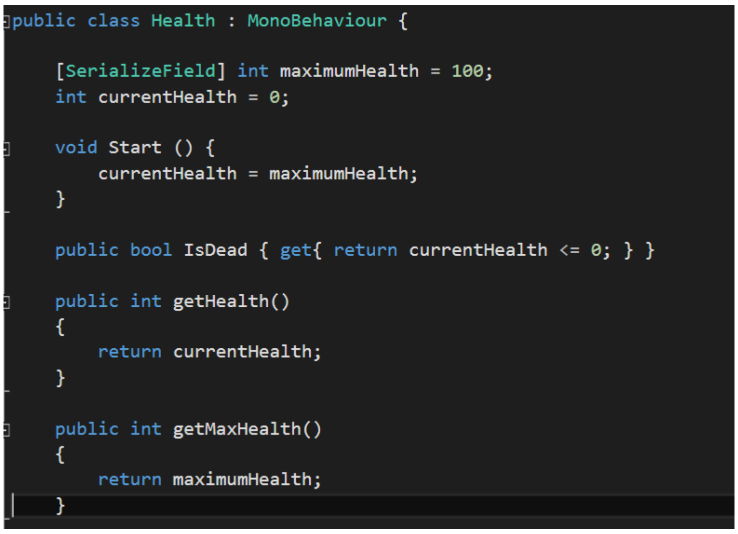
Damaging, Attacking and Interaction

In this tutorial we'll go through some ways to add damage, health and interaction to our scene, as always this isn't the only way, but it's the way I will show you. We are using the base test scene with it's primitives for this, if you haven't already you should do the "setting up a basic scene" tutorial.

Health scripts

A health script at it's most basic needs to be able to do a few key things, store the health, retrieve the health, and modify the health. We set the health script up as a new script called Health, with a serializedfield called maximum health, so we can adjust the maximum health, then another variable , private this time, called currentHealth. We set their default values to 100 and 0, although we can override maximum health in the inspector. This is what it looks like;



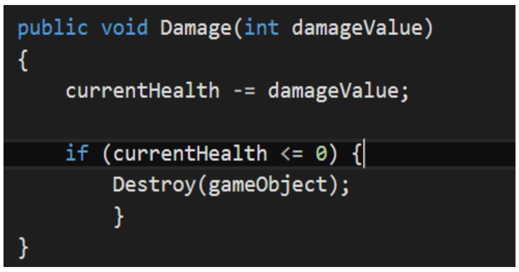
```
public class Health : MonoBehaviour {

    [SerializeField] int maximumHealth = 100;
    int currentHealth = 0;

    void Start () {
        currentHealth = maximumHealth;
    }

    public bool IsDead { get{ return currentHealth <= 0; } }

    public int getHealth()
    {
        return currentHealth;
    }

    public int getMaxHealth()
    {
        return maximumHealth;
    }
```

In Start() we set the currentHealth to be equal to whatever the maximumHealth is, thus giving the player full health.

Below, we make sure of a C# feature called properties, this let's us specify exactly what is returned, we use the get property accessor to signify that we are retrieving the value of the code. So, in the case we can use "IsDead()" to get the result of the code "currentHealth <=0". Which is more like a question, so it gets whether the current health is less than or equal to 0. Returning true if it is, false if it's not.

We then have to public methods to get the current health and max health, we use public methods to get these rather than public variables as public variables can be accessed in lots of ways and values changed, this way you can only read the values and then can change them which brings us to the damage.

