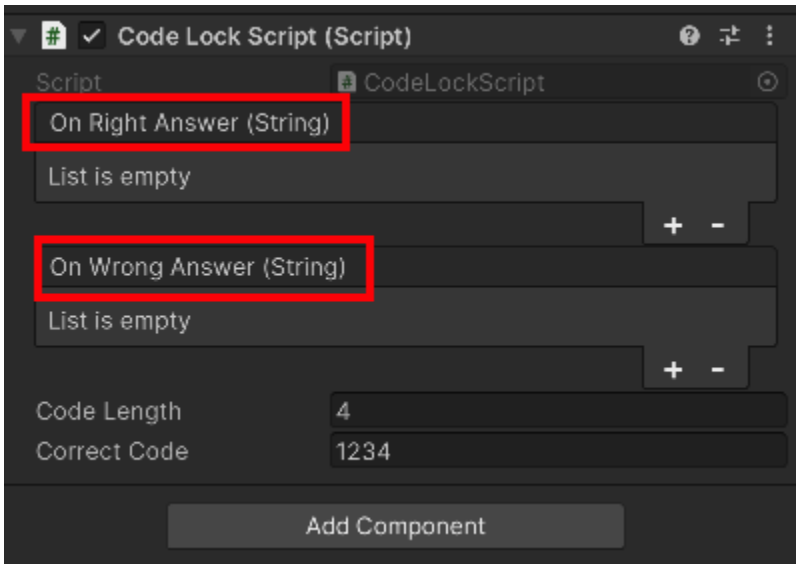


In the Inspector, activate the GameObject by checking the checkbox next to the object's name.

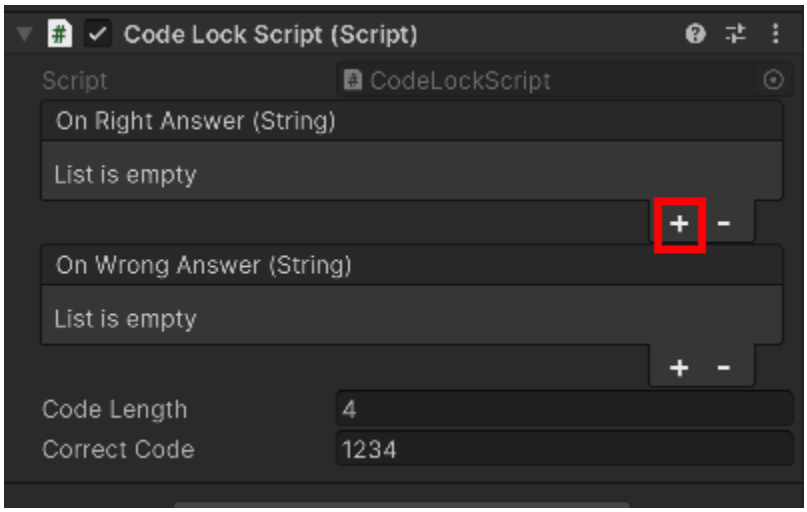


You can do this from a PlayerActivatableby using the ShowMenu script *(see "Showing a UI screen")*.

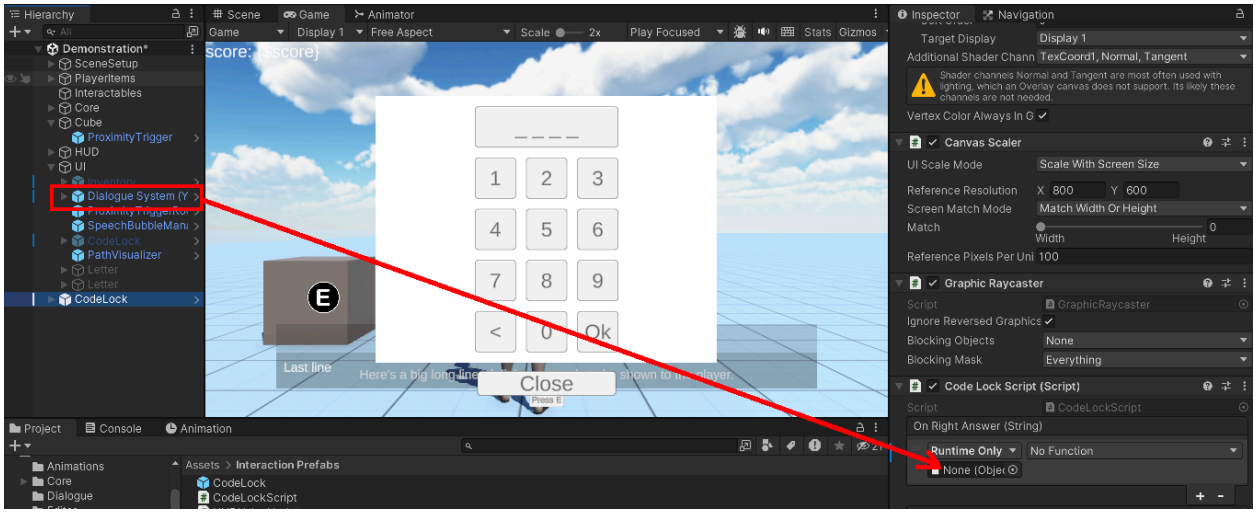
Scroll down in the Inspector to find the "OnRightAnswer" and "OnWrongAnswer" properties.



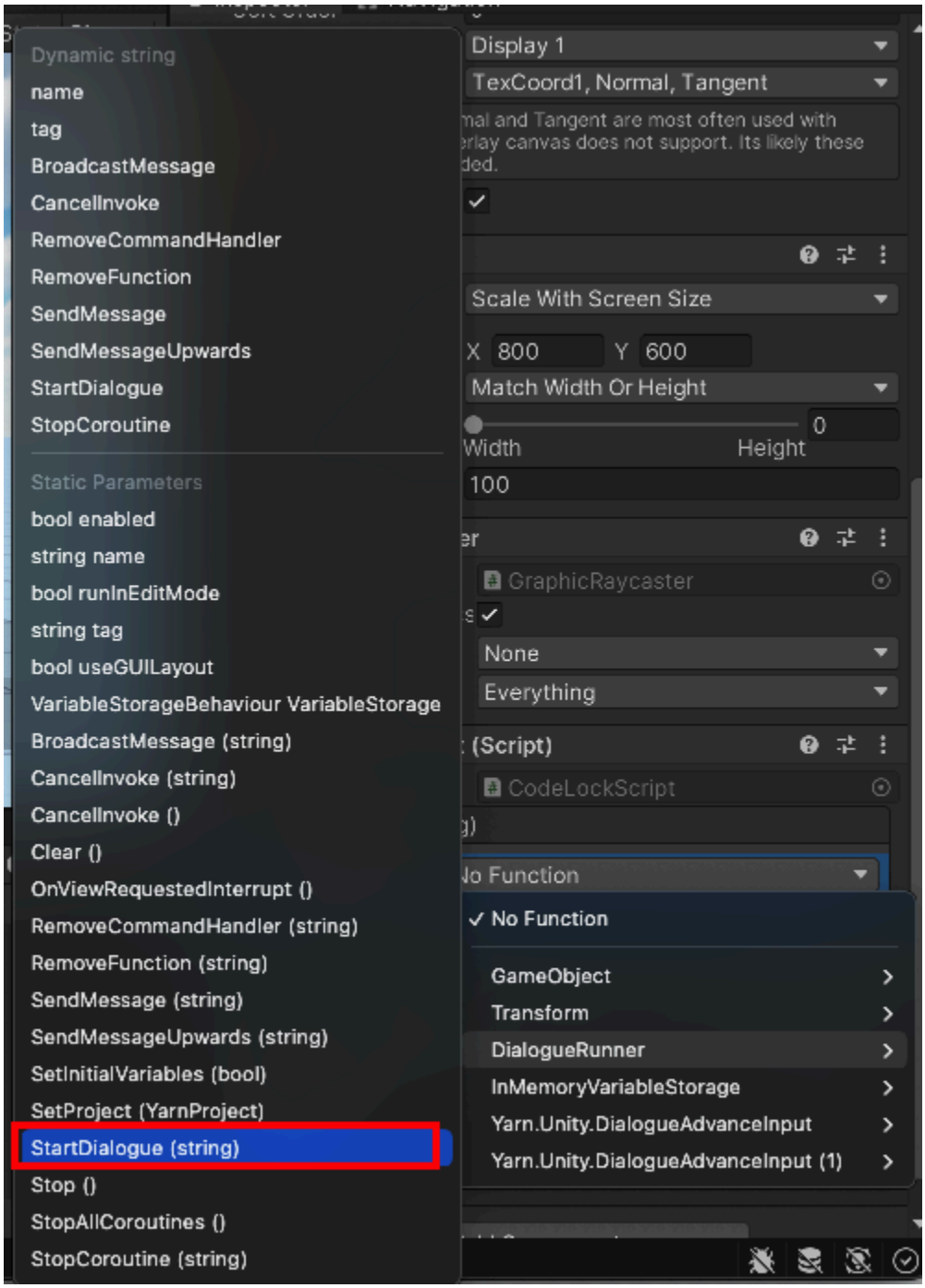
Use the + sign to add an event to these. For this example, we will only add one for the OnRightAnswer property.



