Thomas Zugrav – Game Programming Tutorials

Tutorial 1 – Player movement with gravity

Hello, in this tutorial I will teach you step by step on how to add WASD controls to a 'Player' object in Unity that also accounts for gravity!

By the end of this tutorial, you will know how to:

1 – Set up your project in Unity

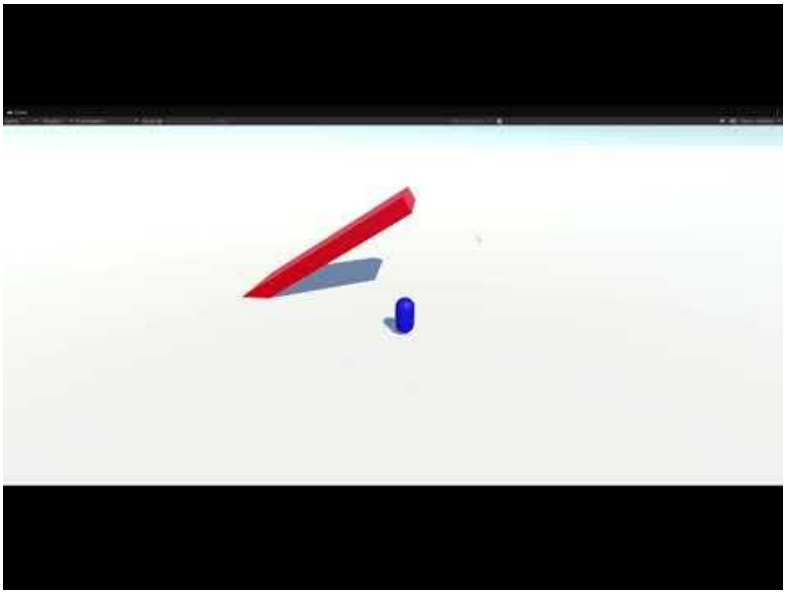
2 – Create a basic playable character

3 – Understand Unity's 'Character Controller'

4 – Create a basic C# script that will allow the use of WASD movement

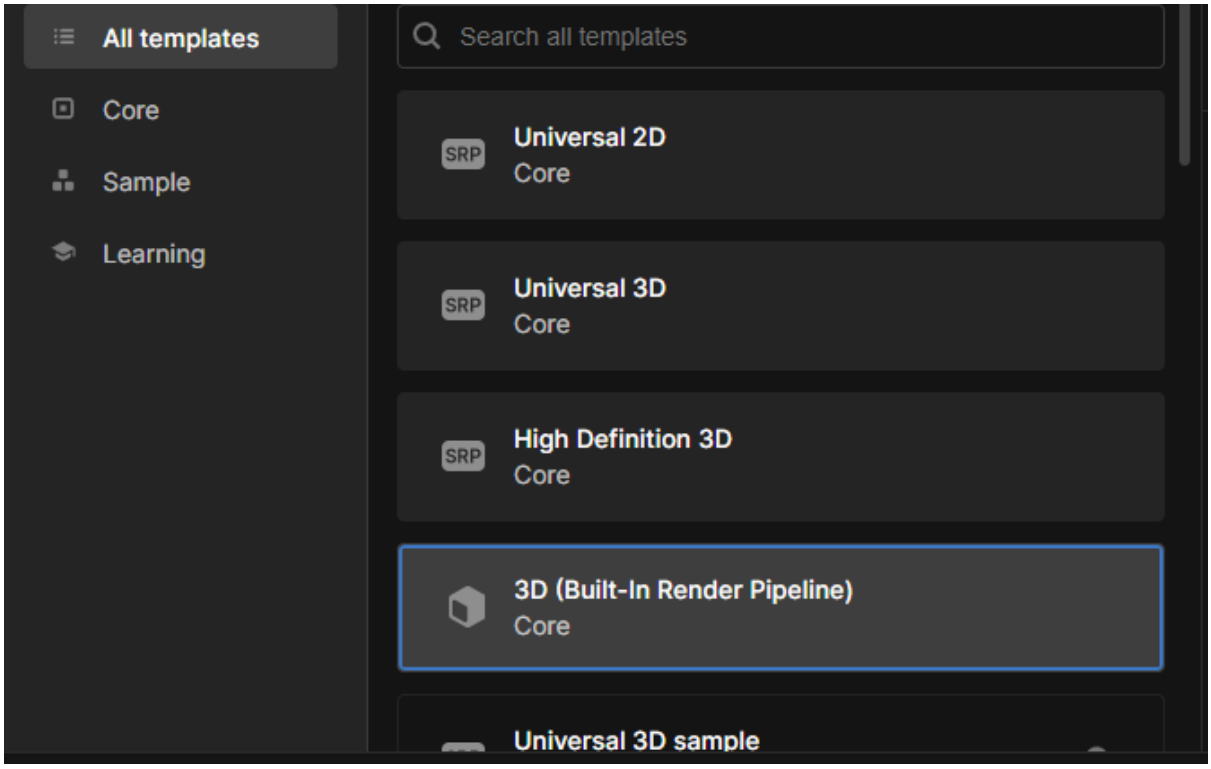
5 – Adjust this C# script to factor in gravity

By the end you should have something like this:



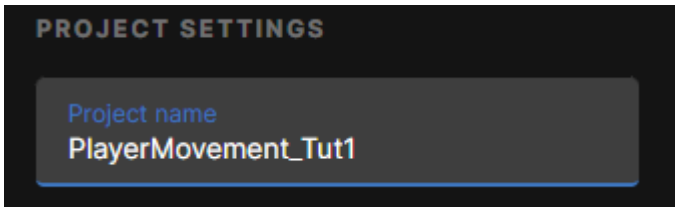
Let's begin by setting up our project in Unity.

We will first launch Unity and navigate to Projects where we will press ‘New project’.



You will then be presented with this template screen. Select ‘3D (Built-In Render Pipeline)’ this is the template that we will use for our project.

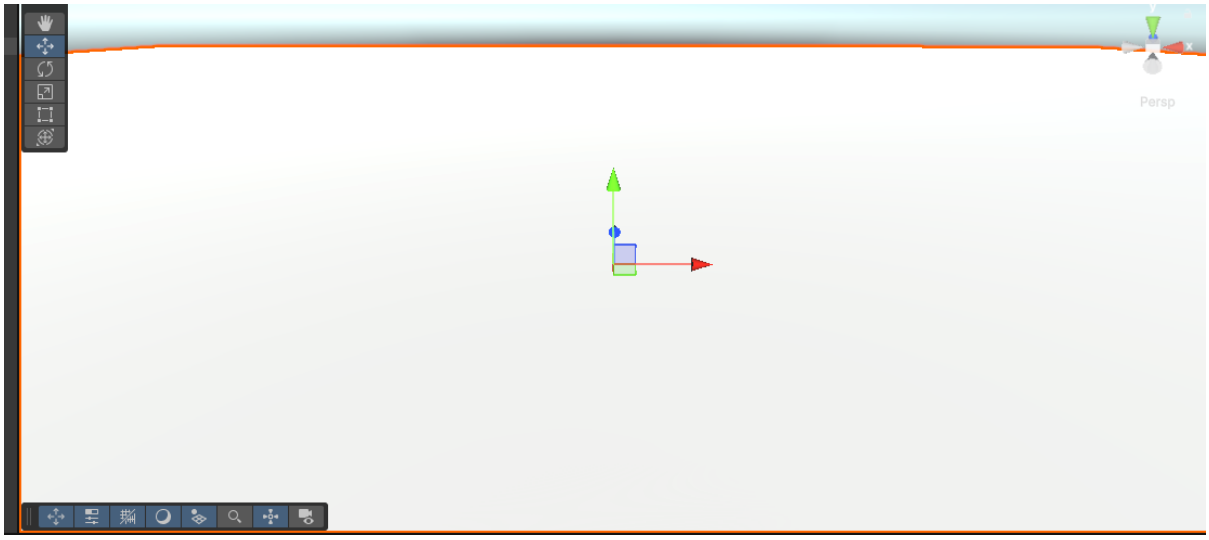
You should then name your project on the right side. In my case I will be naming it “Player_Movement_Tut1” but you should name it whatever fits best for you.



We will now be put into an empty Unity project.

Next, we’re going to head to ‘GameObject’ at the top of our screen then ‘3D object’ and let’s add a plane to our project.

Scale this plane out until it covers most of our view like so



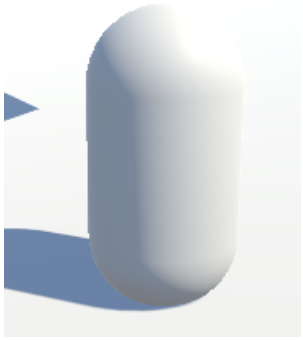
This will be our ground.

Now let's begin creating our playable character.

First head up to 'GameObject' again but this time create an Empty. This should be the first option in the dropdown menu. Name this 'Player'. We will then select this empty in our Hierarchy (located in the top left), right click it head down to 3D Object and select 'Capsule'. Finally we will right click 'Player' In the hierarchy once again this time creating an empty named "GroundCheck" and positioning it at the bottom end of our Capsule shape. We will be using this empty later when implementing gravity.