Damage is it's own function, public so it can be used by other scripts, taking a value for the damage amount. Finally, subtracting that value from the health value and checking to see if that thing is alive or not rounds off the function's purpose.



```
public void Damage(int damageValue)
{
    currentHealth -= damageValue;

    if (currentHealth <= 0) {
        Destroy(gameObject);
        }
}
```

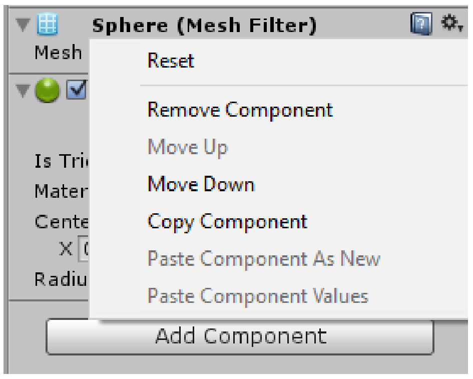
Attacking

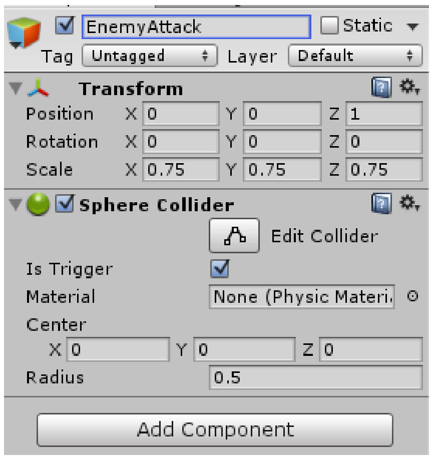
Now we can damage and have health we can take it away, for as long as man has had health, man has tried things that take away that health, this is no exception. Here I'll show you two methods of attacking, the raycast and the melee hit.

Melee

Melee boils down to one thing, when a player is close enough, subtract, we do this using colliders and triggers. We make an area using a trigger or collider and when it's entered we damage the player, this method is called trigger zones and is a fairly simple way of doing things. It's got many uses, one of which is attacking.

So, on our enemies, we need to create a trigger zone for them in front. If you remember, we were using the blue spheres as "enemies" select one and right click on it, then 3D object->Sphere. We're going to use this secondary sphere as our trigger zone, now, you might say you don't want to see a big off white sphere, we'll deal with that but first, as we can see it nicely, resize it and place it where you want. (Note: the Z axis is the front, if you move it to 0,0,1 it should be 1 unit in front of the enemy). So, now we have the sphere positioned, but we don't want to SEE where we're about to be hit. What we can do is take advantage of Unity's component based design and delete the components we don't want. In this case, it's the Mesh Renderer and Mesh Filter. Whilst we're doing this we can also add the Sphere Collider, very important. To do this click the little cog in the top right of the components area and then remove component, do this for both the renderer and filter and add in the sphere collider using add component. Finally, make the collider a trigger, this means it detects collisions but doesn't actually use physics. It can see if something bumps up against it, but can't stop it, it's not "physical".




EnemyAttack Static
Tag Untagged Layer Default

Transform
Position X 0 Y 0 Z 1
Rotation X 0 Y 0 Z 0
Scale X 0.75 Y 0.75 Z 0.75

Sphere Collider
Edit Collider
Is Trigger ✓
Material None (Physic Materi
Center
X 0 Y 0 Z 0
Radius 0.5

Add Component

Now we need to create a new script, call it "EnemyAttack" we make use of the events built into Unity for handling when something goes inside a trigger. There's a few of these, the one we will be using is OnTriggerStay – we want to react and keep reacting for as long as something is inside the trigger. We could literally say "OnTriggerStay- do some damage" but there's a few problems with that, one is that OnTriggerStay runs every frame, damaging every frame is bad for lots of reasons, what happens if frame rate drops for one? But mostly it's imprecise, it could be 30 damage a second, it could be 90. Instead we make it so damage is applied every x seconds.

```
float nextTimeAttackIsAllowed = -1.0f;

[SerializeField] float attackDelay = 1.0f;

[SerializeField] int damageDealt =5;

void OnTriggerStay(Collider other)
{
    if (other.tag == "Player"&&Time.time>=nextTimeAttackIsAllowed) {
        Health playerHealth = other.GetComponent<Health>();
        playerHealth.Damage(damageDealt);
        nextTimeAttackIsAllowed = Time.time + attackDelay;
    }
}
```

Variables are set to handle how long between attacks and the damage dealt. Remember, SerializeField means it's visible in inspector and OnTriggerStay is the function we spoke about earlier, it's called every frame an object is in the trigger.