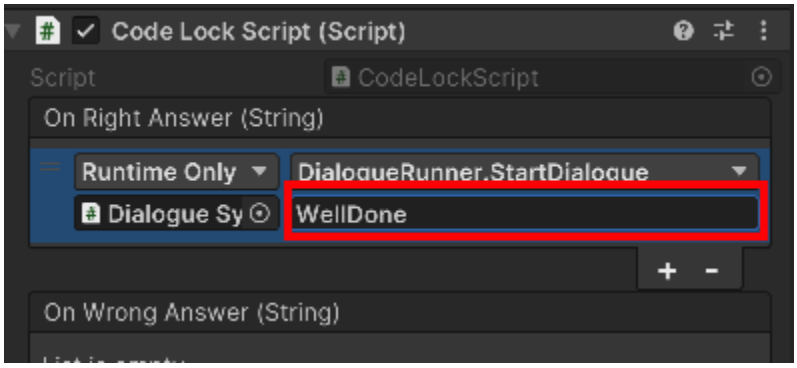
Drag the “DialogueRunner (YarnSpinner)” gameobject from the hierarchy onto the target field of the event. (The one the currently reads “None (Object)”)



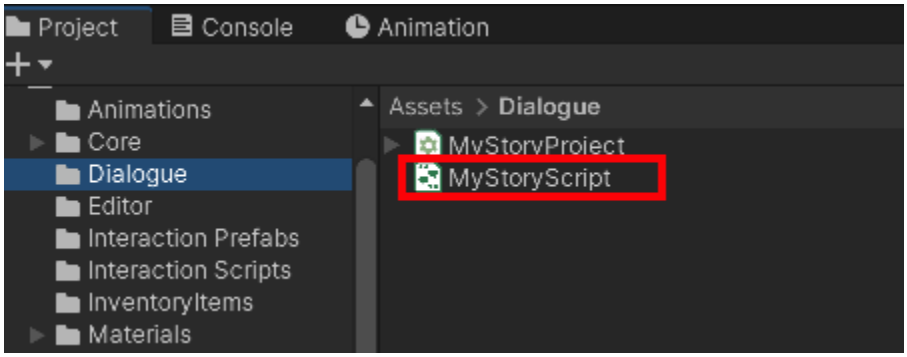
In the dropdown box on the right (The one that currently reads “No Function”), locate the YarnSpinner script and find the StartDialogue function.



Select it and fill in the name of the YarnSpinner node you want to play on the right answer.



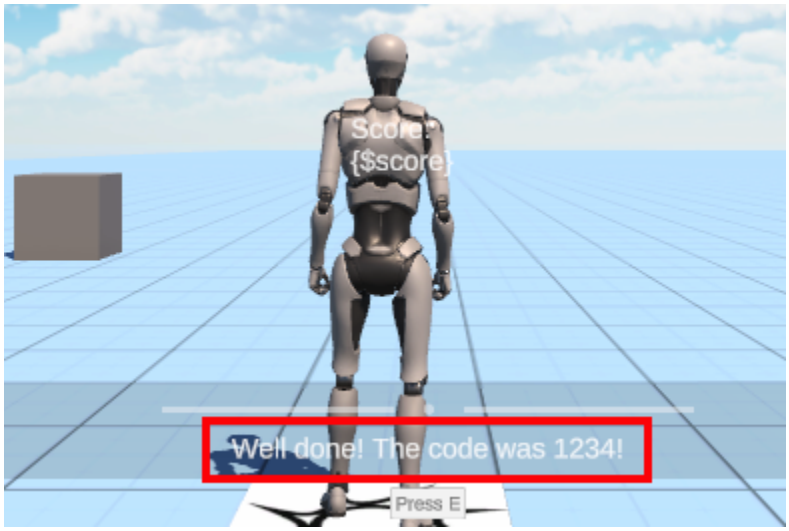
Open the YarnSpinner dialogue script from “Assets/Dialogue”



Write the “WellDone” node as follows. *(Make sure to save the script)*

```
0 references | Show in Graph View
title: WellDone
---
Well done! The code was 1234!
===
```

Now, when the player guesses 1234, the script will show the message.



Loading scenes

In this tutorial, we will guide you through adding a Cube to your scene and setting it up with a script that allows you to load a new scene, while keeping the Player, the Camera and the UI.

Step 1: Open the first scene

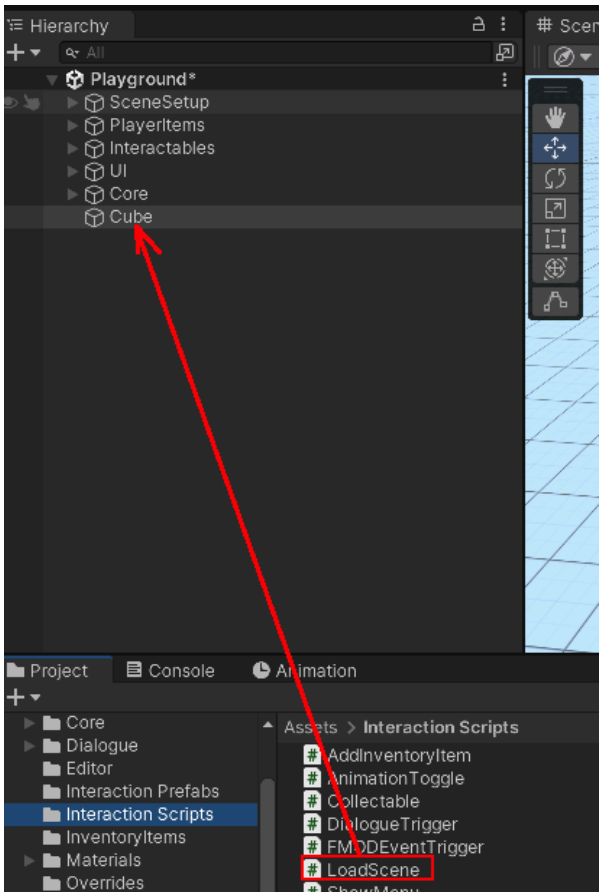
First, locate the scene "Playground" in the “Assets/Scenes” folder. Double-click on "Playground" to open it in the Unity Editor. This scene will serve as our starting point.