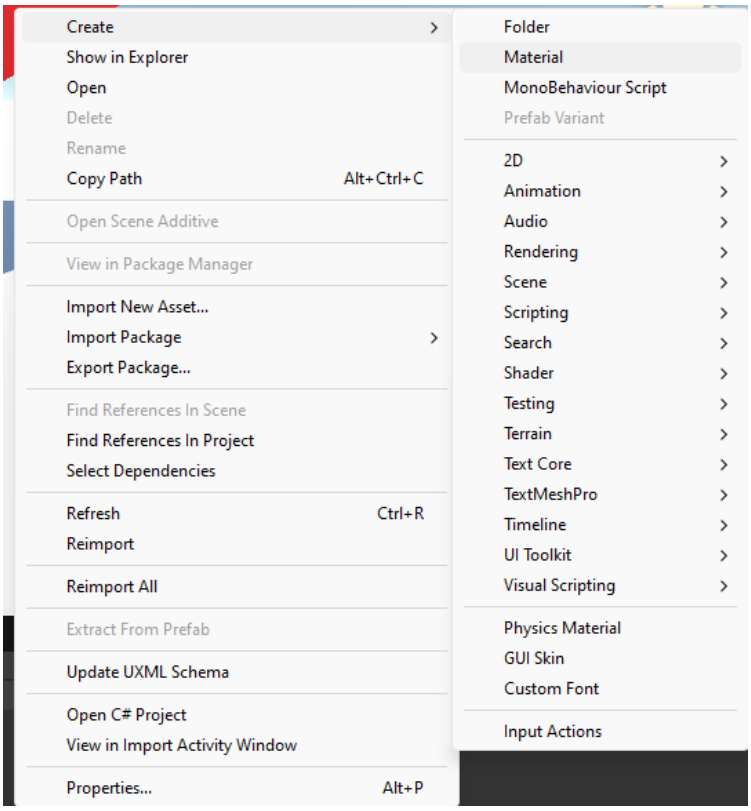
So far you should have this:

And a white capsule shape in our scene. This will be our player model.

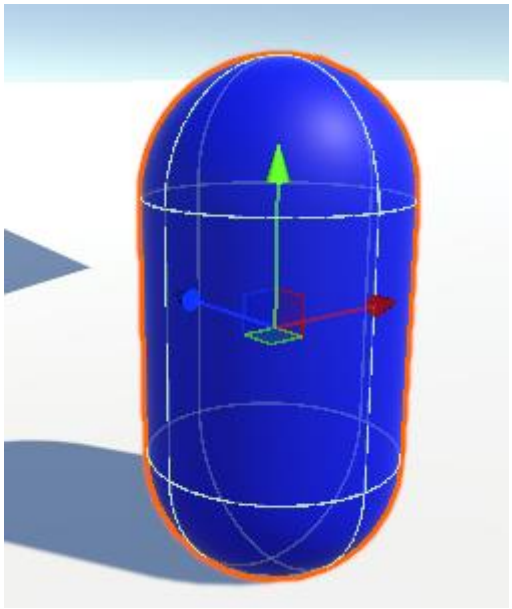


We can also go ahead and make a material for it to help us distinguish the player easily over the white ground.

We will right click in our assets at the bottom of the screen, head to create then create new material like so

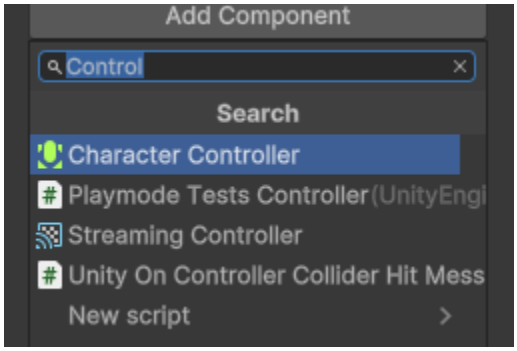


Then double click the new material in your assets head over to ‘Albedo’ under ‘Main Maps’ in the top right and chose a colour of your choice. I will go with blue. We will then drag the material in our assets onto our player.



Perfect, now it is time to add a character controller to our layer. A character controller will provide our player with collisions and many parameters that we can change to customize how the player interacts in the game.

To add a character controller simply select the ‘Player’ in the hierarchy, head to the inspector on the right, click add component and search for ‘Character Controller’.



However, as I have mentioned, the character controller will add collisions to our player, this means we will have to delete the provided ‘Capsule Collider’ which will be found under the empty ‘Capsule’ under our player in the hierarchy. Once you selected the capsule look in the inspector to the right, right click on the ‘Capsule Collider’ component and click ‘Remove Component’. Now we will not have any conflicting collisions.

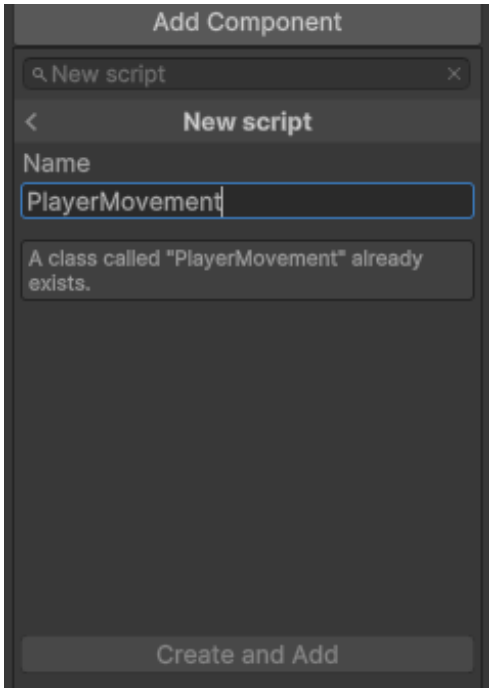
Okay. So far, we should have:

1 – A large plane that will act as our ground.

2 – A Player group in the hierarchy which holds two empties being the ‘Capsule’ and ‘Ground check’ and a material assigned to it.

3 – A character controller component assigned to the Player group.

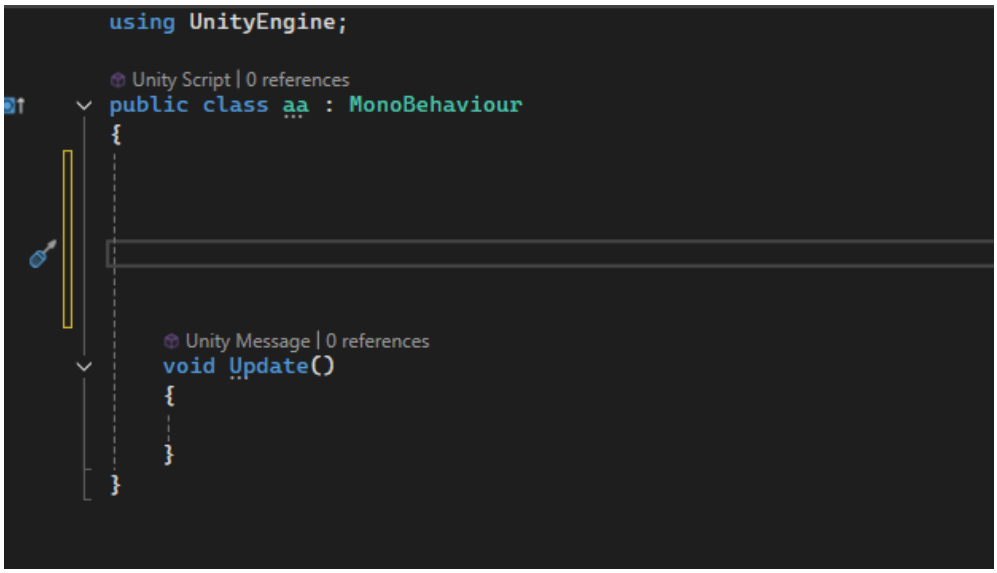
Now we will begin implementing our WASD controls and gravity. First select Player in the hierarchy and in the inspector add a new component, search for ‘New script’ click on what shows up and name your script something a long the lines of “PlayerMovement”. Click create and add at the bottom.



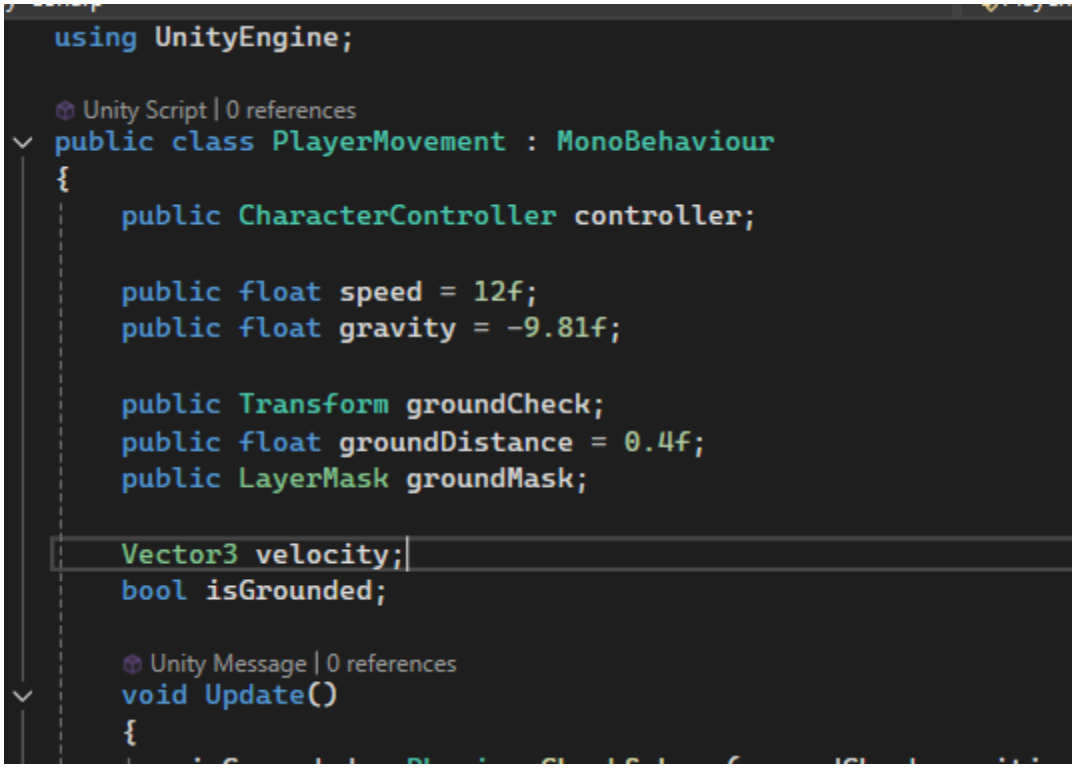
You have now created and assigned a C# script to our Player that we will implement our controls in.