The inside is the bit that might be a bit confusing but line by line it's not too bad, first, we check if other.tag is equal to player. "other" means the collider that is attached to the thing that is inside the trigger we can see it referred to in the parameters, we can access all the information we want about it, but that's just a quick check. We also want the Time.time (the time that the games been running for) to be equal to or greater than the time at which the next attack is allowed.

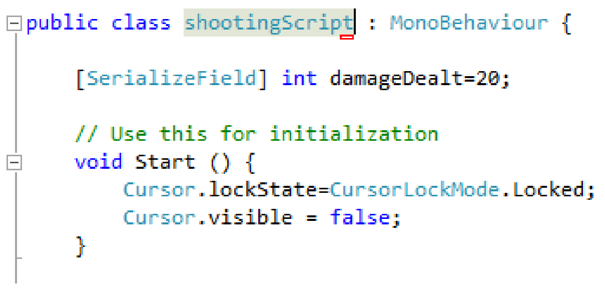
If all this is true, we get the playerHealth script (the one we wrote up above) by using GetComponent on the thing that hit us. Then, we call the Damage function we wrote above too. Finally, we update the time at which we can attack next. Neat and simple.

Raycast

A raycast sounds perhaps more complicated than it is, essentially it is an invisible ray sent out from a point to another point. This is used quite often in games, it's cheap (in terms of resources, system processing power, etc.) and easy, the ray can report back what it hit, how far, in fact they behave largely like colliders and triggers. The nice thing about Raycasts is that they are more or less instant, we can use them for lots of things, in this instance we're just faking the bullet's movement as an instant line. In videogames, this is known as "hitscan" and has it's own pros and cons.

We are going to use a RayCast to allow player attack but there's no reason that you can't use it for NPCs or even static objects. On the player, create a new script called "shootingScript" imaginative no?

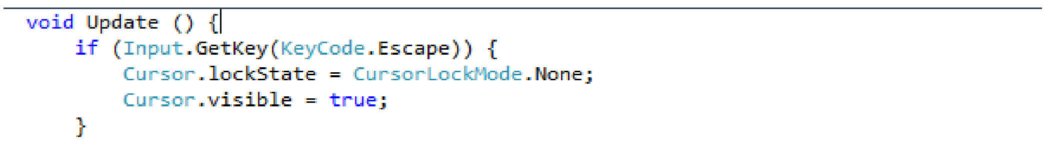
Here's what goes in it;



```
public class shootingScript : MonoBehaviour {

    [SerializeField] int damageDealt=20;

    // Use this for initialization
    void Start () {
        Cursor.lockState=CursorLockMode.Locked;
        Cursor.visible = false;
    }
```

We include a serialize field so that the damageDealt variable can be changed in the editor. We set it's default to 20.

Then, in start we set up the cursor to hide when you start the game. We want it hidden and locked into middle of the screen so it doesn't interfere with our view.



```
void Update () {
    if (Input.GetKey(KeyCode.Escape)) {
        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;
    }
```

Now, in Update we check each turn to see if the escape key is pressed, here you can see a basic if statement which is used for checking values and conditions. We use the Input class (the thing that represents all inputs), to access the GetKey method and then pass it KeyCode.Escape. All this does is checks if escape is pressed, if it is we just do whatever is between the curly brackets, Unity doesn't know what key escape is so we pass it "Keycode." And that has the codes for all the keys we could want. We want to unlock the cursor for menus and changing settings just to make it a little easier. We just reverse what was put in earlier.