Step 2: Add a Cube to the Scene

Next, we will add a Cube to the scene. In Unity's menu bar, navigate to **GameObject > 3D Object > Cube**. This will add a Cube to your scene. You can position the Cube anywhere you like.

Step 3: Attach the LoadScene Script

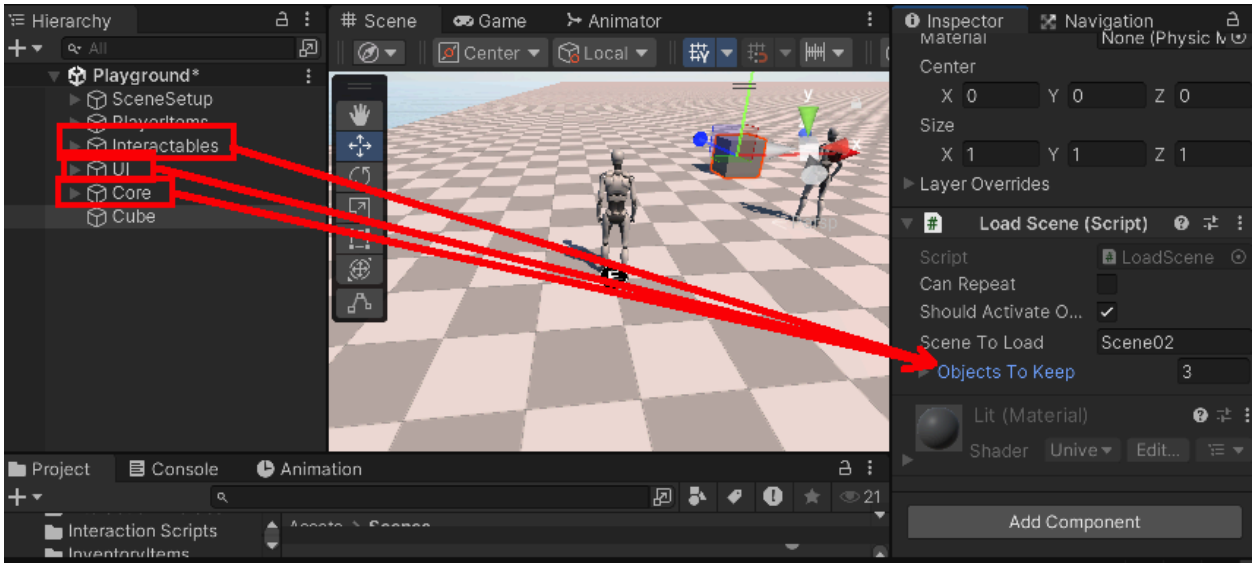
Now, we need to attach the `LoadScene` script to the Cube. You can find the script in the “Assets/Interaction Scripts” folder. Drag it onto the newly created cube.



Step 4: Add GameObjects to the Script's List

With the `LoadScene` script attached, we will now specify which GameObjects should be kept across scenes.

1. In the Hierarchy, locate the GameObjects "PlayerItems", "UI" and “Core”.
2. Drag these GameObjects into the `objectsToKeep` property that is on the `LoadScene` component.



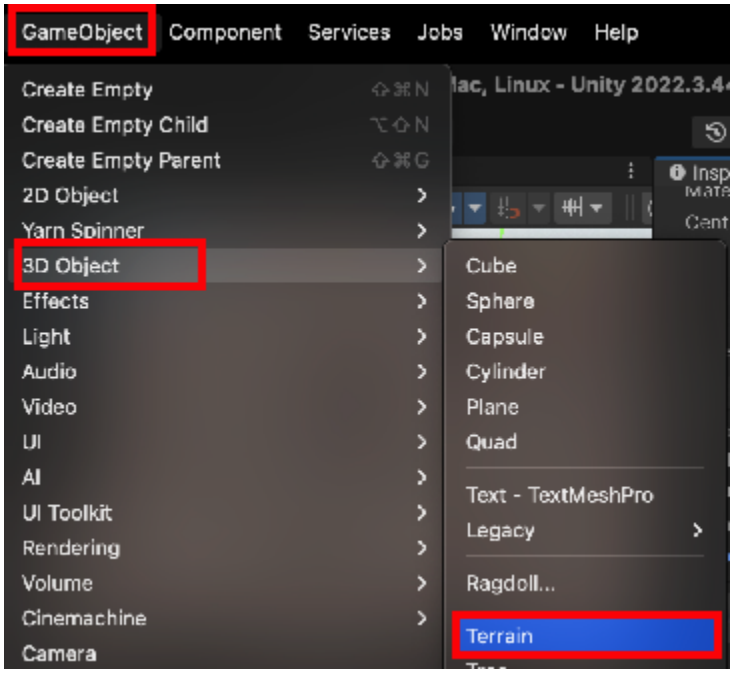
This will ensure that "PlayerItems", "UI" and “Core” will be kept when transitioning to the new scene. By default, Unity will remove all items from the scene when entering a new scene. However, in this case, we want to keep the Player, Camera and UI between scene changes.

Step 5: Create and Save a New Scene

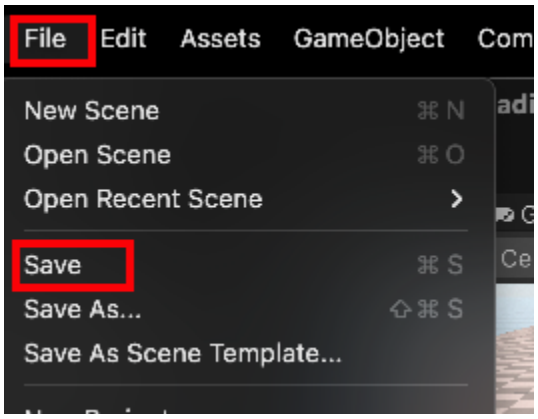
Next, let's create a new scene that we can load.
First save the current scene. Go to **File > Save** to save it.

1. Go to **File > New Scene** to create a new scene.
2. Save this new scene by navigating to **File > Save As...**.
3. Save the scene in the “Assets/Scenes” folder, giving it an appropriate name.

Now go to **GameObject > 3D Object > Terrain** to add a terrain to the scene.



Then, save the scene by choosing **File > Save**.



Step 6: Add the New Scene to the Build

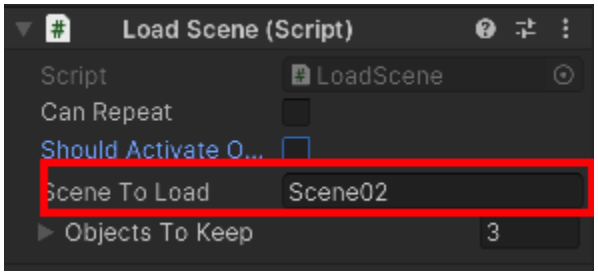
To make sure the new scene can be loaded, it must be added to the Build Settings.