Double click the newly created script in our assets and it should open Microsoft Visual Studio, if this does not open you may need to install it.

Great, now we will begin by deleting the two green lines that begin with “//”. These are just default comments that come with visual studio to explain what the void star and void update mean however we shouldn’t worry about that. Also delete the “void Start()” and the two curly brackets under it as we will not be needed them and it will make our script look cleaner. Now you should have something that looks like this.



Cool now let's begin writing code. First, we will begin by declaring some variables under our public class. We will be adding floats for speed, gravity and grounddistance also a transform, vector3, layermask, bool and a reference to our character controller. Do not worry too much about what this means for now however we must define these variables first so that we can reference them later. We will declare these variables like so.



The public floats for speed and gravity will let us customise the speed and gravity levels of our project in the future. The transform, layermask and bool will be needed when we implement our gravity. When writing this out make sure you copy it exactly as I have written it as programming languages consider things like capitalization and are very specific.

Now we can move on to the void Update().

```
Unity Message | 0 references
void Update()
{
    isGrounded = Physics.CheckSphere(groundCheck.position, groundDistance, groundMask);

    if (isGrounded && velocity.y < 0)
    {
        velocity.y = -2f;
    }

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

    Vector3 move = new Vector3(x, 0f, z);


    controller.Move(move * speed * Time.deltaTime);

    velocity.y += gravity * Time.deltaTime;


    controller.Move(velocity * Time.deltaTime);
}
}
```

Without overloading you with too much information I will just break down what each line of code does in a simple way. Let's ignore the code that adds the gravity as we I will explain how the controls work first. So, the first line of code that is responsible for player movement begins at "float x = Input.GetAxis("Horizontal");" and the line under it "float z = Input.GetAxis("Vertical");" this is how we define which keys the person playing our game must press for the character in game to move. We could also do this by assigning the keycodes to it such as W, A, S, D but we can also use the inputs "Horizontal" and "Vertical" which is what we have done because Unity will recognise this as up left down and right such as if we were using WASD. It is the input for our arrow keys.

The next line under that is

"Vector3 move = new Vector3(x, 0f, z);"

this is a calculation that works with the horizontal and vertical inputs above it which will represent the direction the player wants to move in. We store this in a variable called "move" which we then reference easily in the next line

"controller.Move(move * speed * Time.deltaTime);"

to actually allow the player to move in the direction we calculated earlier. We use “speed” in this calculation to make sure the player moves at the right speed and “Time.deltaTime” to ensure the player moves at a consistent speed. The “speed” in this calculation is what we actually defined at the beginning under “public float speed = 12f;” changing the “12f;” to a higher or lower number will cause our player to move quicker or slower.

Now the gravity is created by checking if the player is touching the ground or not then adjusting the velocity of the player based on the check. The check is done by a CheckSphere we actually added the check already earlier in our project under Player in the hierarchy called “GroundCheck” the actual sphere however is made in our script using this part of the code.

```
Unity Message | 0 references
void Update()
{
    isGrounded = Physics.CheckSphere(groundCheck.position, groundDistance, groundMask);

    if (isGrounded && velocity.y < 0)
    {
        velocity.y = -2f;
    }
```

The first line checks if the player is on the ground by seeing if the sphere collides with the groundMask. We will be applying the groundMask to our plane which is our ground later.

The code under it controls the Y axis velocity of the player when it is grounded.

```
        velocity.y += gravity * Time.deltaTime;


        controller.Move(velocity * Time.deltaTime);
    }
}
```

And finally this code controls the velocity of the player when they are not grounded (falling) and applies the movement of the player including the gravity. The gravity variable we created at the start currently set to -9.81f is what controls the gravitation strength you can set this to whatever you’d like for the kind of gravity needed for your game.

Once you finish writing this code there is just one step left for it to be functional!

Here is the full code to make it easier for you. https://pastebin.com/pDSwqCgn

Finally to finish setting this up we must hook up some things from the hierarchy to the inspector. Select Player in the hierarchy and in the inspector drag the character controller into the ‘Controller’ slot of our script, drag the ‘GroundCheck’ empty we created at the beginning from the hierarchy into the ‘Ground Check’ slot of our script in the inspector and change the “Ground Mask” dropdown to ground. If you cannot see ‘Ground’ we can simply create this layer by clicking on the ‘Layer’ drop down menu in the top right corner clicking ‘Add layer’ then typing ‘Ground’ in an empty ‘User Layer’ slot. Now select the plane we made and change it’s layer to ‘Ground’ also.

When pressing play you should now have a character that you can control using WASD and will fall correctly using gravity.

If you would like to test the gravity you can build a mini ramp using cube GameObjects like I did in my video earlier.

Finally, if you cannot see your player you may have to adjust the camera in your scene by rotating it and moving it out until the player is in view.

Thank you for reading!

Tutorial 2 – First Person Camera

Hey! For this tutorial I will be showing you how to create a working first person camera for your game. For this you will need to already have a character with player movement setup however, if you do not know how to do this, refer to my previous tutorial on player movement and gravity.

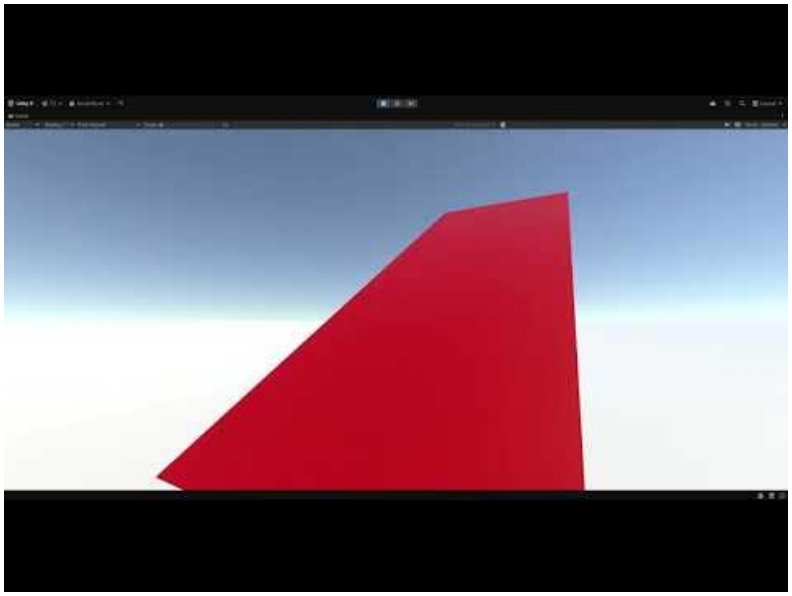
In this tutorial you will learn how to:

1 – Add a first-person camera to your player

2 – Create a Mouse Control script using C#

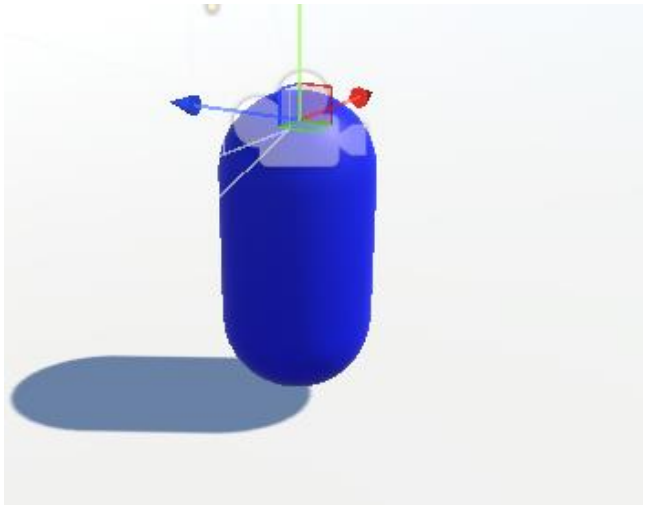
3 – Update your PlayerMovement script to fully implement the new camera

By following this tutorial, you should end up with something like this:



We will begin by creating a new camera and attaching it to the ‘Player’ in the hierarchy this will make the camera a child of the ‘Player’. Once you have done this, position the camera to be where the characters head would be but have it sitting slightly higher to avoid any clipping issues. In the future if you were to have the players head near a ceiling or tree or anything of the sort there could be clipping errors if we do not do this.