Staying in update, underneath this we want to check if fire is pressed, in this case fire is a preset and configured button in Unity. Keys and buttons are different, keys are what you expect, keys on keyboards or controllers, buttons are preconfigured, along with axis they represent a variety of actions that you might want to do. You can see all these in Edit->Project Settings->Input.

```
        if (Input.GetButtonDown ("Fire1")) {
            //Make a raycast with the line starting from center of camera
            Cursor.lockState = CursorLockMode.Locked;
            Cursor.visible = false;
            Ray mouseRay = GetComponentInChildren<Camera>().ViewportPointToRay(new Vector3(0.5f, 0.5f, 0));
            RaycastHit hitInfo;
            //Send the raycast and if the raycast hit something, print out the name to console
            if (Physics.Raycast (mouseRay, out hitInfo)) {
                print(hitInfo.transform.name);
                Debug.DrawLine(transform.position, hitInfo.point,Color.red,5.0f);
                Health enemyHealth = hitInfo.transform.GetComponent<Health>();
                if(enemyHealth !=null)
                {
                    enemyHealth.Damage(damageDealt);
                }
            }
        }
    }
```

This goes directly underneath the first if statement, it checks in a similar way if a fire button is pressed, if it is we lock the cursor similar to above in the first couple of lines then we begin to construct our raycast.

The third line, starting “Ray mouseRay” constructs our ray that will trace the path. We grab the Camera attached to any of the child objects using GetComponentInChildren. We use ViewportPointToRay to convert a point in screen references to draw a ray from that point give it a new Vector3 with 0.5f, 0.5f, 0 that is half way across the screen, half way up the screen and at the exact spot the camera is. That’s where the ray will come from, right in the middle of the screen.

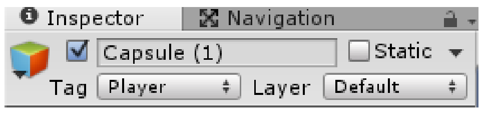
The next line creates a local variable that will hold the results of the raycast (which object it hit, how far it was, etc.). RaycastHit is a struct, this means it’s a group of related variables that be thought of as a set.

Finally, we do the actual raycast, we put it into an if statement so that only if the raycast hit something (and thus returns true) does the next bit run. The “out” keyword is used by C# to indicate that the variable passed (hitInfo) should have its contents changed.

Inside the if statement, if it’s hit something we first do some debugging to show how it works (and if it works). We print out the name of what we hit and draw a line from the character to whatever we hit, this lets us keep track. Then we get down to the business of hitting stuff. We try and get the Health script off whatever we hit using hitInfo.transform (remember, hitInfo contains all sorts of details about what we hit). We then check to make sure we did get a health script and if we did, we do some damage. Simple, mostly.

Attach this script to the Player if it’s not already, now to test it and the meleeing damage.