1. Go to **File > Build Settings...**.
2. In the Build Settings window, click “Add Open Scenes” to add your newly created scene to the list of scenes included in the build.

Step 7: Configure the LoadScene Script

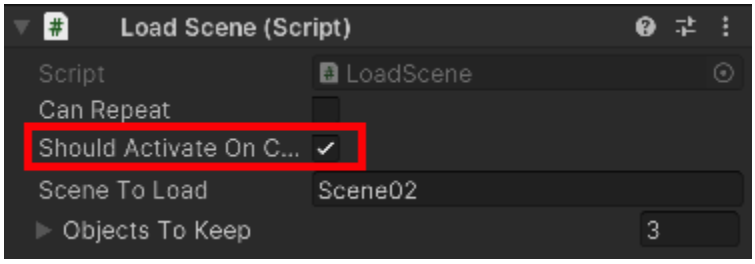
Now that you have created a new scene, we need to set up the Cube to load this scene when activated.

1. Go back to the "Playground" scene by double-clicking on it in the “Assets/Scenes” folder.
2. Select the Cube in the Hierarchy.
3. In the `LoadScene` component, find the `sceneToLoad` field and enter the name of the newly created scene.



In this example, I named the scene “Scene02”.

Finally, check the box that reads “Should activate on Collision”.



Your setup is now complete! When the Player runs into the Cube, the `LoadScene` script will load the specified scene while preserving the Player, the Camera and the UI across the scene transition.

Function guide

This section will show all commands and functions that are made available in YarnSpinner by the framework. They are sorted alphabetically.

activate

Syntax

<<activate target (PlayerActivatable)>>

Description

This command activates all PlayerActivatable components on the specified GameObject. This is typically used to trigger certain interactive elements in the game, such as switches or buttons.

Usage Example

```
// Example of activating an Elevator
<<activate "Elevator">>
```

In this example, the command activates all PlayerActivatable components on the GameObject named Elevator. This might trigger the elevator to move or become operational.

See Also

- set_enabled: Enables or disables a PlayerActivatable.
- toggle_animation: Toggles an animation on a GameObject.

---

animate

Syntax

<<animate target (SimpleAnimationTrigger)>>

Description

This command starts playing the animation associated with the specified GameObject's SimpleAnimationTrigger component.

Usage Example

```
// Example of animating the Temple Gate
<<animate "TempleGate">>
```

In this example, the command starts the animation associated with the TempleGate GameObject.

See Also

- play_animation: Plays a specified animation state on a GameObject.
- toggle_animation: Toggles an animation on a GameObject.

---

disable_controls

Syntax

<<disable_controls target (TogglePlayerControls)>>

Description

This command disables the controls for the specified target, usually the player. You can use this during cutscenes.

Usage Example

```
// Example of controlling player controls
<<disable_controls "Player">>
<<wait 7>>
<<enable_controls "Player">>
```

In this example, the command disables all controls for the player, preventing any input actions.

See Also

- enable_controls: Enables the controls for a target.

---

enable_controls

Syntax

<<enable_controls target (TogglePlayerControls)>>

Description

This command enables the controls for the specified target, usually the player. You can use this after the player controls were disabled using disable_controls.

Usage Example

```
// Example of enabling player controls
<<disable_controls "Player">>
<<wait 7>>
<<enable_controls "Player">>
```

In this example, the command re-enables all controls for the player, allowing input actions.

See Also

- disable_controls: Disables the controls for a target.

---

get_distance

Syntax

```
<<if get_distance("object1", “object2”) < value>>
        //code
<<else>>
        //code
<<endif>>
```

Description

This command checks the distance between two gameobjects.

Usage Example

```
<<if get_distance("Player", "Goal") < 5>>
   You are close to the goal
<<else>>
   You are not close to the goal
<<endif>>
```

In this example, the command checks the distance between the player and the goal and displays an appropriate message.

See Also

- show_path: Shows a path between two objects
- wait_for: Waits for an object to approach another object

---

get_scene

Syntax

```
<<if get_scene() == “scene02”>>
        ..
<<else>
        ..
<<endif>>
```

Description

This command will read the current Unity scene name.

Usage Example

```
<<set $currentSceneName to get_scene()>>
```

In this example, the function loads the current scene name into a variable.

See Also

- set_scene: Loads another Unity scene.

give_item

Syntax

<<give_item target (Inventory) id (string)>>

Description

This command adds an item to the specified target's inventory.

Usage Example

```
// Example of giving a golden coin to the player
// coin should be defined in the folder Assets/InventoryItems
// Resources
<<give_item "Player" "coin">>
```

In this example, the command adds an item with the ID coin and the icon goldencoin to the player's inventory.

See Also

- take_item: Removes an item from the inventory.
- has_item: Checks if an item is in the inventory.

---

has_item

Syntax

```
<<if has_item("tag")>>
        //code
<<else>>
        //code
<<endif>>
```

Description

This command checks if a certain item is in the inventory.

Usage Example

```
<<if has_item("coin")>>
   You have a coin.
<<else>>
   You don't have a coin.
<<endif>>
```

In this example, the command checks if the player has an item with the tag coin and displays an appropriate message.

See Also

- give_item: Adds an item to the inventory.
- take_item: Removes an item from the inventory.

---

hide_path

Syntax

<<hide_path>>

Description

This command hides the current navigation path that is being shown.

Usage Example

```
<<show_path "Player" "NPC">>
<<wait 7>>
<<hide_path>>
```

In this example, the command hides any navigation path that is currently displayed.

See Also

- show_path: Displays a navigation path between two GameObjects.

---

load_scene

Syntax

<<load_scene name (string)>>

Description

This command will load another Unity scene.
Scenes must be added to File->Build Settings first.

Usage Example

```
<<load_scene "InsideTheHouse">>
<<stop>>
```

In this example, the command loads another scene and stops the script.

See Also

- get_scene: Reads the name of the current scene

log

Syntax

<<log message (String)>>

Description

This command will show a message in Unity's debug log.

Usage Example

```
title: Start
---
<<log "Hello world">>
===
```

In this example, the command shows "Hello world" in the debug log.
If it is hidden, bring it up using Window->Panels->Console

play_animation

Syntax

<<play_animation target (GameObject) name (string) [shouldWait (bool) = True]>>

Description

This command plays a specified animation state on the target GameObject.

Usage Example

```
// Example of playing the TurningAnimation on the Ferris Wheel without waiting
<<play_animation "FerrisWheel" "TurningAnimation" false>>
```

In this example, the command plays the TurningAnimation on the FerrisWheel GameObject and does not wait for the animation to complete.

See Also

- animate: Starts an animation associated with a SimpleAnimationTrigger.
- toggle_animation: Toggles an animation on a GameObject.

---

playerpref_load_int
playerpref_load_string
playerpref_load_bool

Syntax

<<set $variable to playerpref_load_type(id)>>

Description

Description This function loads a value from Unity's PlayerPrefs with the specified key (id) and type (type). The type can be string, int, or bool.

Usage Example

```
<<playerpref_save_int "slot1" 16>>

<<set $score=playerpref_load_int("slot1")>>
```

In this example, the player's score is saved to and loaded from PlayerPrefs.

See Also

- playerpref_save: Saves a value to PlayerPrefs.
- playerpref_exists: Checks if a key exists in PlayerPrefs.