This is what you should have so far:



As you can see the top of the camera is slightly sitting outside of the players 'head' do this to avoid clipping.

Now we want to create a new script that will be attached to the camera. Todo this click on the camera in the hierarchy, head to the inspector on the right 'add component' at the bottom and type in 'new script' name this script something a long the lines of 'MouseControl'.

You can now double click this script to open it with Microsoft Visual Studio so we can edit it. We will now delete the green comments that begin with // as these just clutter our script. Okay, now that we are ready to edit this script let's begin by declaring some variables under our public class.

```
public class MouseControl : MonoBehaviour
{

    public float mouseSensitivity = 100f;

    public Transform playerBody;

    float xRotation = 0f;
```

We will add a public float for mouse sensitivity.

A public transform for our playerbody

And a float value for xRotation.

The mouse sensitivity float is what we will use later to decide the speed at which the camera moves based on mouse movement. The public transform playerbody is a reference to our character to allow us to rotate the entire player instead of just the camera when moving our mouse. Finally the xRotation float is there to prevent the camera from over rotating.

We can now move on to the Update() method. We will begin by adding in our mouse inputs so that we can use our mouse to move the camera when in game. We do this like so:

```
void Update()
{
    float mouseX = Input.GetAxis("Mouse X") * mouseSensitivity * Time.deltaTime;
    float mouseY = Input.GetAxis("Mouse Y") * mouseSensitivity * Time.deltaTime;
```

This allows us to be able to use our mouse on the X and Y axis to control the camera. We multiply this by our mouse sensitivity to determine the speed and by time.deltatime to ensure the movement is consistent across our frame rate. So now that we have introduced the mouseSensitivity variable in our update method any changes we make to the sensitivity float will affect the speed to our liking. A value of 100f is a good starting point to experiment with.

We now are reaching the final parts of the code to making our first person camera, continuing in the update method we will add this code:

```
        xRotation -= mouseY;
        xRotation = Mathf.Clamp(xRotation, -90f, 90f);

        transform.localRotation = Quaternion.Euler(xRotation, 0f, 0f);
        playerBody.Rotate(Vector3.up * mouseX);

    }
}
```

As I mentioned earlier the xRotation will be used in this part of the code now to make sure the camera can't over rotate. We do this by clamping the roation to a value of -90f and 90f. This will ensure the Camera will stop rotating once it hits those values. Finally, the last two lines of code are what are responsible for allowing the camera and player to move based on mouse movements.

To finalise our MouseControl script we can add this simple line of code in our Start() method so that our cursor becomes locked when we are playing our game pretty much making the cursor invisible when we are using the first person camera.

```
void Start()
{
    Cursor.lockState = CursorLockMode.Locked;
}
```

Great! Now our MouseControl script is finished, we have just a few more steps to finish implementing the first person camera to our game.

Here is the full code to make it easier for you to paste in or to have a look at! https://pastebin.com/D4FxM1HT

For the next step we will now move on the your script that manages player movement if you followed my previous tutorial it should be named something

along the lines of “PlayerMovement”. In this script there is one line of code we must change so that the player can use first person correctly. The line being this:

```
Vector3 move = new Vector3(x, 0f, z);
```

We will replace this line with “Vector3 move = transform.right * x + transform.forward * z;” this is to ensure that our player moves in the direction that the camera is facing instead of moving relative to the global world axes.

The final step now is just to link our Player to the Player Body slot under our Mouse Control script located in our Camera.

We do this by selecting our Camera in the hierarchy and heading to the bottom of the inspector where our script is showing, then simply drag the player from the hierarchy into this slot!

You have now fully implemented a working first person camera to your game! Press play and you should be able to move around and look around as if you were viewing things from a first person angle.

Tutorial 3 – Shift to sprint!

Hello, welcome to my tutorial on how to implement a shift to sprint feature in your game. This will be a relatively basic and easy to implement feature as we will not be adding any advanced systems such as stamina however, this script will be easy to update in the future and develop further.

For this tutorial I have assumed you already have a working player movement script in place that allows you to use WASD or Arrow Keys to move your player around. If you do not, you can refer to my first tutorial on player movement.

By the end of this tutorial, you will know how to

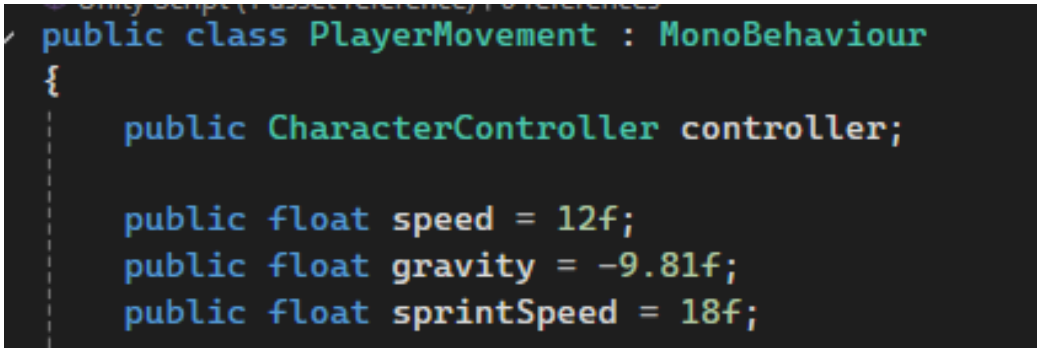
- Implement a sprinting mechanic to your game
- Update existing scripts to implement new features
- Work with if and else statements

To begin adding this feature into our game we will start with opening our script accountable for player movement. For me this will be the ‘PlayerMovement’ script and should be the same for you if you followed my movement tutorial. If not then just open up your player movement script and let’s begin adding sprint functionality.

Your script should look something like this:

```
Unity Script (1 asset reference) | 0 references
public class PlayerMovement : MonoBehaviour
{
    public CharacterController controller;

    public float speed = 12f;
    public float gravity = -9.81f;

    public Transform groundCheck;
    public float groundDistance = 0.4f;
    public LayerMask groundMask;

    Vector3 velocity;
    bool isGrounded;

    Unity Message | 0 references
    void Update()
    {
        isGrounded = Physics.CheckSphere(groundCheck.position, groundDistance, groundMask);

        if (isGrounded && velocity.y < 0)
        {
            velocity.y = -2f;
        }

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

        Vector3 move = transform.right * x + transform.forward * z;


        controller.Move(move * speed * Time.deltaTime);

        velocity.y += gravity * Time.deltaTime;


        controller.Move(velocity * Time.deltaTime);
    }
}
```

We will first begin by adding a new variable in our public class. The new variable we add is going to be a public float for sprintSpeed so that we can reference it later and also adjust it's value to determine the speed our player will have when sprinting. We will write this in like so



For our sprintSpeed value 18f is a good start however we can always adjust this later to increase or decrease the speed when sprinting. This all depends on what your game is and how sprinting will fit into your gameplay design.

Also keep in mind that we are just implementing a feature in our existing code. Do not copy extra code that I might have that you may not as this could lead to problems. If you want the exact code that I am using refer to my player movement and gravity tutorial so there will be less confusion.

Next we will modify some code in our Update() method to allow us to hold Shift when in game to change our players speed. I will show you how and why we do this:

```
Unity Message | 0 references
void Update()
{

    isGrounded = Physics.CheckSphere(groundCheck.position, groundDistance, groundMask);

    if(isGrounded && velocity.y < 0)
    {
        velocity.y = -2f;
    }

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



    Vector3 move = transform.right * x + transform.forward * z;

    if (Input.GetKey(KeyCode.LeftShift))
    {
        controller.Move(sprintSpeed * Time.deltaTime * move);
    }
    else
    {
        controller.Move(move * speed * Time.deltaTime);
    }

    velocity.y += gravity * Time.deltaTime;

    controller.Move(velocity * Time.deltaTime);

}
}
```

As you can see the script is for the most part unchanged aside from a major if/else statement that we must add for our sprinting to work.

```
Vector3 move = transform.right * x + transform.forward * z;

if (Input.GetKey(KeyCode.LeftShift))
{
    controller.Move(sprintSpeed * Time.deltaTime * move);
}
else
{
    controller.Move(move * speed * Time.deltaTime);
}
```

We pretty much have to add this so that when LeftShift is held the controller (player) will move with sprintSpeed if LeftShift is not held then the controller will move with the original speed value we have set at the beginning of our script. A common mistake is using GetKeyDown instead of GetKey. The difference is that GetKeyDown only triggers once when the key is pressed, whereas GetKey checks continuously while the key is held down. For sprinting, we want to use GetKey so that the player can keep sprinting as long as the key is held. Make sure you use the right one!

Finally, we can refine our script to make it more efficient by reordering a certain calculation. Visual studio actually spotted this already and is recommending us to change it. This is seen by the three dots under the blue 'move'. The best way to order this calculation is like so:

Controller.Move(speed * Time.deltaTime * move);

This isn't too important to do as the script will work either way however it is a good habit to get into to ensure that there aren't redundant calculations made. We pretty much will get the same outcome by writing the code out this way but in a more efficient way.

Now we can CTRL + S to save our script, exit out and try our game! You should be able to hold LeftShift to move a lot faster. If you would like to change the sprint key to something more custom you can always pick another Key. You must know the keycodes for our keyboard keys however, you can find these in Unitys documentation under KeyCode enumeration. E.g space would be KeyCode.Space.

