**Game Programming Tutorials – Inayah Iqbal**

Below you will find 4 comprehensive tutorials on making a variety of key features in Unity.

Contents

* 3D walk Movement
* First Person Camera movement
* Jump Movement
* Trigger events (using trigger box and UI)

Sources

- <https://youtu.be/f473C43s8nE?si=dlEs0UjiPCOrJkGP>

- <https://youtu.be/MXGEQU3pHlI?si=W79p-_yXpHljJC26>

- https://youtu.be/-Rs7gfNKAm0?si=wmlIAguwwsxSuk4v

**How to make 3D Walk movement with force**

To outline the system, the player will move based on the direction the camera is facing. When moving forward the player character will have a large force which makes them move very quickly, kind of like super speed.

**Step 1:** First, we’ll make a new script called ‘PlayerMovement’ and define the variables we’ll be using in the public class:

public class PlayerMovement : MonoBehaviour

{

public float moveSpeed;

public Transform orientation;

float horizontalInput;

float verticalInput;

Vector3 moveDirection;

Rigidbody rb;

Then make a new void called private void MyInput() where we’ll get the players input:

private void MyInput()

{

horizontalInput = Input.GetAxisRaw("Horizontal");

verticalInput = Input.GetAxisRaw("Vertical");

}

Make sure to run MyInput() on void Update() so that we’re always checking the players input

private void Update()

{

MyInput();

}

**Step 2:** Next we’ll code the actual movement by using the players input and multiplying it to create the movement. We’ll also code the force, which you can increase, or decrease based on what value you input:

private void MovePlayer()

{

moveDirection = orientation.forward \* verticalInput + orientation.right \* horizontalInput;

rb.AddForce(moveDirection.normalized \* moveSpeed \* 10f, ForceMode.Force);

}

Then lastly, we’ll check if the rigid body is moving in voide Start():

private void Start()

{

rb = GetComponent<Rigidbody>();

rb.freezeRotation = true;

}

Finally add this script to the ‘Player’ empty containing the Rigid Body and set the MoveSpeed to ‘7’ and drag the ‘orientation’ into the orientation variable. And we’re done! Test it out and have fun.

**How to make First Person camera movement**

**Step 1:** The basics of this system requires the camera to move along the ‘X’ and ‘Y’ axis using the players mouse. The first thing to do is create a new script called ‘PlayerCamera’.

Inside public class PlayerCamera() we’ll need to create some variables such as:

public float senseX;

public float senseY;

public Transform orientation;

float xRotation;

float yRotation;

**Step 2:** Next in void Start() we’ll need to make sure our mouse is locked and not visible when playing the game just to make it neater when we are playing our game. So simply type this within void start:

void Start()

{

Cursor.lockState = CursorLockMode.Locked;

Cursor.visible = false;

}

**Step 3:** Next in void Update() we’ll do the actual coding for the functionality, how it works is we’ll be getting the players mouse ‘X’ axis and multiplying it to create the camera movement effect we would like. So, within void Update() write this:

void Update()

{

float mouseX = Input.GetAxisRaw("Mouse X") \* Time.deltaTime \* senseX;

float mouseY = Input.GetAxisRaw("Mouse Y") \* Time.deltaTime \* senseY;

yRotation += mouseX;

xRotation -= mouseY;

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

transform.rotation = Quaternion.Euler(xRotation, yRotation, 0);

orientation.rotation = Quaternion.Euler(xRotation, yRotation, 0);

}

On the 5th line you’ll notice the code xRotation = Mathf.Clamp(xRotation, -90, 90f);

This code is very important as it stops the player from rotating the camera over 90 degrees into an awkward position. After to implement this, we must use Quaternion.Euler, which is Unity’s way of handling rotations.

**Step 4:** Now we’ll set up our player. We’re first going to add an ‘empty’ into our scene and rename it to ‘Player’ we’ll set its rigid body to interpolate and collision detection to constant.

Now you can add in your actual player object, which in this example we will just be using any shape of your choice. We’ll want to parent it to the ‘Player’ empty we made earlier.

Finally we’ll create 2 more empty objects that will help us set the position of ‘Orientation’ and ‘CameraPosition’ we’ll move the origin of both these objects to wherever we’d like the camera to be. Make sure to parent these as well to the ‘Player’ empty. Note: we should parent everything to keep everything in one place and obviously so they will all move together.

**Step 5:** Now we’ll set up the camera. Instead of parenting the PlayerCamera to the player because that can be buggy sometimes, we’ll parent the camera into an another ‘empty’ called ‘camera holder’, to make sure it follows the player still we’ll just make a simple script called ‘cameraFollowPlayer’ which just includes this code:

public class CameraFollowPlayer : MonoBehaviour

{

public Transform cameraPosition;

void Update()

{

transform.position = cameraPosition.position;

}

Don’t forget to assign this script to the camera holder.

**Step 6:** Finally assign the ‘PlayerCamera’ script we made earlier to the ‘camera position’ empty, adjust the values of SensX and SensY (which just adjusts how sensitive the rotation is) to values like ‘400’ and drag the orientation empty into the orientation and you should be done. You can adjust the rotation limits more if you would like them more or less than 90 degrees as well as adjust the sensitivity easily as well.

**How to create 3D Jump movement**

First we’ll create a new script called ‘Jump’, the code itself is quite simple, the premise is that we’ll be using a raycast to see how far off the ground the player is. We’ll define some variables and do the following:

public class Jump : MonoBehaviour

{

public float jumpForce = 10;

public float groundDistance = 1f;

Rigidbody rigidBody;

void Awake()

{

rigidBody = GetComponent<Rigidbody>();

}

bool IsGrounded()

{

return Physics.Raycast(transform.position, Vector3.down, groundDistance);

}

void Update()

{

if (Input.GetMouseButtonDown(0) && IsGrounded())

{

rigidBody.velocity = Vector3.up \* jumpForce;

}

}

Here we are simply checking if the mouse button is pressed down to start the jump in void update(). We also do some math, adding the jump force to make us move up vertically. Lastly add this script to the player empty and you may adjust the values of Jump Force and ground distance in the insepctor to whatever suits your game the most, and we should be done.

**How to trigger events using trigger boxes (using UI)**

In this system we’ll trigger an event, in this example will be to show a UI element when the player overlaps with a trigger box. We’ll first add a cube and turn off the mesh renderer so we can only see its outline, this will also allow the player to walk through it as well. Alos set the trigger to ‘IsTrigger’.

Next, we’ll set up the UI elements, we’ll add a canvas to the scene and then add some text using ‘TextMeshPro’ make sure this is parented to the canvas, you can edit the text within the inspector to change its size and color.

Next we’ll create a new script, for this example we’ll call it ‘ShowUI’ and do the following

public class ShowUI : MonoBehaviour

{

public GameObject uiObject;

void Start()

{

uiObject.SetActive(false);

}

void OnTriggerEnter(Collider player)

{

if (player.gameObject.tag == "Player")

{

uiObject.SetActive(true);

}

}

}

To put it simply all we’re doing is on void Start() we’re making sure the text is hidden, then creating a new event ‘OnTriggerEnter’ to set the UI text to active so that it can be seen. Finally add this script to the trigger box and we should be done.