---

playerpref_save_string
playerpref_save_int
playerpref_save_bool

Syntax

<<playerpref_save_type(id, value)>>

Description

These functions save a value to Unity's PlayerPrefs with the specified key (id) and type (type). The type can be string, int, or bool.

Usage Example

```
<<playerpref_save_int "slot1" 16>>

<<set $score=playerpref_load_int("slot1")>>
```

In this example, the player's score is saved to and loaded from PlayerPrefs.

See Also

- playerpref_load: Loads a value from PlayerPrefs.
- playerpref_exists: Checks if a key exists in PlayerPrefs.

playerpref_exists

Syntax

`<<playerpref_exists id>>`

Description This command checks if a key (`id`) exists in Unity's PlayerPrefs.

Usage Example

```
// Check if player's name exists in PlayerPrefs
<<if playerpref_exists "player_name">>
   Player name exists!
<<else>>
   No player name found.
<<endif>>
```

In this example, the script checks if the player's name has been saved in PlayerPrefs and displays a message accordingly.

See Also

- `playerpref_save`: Saves a value to PlayerPrefs.
- `playerpref_load`: Loads a value from PlayerPrefs.

respawn

Syntax

<<respawn target (RespawnBehaviour)>>

Description

This command respawns the player at a specified location.

Usage Example

```
// Example of respawning the player
<<respawn "Player">>
```

In this example, the command respawns the player at their designated respawn location.

See Also

- set_spawn: Sets the spawn point to a certain GameObject.
- teleport: Teleports a GameObject to a location.

---

say

Syntax

<<say target (GameObject) message (String) [offset_y (number)=0] [shouldWait (bool) = True]>>

Description

This command shows a speech bubble with the specified message at the target GameObject.

Usage Example

```
// Example of making the Tavern Keeper speak
<<say "TavernKeeperHead" "Welcome traveler, what can I get for you?">>
<<say "TavernKeeperHead" "We got some fine ale from the Royal Kingston brewery">>
```

In this example, the command displays the message "Welcome traveler, what can I get for you?" as a speech bubble above the TavernKeeperHead GameObject.

See Also

- wait_for: Waits for a condition to be met before continuing.

Syntax

<<set_enabled target (PlayerActivatable) [mode (bool) = True]>>

set_enabled

Description

This command enables or disables a PlayerActivatable component on the target GameObject. You can for instance disable objects at the start of the story and enable them later on in a dialogue.

Usage Example

```
// Example of enabling an interactable NPC
<<set_enabled "InteractableNPC" false>>
<<wait 7>>
<<set_enabled "InteractableNPC" true>>
```

In this example, the command enables and disables the PlayerActivatable component on the InteractableNPC GameObject.

See Also

- activate: Activates all PlayerActivatable components on a GameObject.
- set_enabled_in_unity: Completely enables or disables a GameObject in Unity

---

Syntax

<<set_enabled_in_unity target (PlayerActivatable) [mode (bool) = True]>>

set_enabled_in_unity

Description

This command enables or disables the target GameObject entirely.
You can for instance disable objects to make them disappear.
This command is different from set_enabled, as it completely enables or disables the object in Unity, rendering it invisible and disabling (most) interactions.

Usage Example

```
// Example of fully enabling an interactable NPC
<<set_enabled_in_unity "NPC" false>>
<<wait 7>>
<<set_enabled_in_unity "NPC" true>>
```

In this example, the command enables and disables the NPC GameObject.

See Also

- activate: Activates all PlayerActivatable components on a GameObject.
- set_enabled: Enables or disables all PlayerActivatable components on a GameObject

---

set_spawn

Syntax

<<set_spawn target (RespawnBehaviour) spawnPoint (GameObject)>>

Description

This command sets the spawn point to the specified GameObject.

Usage Example

```
// Example of setting the spawn point to Chapter2Spawn
<<set_spawn "Player" "Chapter2Spawn">>
```

In this example, the command sets the player's respawn point to the Chapter2Spawn GameObject.

See Also

- respawn: Respawns the player at a certain location.

---

show_menu

Syntax

<<show_menu gameObject>>

Description

This command shows a UI menu. Technically, it will activate it. It will search the entire hierarchy to find the inactive object, which makes it slow for large projects.

Usage Example

```
// Example of showing a UI menu from the script
<<show_menu "CodeLock">>
```

In this example, the command will show the CodeLock menu from the script.

See Also

- wait_until_close: Waits for the UI element to be closed

---

show_path

Syntax

<<show_path from (GameObject) to (GameObject) [shouldWait (bool) = True] [minDistance (number) = 2]>>

Description

This command shows a navigation path from one GameObject to another.

Usage Example

```
// Example of showing a path from the Player to the Castle Gate
<<show_path "Player" "CastleGate">>
```

In this example, the command shows a navigation path from the player to the CastleGate GameObject.

See Also

- hide_path: Hides the current path that is shown.

---

take_item

Syntax

<<take_item target (Inventory) id (string)>>

Description

This command removes an item from the specified target's inventory.

Usage Example

```
// Example of taking a coin from the player
// The id can be set in give_item or in the AddInventoryItem component
<<take_item "Player" "coin">>
```

In this example, the command removes the item with the ID coin from the player's inventory.

See Also

- give_item: Adds an item to the inventory.
- has_item: Checks if an item is in the inventory.

---

teleport

Syntax

<<teleport target (RespawnBehaviour) destination (GameObject)>>

Description

This command teleports the specified target to a destination GameObject.

Usage Example

```
// Example of teleporting the player to the Dungeon Chapter
<<teleport "Player" "DungeonChapter">>
```

In this example, the command teleports the player to the DungeonChapter GameObject.

See Also

- respawn: Respawns the player at a certain location.

---

toggle_animation

Syntax

<<toggle_animation target (AnimationToggle)>>

Description

This command toggles the animation on the specified GameObject's AnimationToggle component.

Usage Example

```
// Example of toggling the Boat animation
<<toggle_animation "Boat">>
```

In this example, the command toggles the animation associated with the Boat GameObject.

See Also

- play_animation: Plays a specified animation state on a GameObject.
- animate: Starts an animation associated with a SimpleAnimationTrigger.

---

walk_to

Syntax

<<walk_to target (WalkBehaviour) [destination (GameObject) =] [distance (number) = 0] [shouldWait (bool) = True]>>

Description

This command makes an NPC walk to a specified location. It needs to be a Character setup and have WalkToBehaviour added. (The demo scene has an NPC setup)

Usage Example

```
// Example of making an NPC walk to the Sword Puzzle
<<walk_to "NPC" "SwordPuzzle">>
```

In this example, the command makes the NPC walk to the SwordPuzzle GameObject.

See Also

- wait_for: Waits for a condition to be met before continuing.

---

wait_for_location

Syntax

<<wait_for_location gameObject (GameObject) target (GameObject) [threshold (number) = 1.5]>>

Description

This command waits for one GameObject to be within a certain distance of another GameObject.

Usage Example

```
// Example of waiting for the player to approach an NPC
<<wait_for_location "Player" "NPC">>
```

In this example, the command waits for the player to be within 1.5 units of the NPC.

See Also

- walk_to: Makes an NPC walk to a location.

---

wait_until_closed

Syntax

<<wait_until_closed gameObject>>

Description

This command waits for a UI to be closed. Technically, it first waits for it to show, then it waits for it to close again.