If you want to increase your sprintSpeed simply change the sprintSpeed value at the top of our script for example if you want to be twice as fast when sprinting you can change it to 24f as our regular speed is 12f.

Thank you for reading!

Full code: https://pastebin.com/p8bSQarg

Tutorial 4 – Making a simple fetch quest

Hi! Welcome to my Unity tutorial on how to code a basic fetch quest in C#.

By the end of this tutorial, you will have learnt how to:

- **Setup an interactable NPC that will give the player a quest**
- **Setup a fetchable game object the player can pick up**
- **Link two scripts together to ensure compatibility**
- **Setup clear and helpful UI prompts/dialogue**

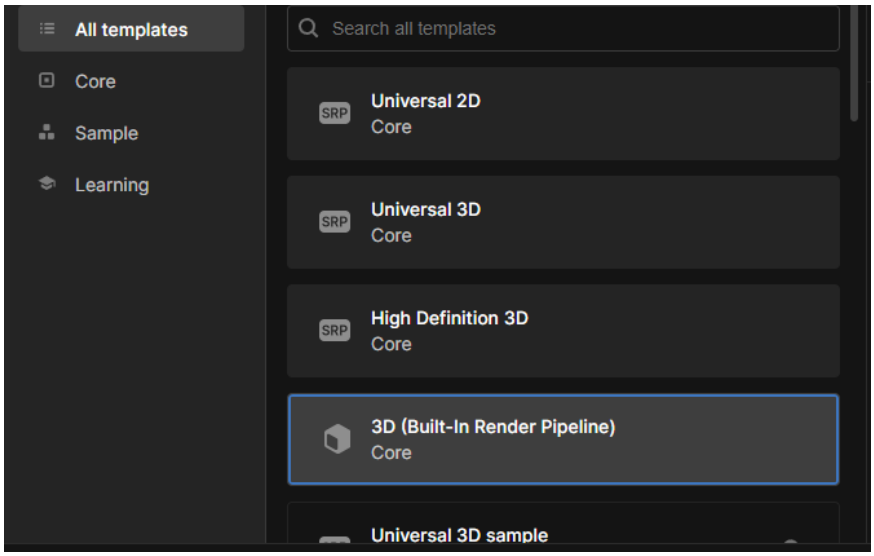
For this tutorial I highly recommend you already have a player with player movement implemented so you can walk around and interact with the quest we will be making today.

I'd also recommend creating a basic scene to fit in your game where your quest will take part in however, this isn't necessary as I will be going over how to make a basic environment for our quest anyway.

Let's start!

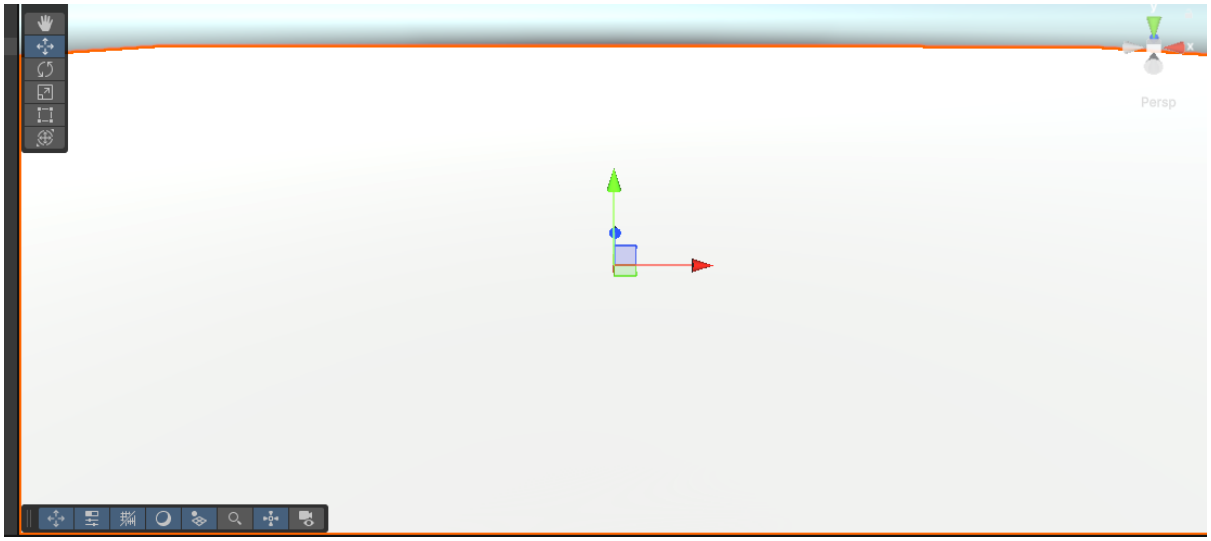
First let's create a new Unity project and begin making our basic environment. You can go as basic or advanced as you want but In my case I will just use basic shapes to make a building for my NPC and also ground to walk on and place our fetchable item on later.

We can create our project by opening Unity, clicking New Project in the top right corner and then naming our project and selecting the '3D (Built-In-Render Pipeline)'.

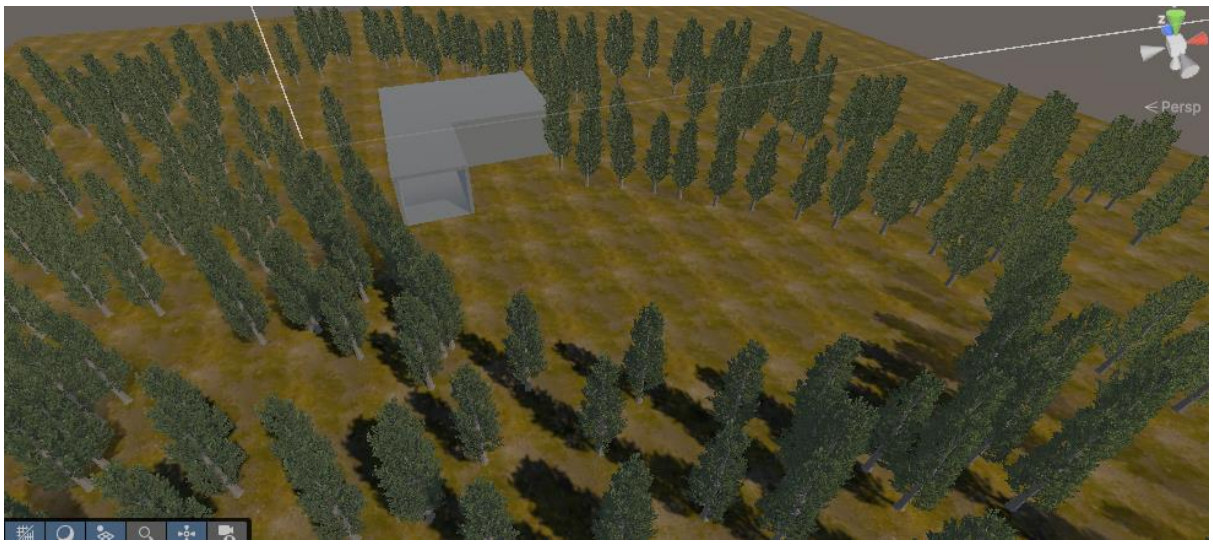


Setting up our Unity scene

We can now add in a Plane to our scene to use for the ground. To do this navigate to GameObject at the top then under '3D Object' select Plane. Scale this plane greatly to cover a big portion of our view like so



Now it is up to you on how you would like your environment to looks. You can create building or areas using basic objects like cubes and basic tools like the scale tool and move tool to position and build simple blockouts to populate your environment. I went with something simple like this.



It consists of ground, trees from the asset store and a building I made to put my NPC in.

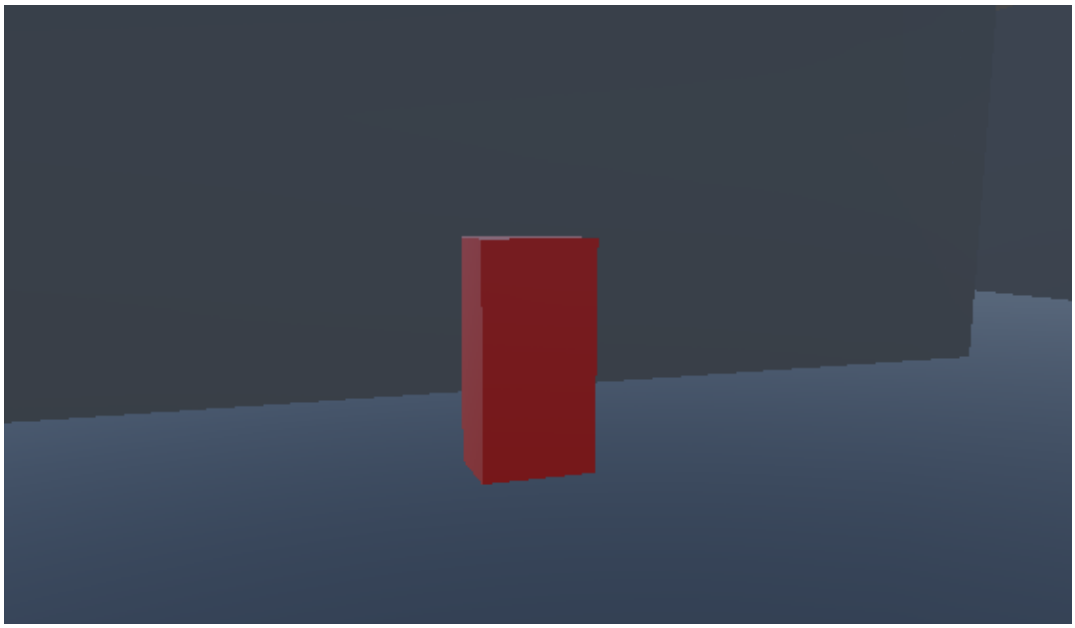
As you can see I have also added texture to the floor, but this isn't necessary.

(I just couldn’t stand the bright white).

Now import your player and your player movement script to this scene or create a new one from scratch so we can begin creating our quest. If you do not have a player character with movement you could use, refer to my first tutorial which covers Player movement and gravity and also my second tutorial on adding a first-person camera. That will be enough to allow you to start working with this tutorial.

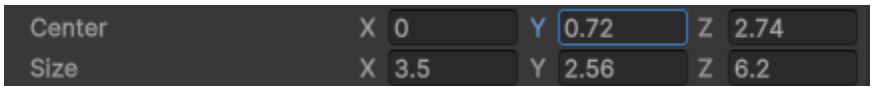
We will first begin by setting up our scene further by adding some placeholder assets that will be needed for our quest. This will be for: the quest NPC, the fetchable item the Quest Tracker UI and the Dialogue UI.

Lets begin by making our NPC! For making our NPC we can go with a simple shape like a Cube or a Sphere. I will go with a Cube. Add a cube to your scene by navigating to ‘GameObject’ like we did with the Plane then ‘3D Object’ and then select Cube. Place this cube by using the ‘Move’ tool wherever you want and scale it on the Y axis so that it resembles a rectangle. This will be our ‘NPC’. You can also create a new material by right clicking in the Assets at the bottom of your screen and creating ‘Material’ drag the newly created material onto your NPC then click on the newly created material in your Assets and modify the ‘Albedo’ in the top right to a colour of your liking. I will go with red.

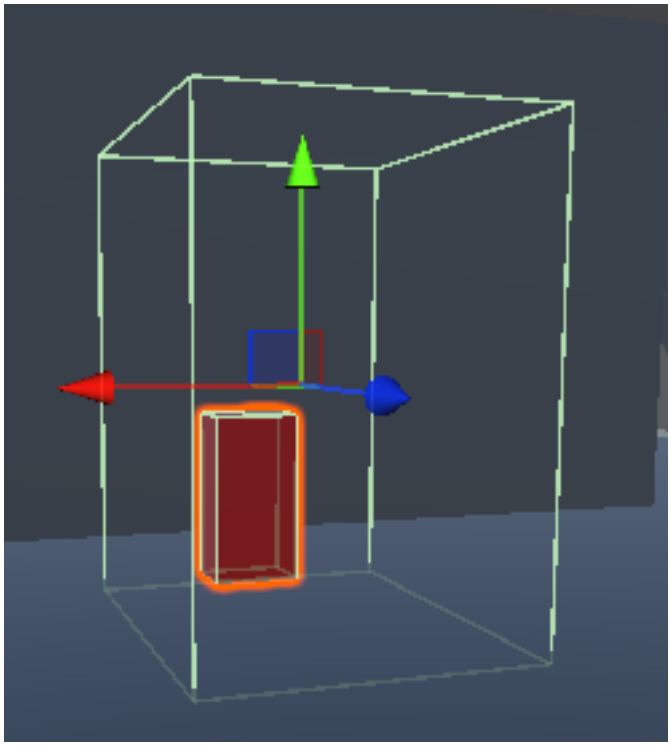


To save us from back tracking let’s also give our ‘NPC’ a triggerable box collider that we will be using later. To do this select your ‘NPC’ and go to the inspector

on the right. Click add component. Next search for ‘Box collider’ and click it. We will now update the size and position of this collider by modifying the X, Y, Z values under ‘Size’ and ‘Center’ in the inspector. I went with these values.



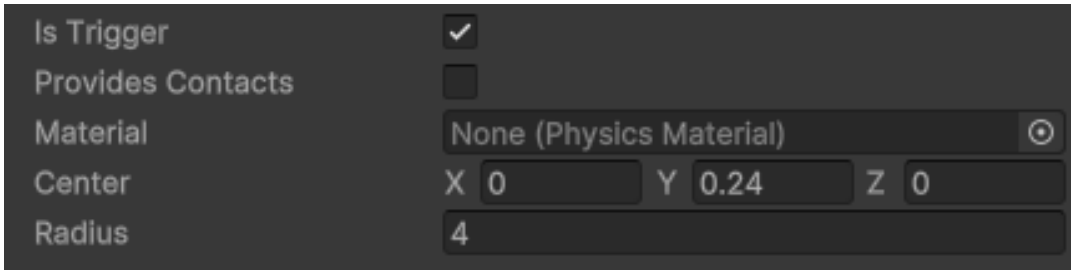
Your box collider should know look something like this.



We also must tick the ‘Is Trigger’ checkbox in our box collider so that we can use it as a trigger in our code later.

Now we can repeat these steps for adding our fetchable item however this time we will be adding a ‘Sphere’ instead of a cube and also adding a ‘Sphere Collider’ to this sphere instead of a box collider as it will shape it better.

The values I went with for the sphere collider size is as so:



Make sure you also tick the ‘Is Trigger’ box too.

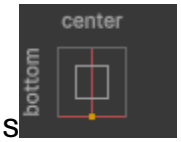
You should have something like this.

Position your ‘Item’ somewhere further but not too far from the NPC to simulate a fetch quest experience. For now, you can however place it near to the NPC for testing purposes, so you don’t have to walk 5 minutes to find your item whenever you want to test something 😊 .

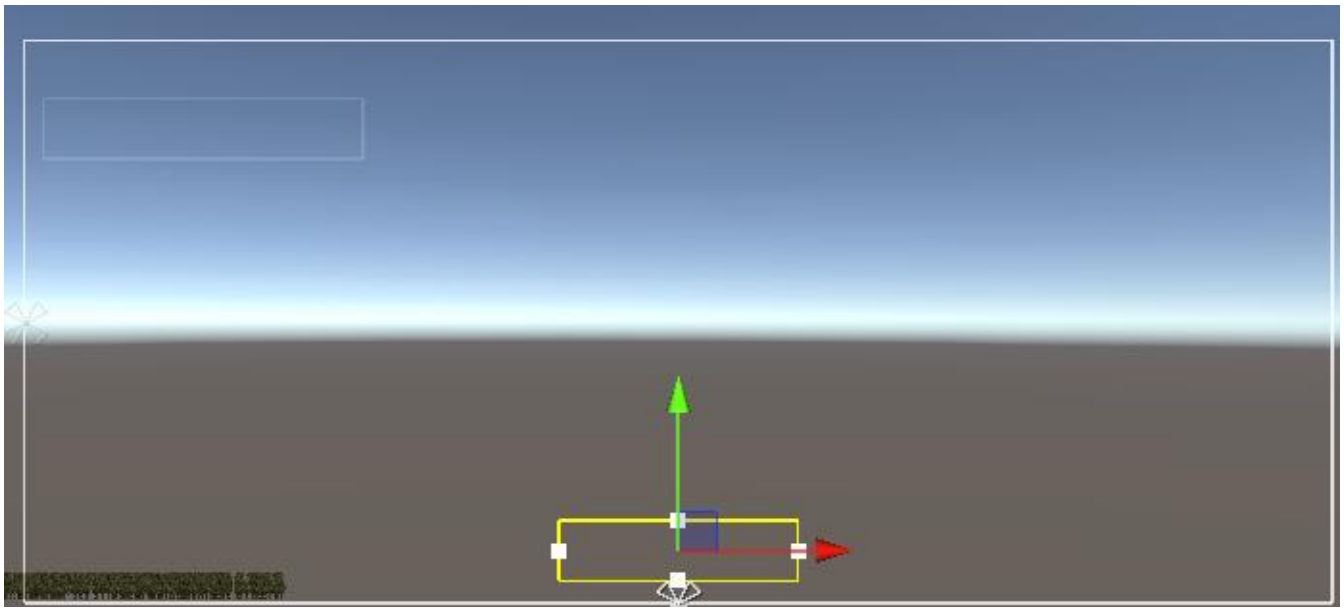
To finish setting up our scene we must now add our UI. Don’t worry, we will simply be using basic text boxes with clear fonts/sizing/colours for simple but straightforward UI. You can always customise and improve this later.