Usage Example

```
// Example of waiting for the player to close a UI element
<<wait_until_closed "Letter">>
```

In this example, the command waits for the player close the Letter UI.

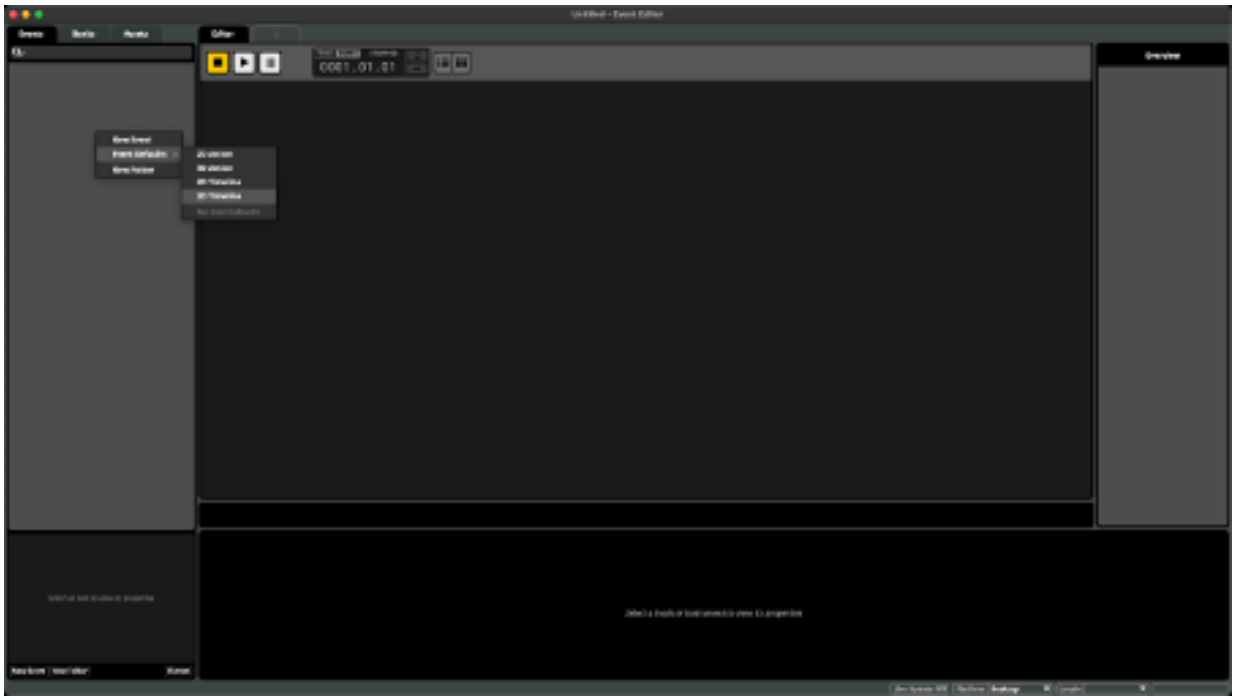
See Also

- show_menu: Shows a UI node

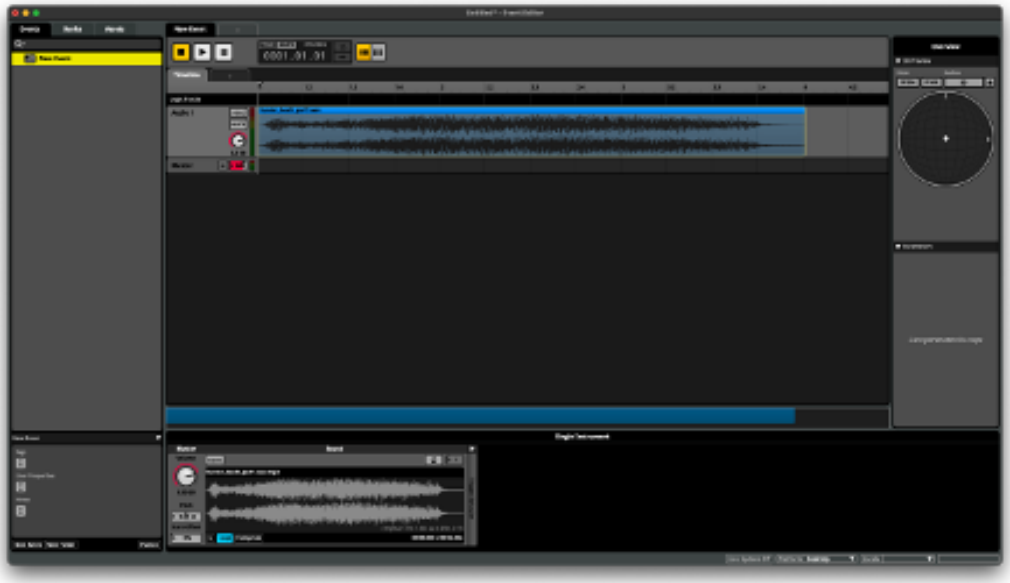
FMOD Integration Setup for Unity Project

1. Visit FMOD's Website:
    - Navigate to fmod.com.
    - Create a free account and head over to the download section.
    - Download both FMOD Studio (2.02) and the FMOD for Unity integration (2.02) package.

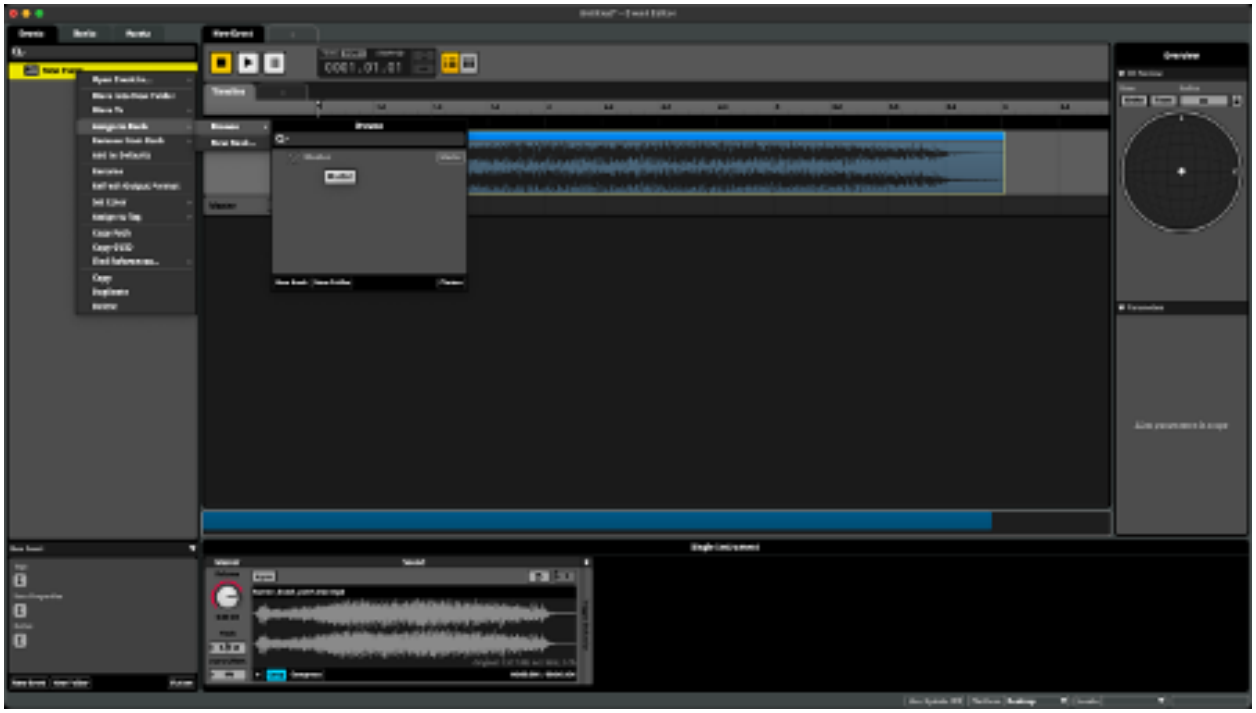
2. Create a New FMOD Project:
    - Launch FMOD Studio and create a new project.
3. Set Up an Event:
    - Create a new event using the 3D timeline.
    - Give the event an appropriate name to describe its purpose.