To begin adding our UI we must first create a ‘Canvas’ to do this first click on the ‘2D’ button near the top right of your scene view to put your view in 2D. Then head to the Hierarchy right click in an empty space head to > UI > Canvas (at the bottom) and add your canvas in. Name this something like “UI” or “User Interface”. We will then make two children for our UI by right clicking it then heading to UI > Text – TextMeshPro and adding your textmeshpro element in. Do this twice. Name the first one ‘QuestMessage’ and the second one ‘QuestTracker’

You should know have something like this in your Hierarchy:

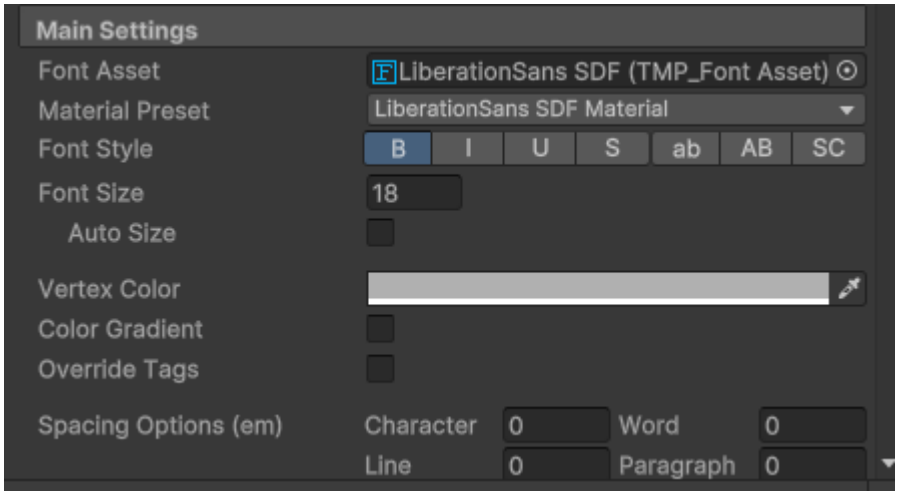
Next look back to your Scene view, select the QuestTracker, press W and move it somewhere to the top left of the canvas. Also set the 'Width' under 'Rect transform' in the inspector to 267. Then position the QuestMessage by selecting it in the Hierarchy and clicking on this 

in the inspector (topleft) holt ALT and select the Bottom Center option, then use the move tool to move it slightly up so it isn't stuck to the bottom of the canvas. Your canvas should look like this:

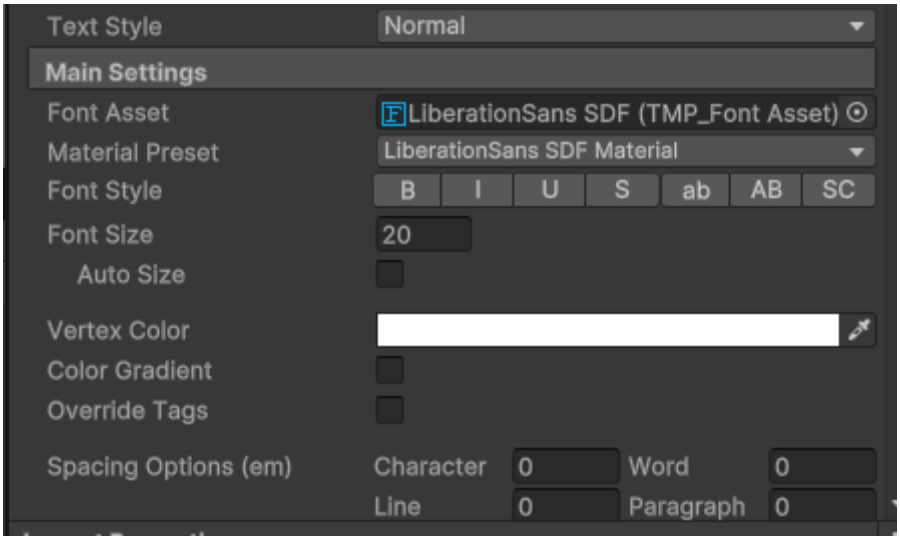


Pretty much once we finish our script the bottom text box will be used for things like item pick up and NPC interaction prompts while the top left text box will be our quest tracker (a prompt to remind the player of their current quest) and NPC dialogue.

Copy these settings for the text although you can experiment with different options.



For TextMessage ^ colour hex: B0B0B0



For QuestTracker ^ colour hex: FFFFFF

These settings are accessed by selecint either the QuestTracker or TextMessage UI element in the hierarchy then looking in the Inspector under ‘TextMeshPro’.

I chose these settings as I found the text looks clear and clean but you should experiment with them to see what fits best for you.

Now clear any text in the text inputs and lets move on. (We will be adding any text through script).

We have now finished setting up our Scene ready for coding! We have an NPC with a triggerable collider, an Item with a triggerable collider and our two UI text elements, we can now move over to the coding part of this tutorial.

Making our C# Scripts

Okay now we can begin writing out our scripts to make everything interactable. Pretty much our quest will work like this:

>The player goes to the NPC and gets prompted to accept a quest

>The player accepts the quest. They are prompted with details on how to complete the quest

>The player then fetches an item (required for quest completion)

>The player returns to the NPC where they are prompted to complete the quest.

So, because we are involving two GameObjects being the NPC and the Item, we will have to create two scrips and then reference them with one another.

Let's begin with the NPC script.

First select the NPC head to the bottom of the inspector and 'Add component' we will search for 'New script' name this script "NPCquest" and click create and add.

Double click the newly created script in our 'Assets' to open it up with Microsoft Visual Studio.

We will first delete the green comments that begin with // these are not necessary. We can then also delete the "Void Start()" method and the curly brackets underneath it as we will not need this.

Now we will continue by making sure TextMeshPro (what we used for our UI) is being recognised in our script we can do this by adding this line of code "using TMPro;" like so:

```
using UnityEngine;
using TMPro; // Include this to use TextMeshPro
```

As you can see I have also added some green comments throughout this script you do not need to copy these. I have added the comments just because there will be quite a lot going on in this tutorial so it can be easier for you to understand what each part of the code does and also as a reminder for myself or anyone else

reading it. Anything that's green and has // before it isn't required but could also help remind you in the future so copy them if you'd like.

Now let's begin declaring some important variables for our script to work.

```
using UnityEngine;
using TMPro; // Include this to use TextMeshPro

Unity Script (1 asset reference) | 1 reference
public class NPCQuest : MonoBehaviour
{
    private bool playerInRange = false;
    private bool questAccepted = false;
    private bool questComplete = false;
    public TextMeshProUGUI questMessage; // UI for NPC quest messages
    public TextMeshProUGUI questTracker; // UI for quest tracker in the top left

    2 references
    public bool IsQuestAccepted()
    {
        return questAccepted;
    }
```

Here we have a bool for playerInRange which we will use to decide if the player is in the collider of the NPC, a bool for questAccepted which we will use to determine if the quest has been accepted and a questComplete boolean for if the quest has been completed or not. As you can see we also have another bool at the bottom, this bool: "IsQuestAccepted()" will be used in our second script for our Item so that it can check if the quest is accepted in our NPC script before proceeding.

The "public TextMeshProUGUI questMessage;" and questTracker are for when we need to reference the UI text boxes we created earlier for updating dialogue or prompts.

Now we will declare some private methods at the bottom of our script these will let us control our UI text.

We will write these out below the curly brackets of the "void Update()"

```
    private void DisplayMessage(string message)
    {
        if (questMessage != null)
        {
            questMessage.text = message;
        }
    }

    1 reference
    private void ClearMessage()
    {
        if (questMessage != null)
        {
            questMessage.text = "";
        }
    }

    5 references
    private void SetQuestTracker(string message)
    {
        if (questTracker != null)
        {
            questTracker.text = message;
        }
    }


    0 references
    private void ClearQuestTracker()
    {
        if (questTracker != null)
        {
            questTracker.text = "";
        }
    }
}
```

The top two will let us display or clear the text in our Quest Message box on our UI while the bottom two will control the Quest Tracker instead.

Now just above the “private void DisplayMessage(string message)” make some space for this public method that we will add:

```
1 reference
public void MarkQuestComplete()
{
    questComplete = true;
    SetQuestTracker("Current Quest: Item collected! Return to the NPC to finish your quest.");
}
```

The function here is in place to not only update the quest tracker once the player collects the item but also to be referenced in our item script so that it knows when the quest is marked as completed. We can tell that this will be referenced in our other script because we made it ‘public’ that will allow us to reference it in scrips later whereas the “private void ClearQuestTracker()” for example, will only be used in THIS script.

Now let’s move back to the ‘void update()’ under here we will manage the interactability of the player and NPC.

```
Unity Message | 0 references
void Update()
{
    if (playerInRange && Input.GetKeyDown(KeyCode.E))
    {
        if (!questAccepted)
        {
            // Accept the quest
            questAccepted = true;
            DisplayMessage("Quest Accepted: Please fetch the item!");
            SetQuestTracker("Current Quest: Fetch the item. It is located outside in an opening.");
        }
        else if (questAccepted && questComplete)
        {
            // Complete the quest
            DisplayMessage("Quest Complete: You returned the item!");
            SetQuestTracker("\"Thank you traveller!\"");
            questAccepted = false; // Disable future interactions

            Invoke("ClearMessage", 2f);
            Invoke("ClearQuestTracker", 3f);


        }
    }
}
```

Pretty much what we are doing here is adding some if and else statements to decide whether the player has accepted the quest or whether the player has completed the quest. If the player is in range of the NPC and presses ‘E’ the questAccepted bool switched to true and two messages will be displayed for the player to see. These messages will appear in those text boxes we set up earlier and you can change the text to whatever you would like and would fit into your game. I

went with some pretty common and basic text. But if the player has accepted the quest and the quest is complete (both bools must be true) then it will display the quest complete messages and dialogue from the NPC thanking the player. Setting the questAccepted back to false after this is to ensure that no further interactions can be made.

The two invoke lines under this is to control how long these messages stay on the screen. The message “Quest Complete: You returned the item!” will stay on the screen for 2f (2seconds) and the quest tracker message will remain for 3f. You can change these values to whatever you think would work best.

Now under the void update() section we will add another private method that will provide prompts and dialogue for the player when in range of the NPC.

```
Unity Message | 0 references
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player"))
    {
        playerInRange = true;

        if (questComplete)
        {
            DisplayMessage("Press 'E' to complete the quest.");
            SetQuestTracker("");
        }
        else if (!questAccepted)
        {
            DisplayMessage("Press 'E' to accept the quest.");
            SetQuestTracker("\"Hello traveller! Would you mind finding an Item I have lost somewhere outside?\"");
        }
    }
}
```

What this method does is checks if the player is inside the collider by using a tag. Before we continue let’s make sure we have a ‘Player’ tag on our Player character in Unity. To do this head back to Unity select your Player in the Hierarchy and in the Inspector at the top right it should say ‘Tag’ make sure this tag is set to ‘Player’ if there is no such tag you can create one by clicking the drop down menu and selecting ‘Add tag’ at the bottom. Make sure you name the new tag “Player” as in our script we already referenced this tag.

So, if the player is inside of the collider and the quest is complete then it will prompt the player with instructions on how to complete the quest. In my case it is by Pressing E, it also sets the quest tracker to an empty string as now there is no more quests active. However, if the quest isn’t complete it will prompt the player to

press E to accept the quest and also sets the quest tracker to be the dialogue of the NPC. You can simply change the string to whatever you want to make the NPC say whatever would fit best for your quest. If you want to add speech marks to your dialogue however, you must add \ before introducing them as that's the way C# differentiates between speech marks to determine a string and actual speech marks that will be displayed inside the string.

We are now almost done with our NPC script!

To finish our code, we finally need to add another method this time to determine that the player is not inside of the NPC's collider. We can do this like so:

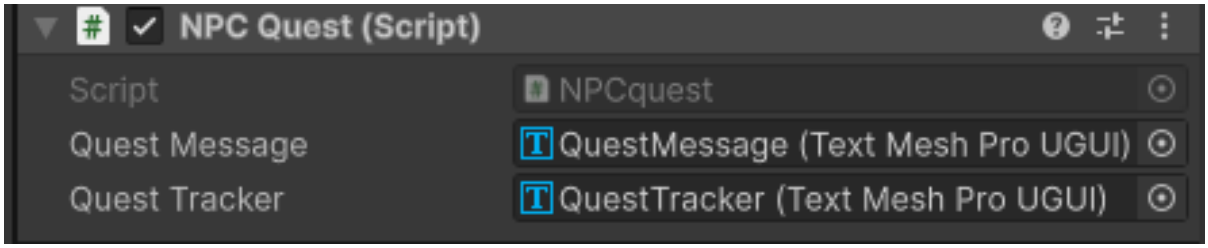
```
Unity Message | 0 references
private void OnTriggerExit(Collider other)
{
    if (other.CompareTag("Player"))
    {
        playerInRange = false;

        if (!questComplete)
        {
            ClearMessage();
        }
    }
}
```

This will update the playerInRange bool to false as the player is now outside of the collider "onTriggerExit(Collider other) and also clear the questMessage so that it isn't on the screen when the player moves in range of the collider but then back out again. Therefore you could walk in and out of the collider and the message will appear and disappear as needed.

It might seem redundant to have "playerInRange" in this script as we already use OnTriggerExit and OnTriggerEnter however it can be useful for the future if you were to add sound/visual cues or anything of the sort to your quest. If you were to add a sound cue for example you could simply use playerInRange = true/false in an if statement to then play the sound cue accordingly instead of typing out private void OnTriggerExit/Enter every time.

Okay awesome! We have now finished coding our 'NPC'. You can indeed test that this works by pressing play however before you do that we must plug in a couple

things in the inspector tab. Select your NPC and navigate to where the script is in the Inspector, should be near the bottom, then we must plug in the QuestMessage and QuestTracker from our hierarchy (if you cannot see them make sure you click the arrow next to “UI”) into their designated slots under our script like so:



Now save everything press play and you can test your NPC script. You should be able to interact with the NPC to a certain point until you are required to pick up the item. Of course, we haven’t coded that in yet which is what we will move on to next!

Okay to begin with our item script let’s create a new script by clicking on our item going to the inspector and creating a new script. Let’s call this “FetchItem”.

Now once we open this script let’s once again delete all the green comments as we do not need these.

Next:

```
using UnityEngine;
using TMPro;
using System.Collections;
```

At the top of your script make sure once again that we are using TMPro; this time we also want to add System.Collections; as this will enable us to use a coroutine method that I will show you later in the script.

Now just like before let’s begin with declaring our variables.

```
Unity Script (1 asset reference) | 0 references
public class FetchItem : MonoBehaviour
{
    public TextMeshProUGUI questMessage;
    private bool playerInRange = false;
    private bool isCollected = false;
    public NPCQuest npcQuest;
```

This time for our variables we only need the questMessage as we will not be using this script to update our questTracker whatsoever. That is being done in the previous script.

We also want to add a isCollected bool so we can determine if the item has been collected or not.

And a public reference to our NPCquest script (the script we just worked on) as this will be how we can make our two scripts work together.

Since a lot of what I will be explaining is similar to what we did in our NPC script I will try to summarise what I can so I do not repeat myself. Let's move on to adding some important methods. We will add these once again under the curly brackets of our "void Update();"

```
2 references
private void DisplayMessage(string message)
{
    if (questMessage != null)

    {
        questMessage.text = message;

    }

}

2 references
private void ClearQuestMessage()
{
    if (questMessage != null)
    {
        questMessage.text = "";

    }


}
```

We have these two methods here which aid in displaying our quest messages and also clearing them off our screen.

```
Unity Message | 0 references
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player") && !isCollected)
    {
        playerInRange = true;

        if (npcQuest.IsQuestAccepted())

        DisplayMessage("Press 'E' to pickup.");


    }

}
Unity Message | 0 references
private void OnTriggerExit(Collider other)
{
    if (other.CompareTag("Player") && !isCollected)
    {

        playerInRange = false;

        ClearQuestMessage();
    }

}
```

Just above those we also have these two methods that determine if the player is in the collider of the item or not. As you can see we actually reference our previous script here in the line “if (npcQuest.IsQuestAccepted())” so that it can check if the quest has been accepted. If it is then prompt the player to pick the item up.

In the OnTriggerExit however, if the player isn’t in range clear any quest messages. This is so the prompt doesn’t remain on your screen if you decide to leave the collider before picking the item up.

Next let’s move up to our void Update method.

In here we will allow the use of the ‘E’ key to allow us to pick the item up. Because we do not have an inventory or a way to actually store this item we will create an illusion that the player has picked the item up by hiding it from view when the player presses ‘E’ while standing in its collider. This is how we do this:

```
void Update()
{
    if (playerInRange && Input.GetKeyDown(KeyCode.E) && !isCollected && npcQuest.IsQuestAccepted())

    {
        isCollected = true;


        Renderer[] renderers = GetComponentsInChildren<Renderer>();
        foreach (Renderer renderer in renderers)
        {
            renderer.enabled = false;  // Hide the mesh renderer
        }

        Collider[] colliders = GetComponentsInChildren<Collider>();
        foreach (Collider collider in colliders)
        {
            collider.enabled = false;  // Disable the collider so player can't interact anymore
        }


        DisplayMessage("Item collected! Now return to NPC.");

        if (npcQuest != null)
        {
            npcQuest.MarkQuestComplete();

        }

        StartCoroutine(ClearMessageAfterDelay(2.4f));
    }
}
```

So if the player is in range and they press ‘E’ on their keyboard AND the quest is accepted in our npcQuest script then the bool: isCollected becomes true. Also the mesh and its collider are hidden so that the player can’t see the item anymore and can’t interact with its collider. It also then displays the message “Item collected! Now return to NPC,”. We must hide the mesh and collider instead of simply destroying it as by destroying it then our script cannot interact with our TextMeshPro anymore so the following message “Item collected….” Wouldn’t appear. Hiding it instead of destroying it means that the item’s components can still be accessed when needed.

It then marks our quest as complete since the player just has to go back and speak to the NPC again.

As I mentioned earlier we use a coroutine in this script. We use this coroutine to clear the item collected message after 2.4 seconds. You might be wondering why

not just use ‘Invoke’ instead like we did before? The truth is that I am not entirely sure because I have first tried using ‘Invoke’ to clear the message but nothing would happen no matter what I tried the message still stayed on the screen after 2.4 seconds, so I tried another way of clearing it and this one seemed to work. Do not worry though as using invoke or a coroutine does pretty much the same thing so we can easily get away with using a coroutine in this case. To make sure the coroutine works we also have to add this line at the bottom of our script.

```
    1 reference
    private IEnumerator ClearMessageAfterDelay(float delay)
    {
        yield return new WaitForSeconds(delay);
        ClearQuestMessage();
    }

}
```

Now we have pretty much everything written out and set up for our quest to be fully functional!

All that is left is plugging our QuestMessage and our NPCquest script into our Item’s script in the inspector like so

You now should have a working fetch quest for your game! Of course, replace any placeholder models with anything you’d like such as real character models or fitting items like maybe a lost weapon or tool. You can customise most things as you wish.

Thank you and good luck!

Pastebins:

Script1 - https://pastebin.com/pEpa6DLw

Script2 - https://pastebin.com/uRH8N1L0