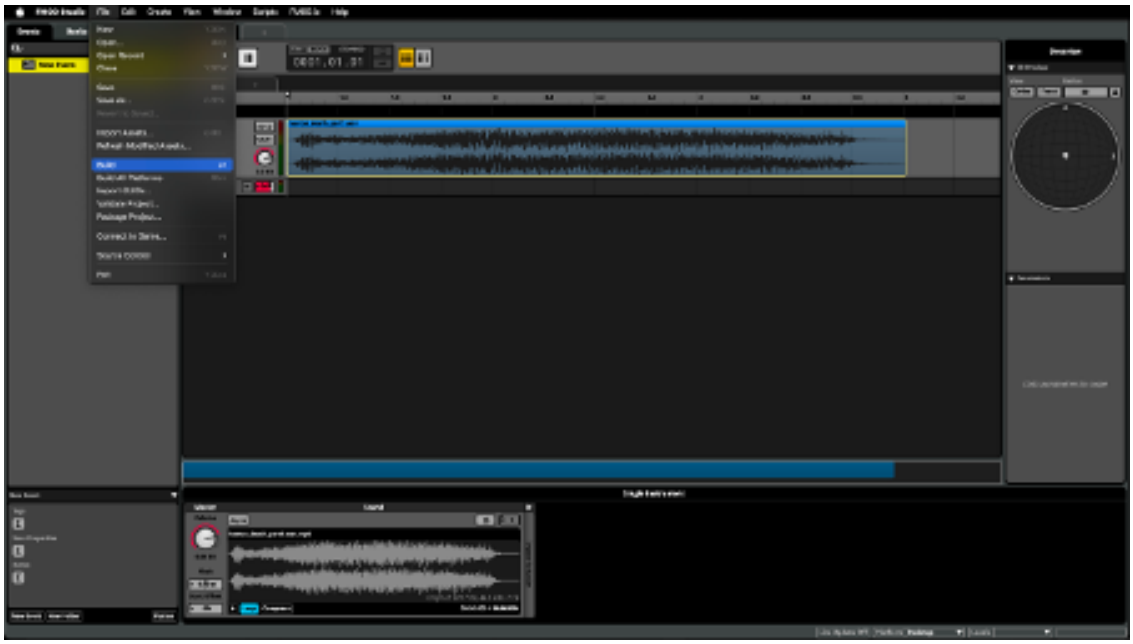
4. Add Sound to the Event:
    - Drag and drop a sound file onto audio channel 1 of your event.



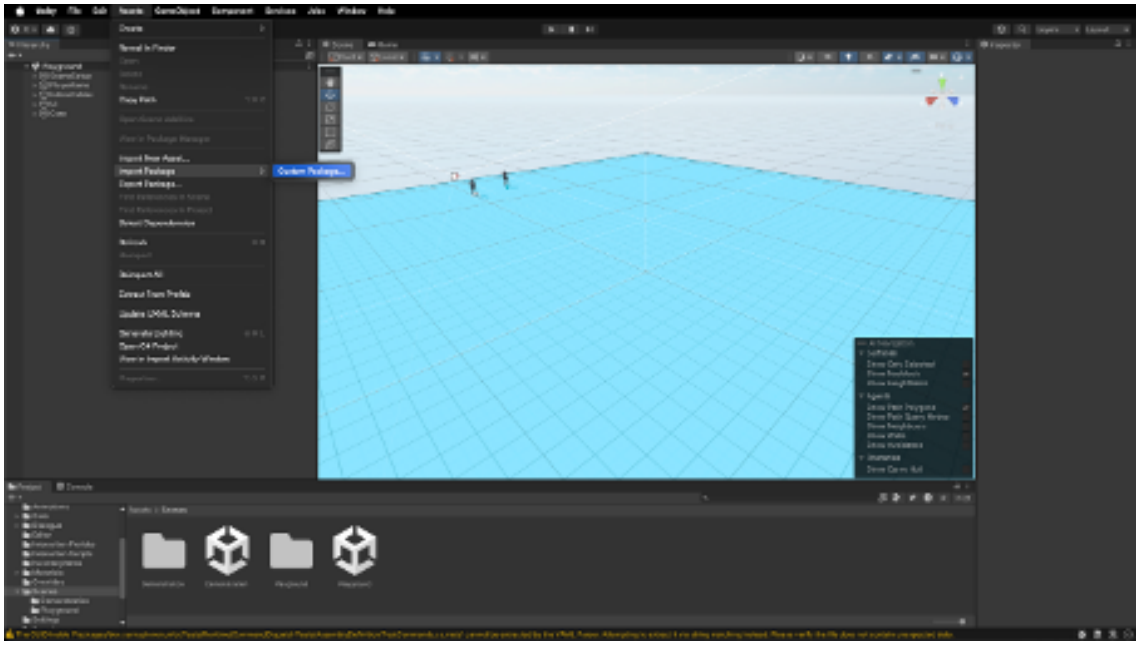
5. Assign Event to Bank:
    - Right-click on the event.
    - Select "Assign to Bank," then choose "Browse" and select the "Master Bank."

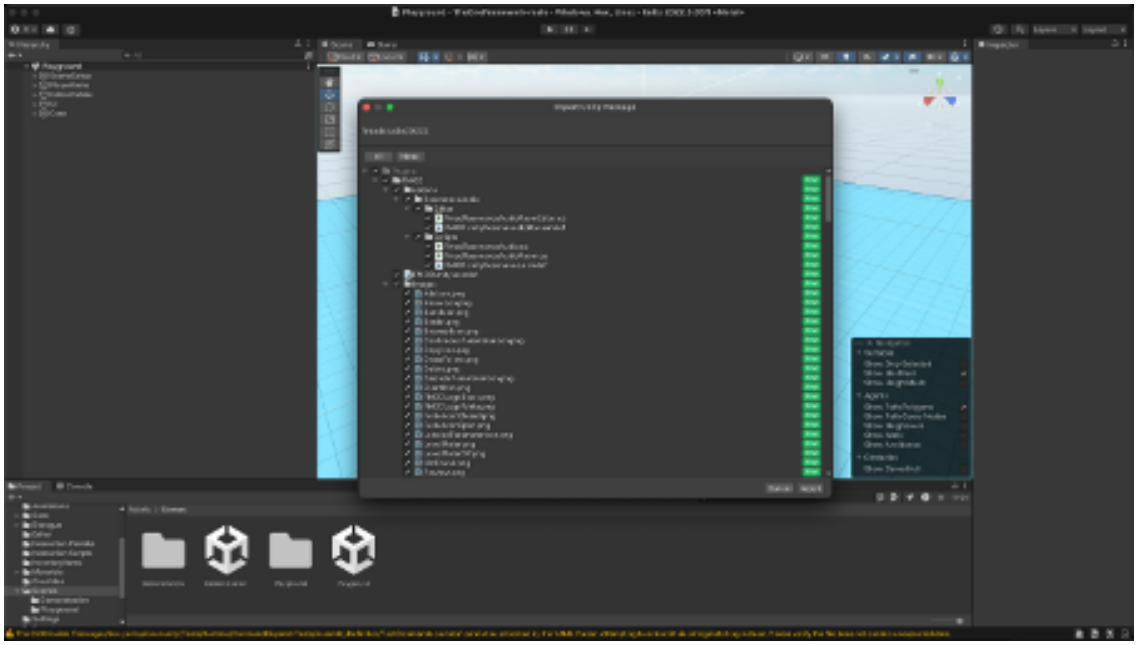


6. Organize Your Unity Project:
    - In your Unity project directory, create a new folder named "FMOD."
    - Save your FMOD project in this new "FMOD" folder.
7. Build the Bank:
    - Once everything is set up, build your bank to finalize the integration.

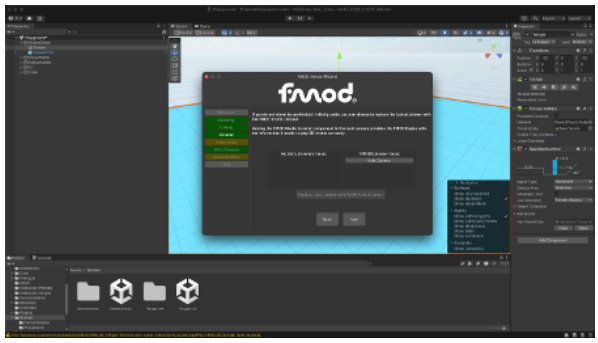


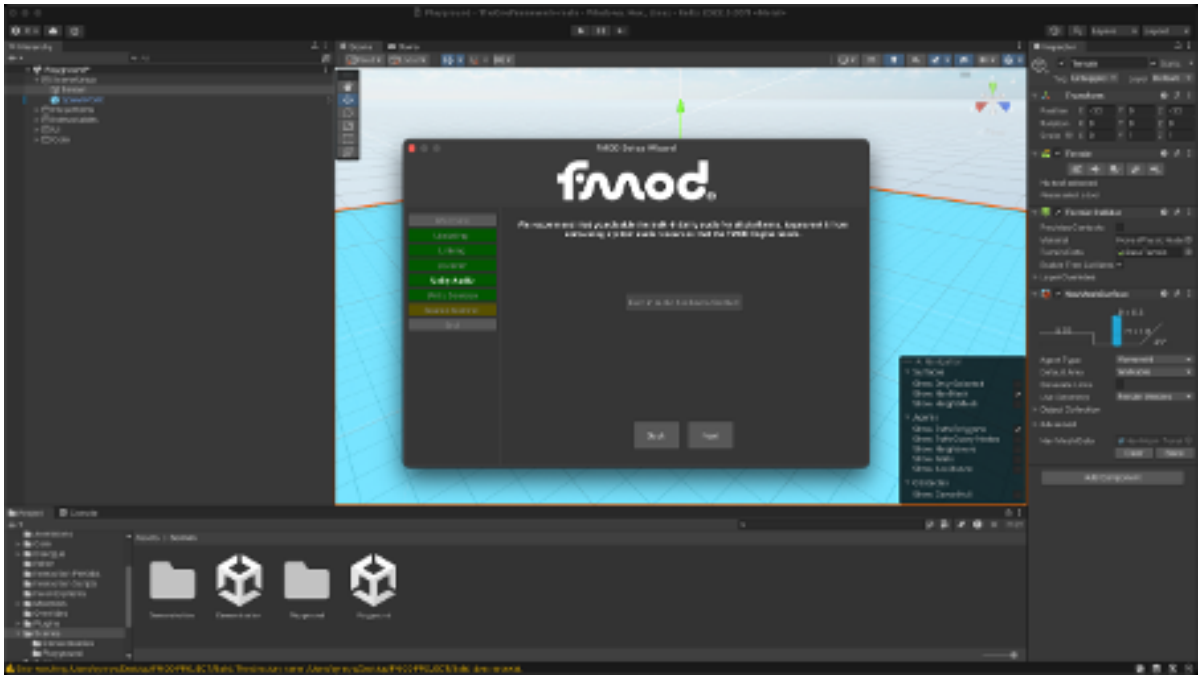
8. Import FMOD Unity Package:
    - Open your Unity project.
    - Go to Assets > Import Package > Custom Package.
    - Browse to the location where you downloaded the FMOD for Unity package and select it.
    - Click Import to add the package to your Unity project.





9. Run FMOD Setup Wizard:
    - After importing, a prompt for the FMOD setup wizard will appear.
    - Follow the setup wizard steps:
        i. Step 1: Locate and select your FMOD Studio project by clicking Browse.
        ii. Step 2: Replace Unity Listener with FMOD Audio Listener.
        iii. Step 3: Disable Unity built-in Audio.
        iv. Click Next on all other steps to complete the setup.





Installing YarnSpinner from git

Installing git
https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Installing YarnSpinner using git
Use url: https://github.com/YarnSpinnerTool/YarnSpinner-Unity.git

Unity 2019.3 and later

If you're using Unity 2019.3 or later, you can add the package directly:

1. In Unity, open the Window menu, and choose Package Manager.
2. Click the + button, and choose "Add package from git URL".

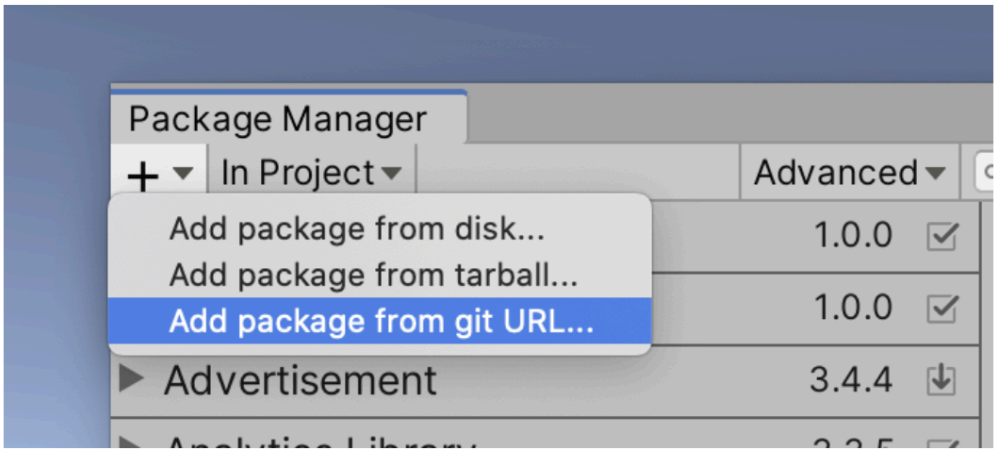


Figure 1: Adding a package using a Git URL.

1. Enter the following URL:
   `https://github.com/YarnSpinnerTool/YarnSpinner-Unity.git`.
2. The project will download and install.

Changelog

30 Jun 2024 - first version published internally
1 Jul 2024 - added Installation section
4 Jul 2024 - added FMod section