Setting up and testing damage

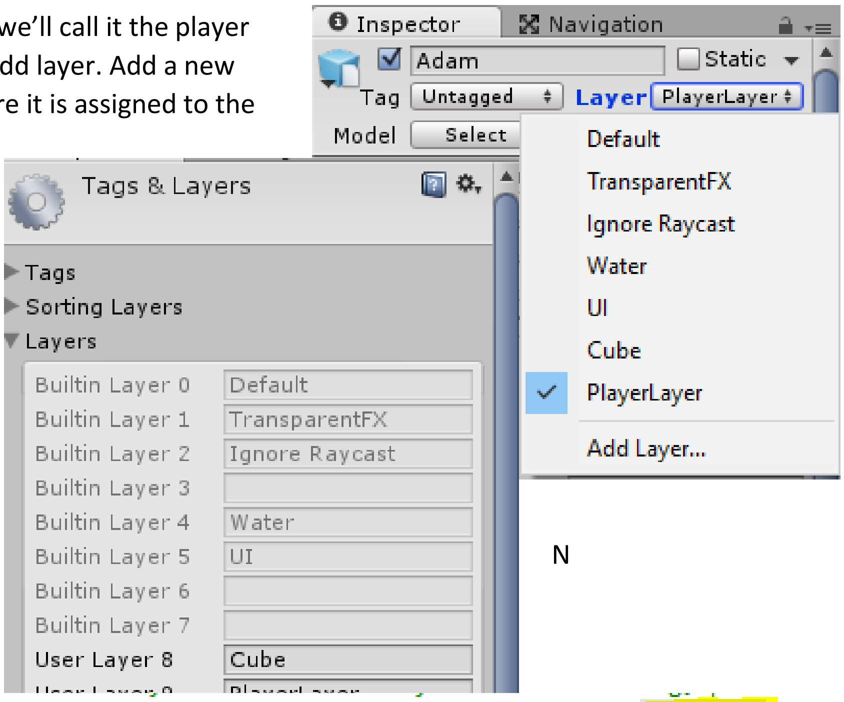


Capsule (1) Static
Tag Player Layer Default

You need an enemy, with the appropriate health script attached and the enemy attack configured as described. We then add a health script to the player and ensure the Player is tagged as “Player” (select the Player capsule and it’s as shown in the screenshot). Now try it out, move your player until he gets into the circular attack bit, if you stay there long enough he should die. You may be finding this doesn’t happen, what’s actually happening is that the raycast is hitting the “EnemyAttack” collider, there’s an easy fix for this, select the enemyAttack object and set it’s layer in the top right to “ignore Raycast”.

It may be that you have an issue with damaging via raycasts, the problem is that the position of the camera means the raycast shoots through the player that shoots and damages them. It's a fairly easy fix to execute but requires a few changes.

The first is the addition of a new layer, we'll call it the player layer, click on your PLAYER, and go to add layer. Add a new layer called "PlayerLayer" and make sure it is assigned to the player.


Inspector | Navigation

Adam | Static

Tag: Untagged | Layer: PlayerLayer

Model: Select

- Default
- TransparentFX
- Ignore Raycast
- Water
- UI
- Cube
- ✓ PlayerLayer
- Add Layer...

Tags & Layers

- Tags
- Sorting Layers
- Layers

Builtin Layer 0	Default
Builtin Layer 1	TransparentFX
Builtin Layer 2	Ignore Raycast
Builtin Layer 3	
Builtin Layer 4	Water
Builtin Layer 5	UI
Builtin Layer 6	
Builtin Layer 7	
User Layer 8	Cube

The screenshots to the right show this, then there is some change to the shooting script to change these.

So, open it up and add the two bits highlighted in yellow. The first is a new variable which is the layer we want to ignore. We've put the player on that layer so it'll be ignored.

N

```
[SerializeField] int damageDealt=20;
[SerializeField]
LayerMask layermask;
```

```
if (Physics.Raycast (mouseRay, out hitInfo,100,~layermask)) {
    print(hitInfo.transform);
```

The next bit, in the if statement is two small additions. The 100 is the distance the raycast will shoot, but the real difference is using the layermask variable. Note the ~ this inverts the layermask, meaning you will IGNORE all items on that layer, without it, we'll only be able to hit things on that layermask.

But, we also want to be able to be able to ignore the "ignoreRaycast" layer, so, we make one more change; in start, we use |= which, without going into huge detail, adds the ignoreraycast layer to our layermask list. The ~ then inverts it (meaning we only hit the ones not listed), then, in Update we just delete the ~ infront of layermask (as we've done this step in Start()).

```
// Use this for initialization
void Start () {
    anim = GetComponent<Animator>();
    Cursor.lockState=CursorLockMode.Locked;
    Cursor.visible = false;
    layermask |= Physics.IgnoreRaycastLayer;
    layermask = ~layermask;
}
```

```
//Send the raycast and if the raycast hit something check to see if it has health
if (Physics.Raycast (mouseRay, out hitInfo,100,layermask)) {
    Health enemyHealth = hitInfo.transform.GetComponent<Health>();
    if(enemyHealth !=null)
    {
        enemyHealth.Damage(damageDealt);
    }
}
```

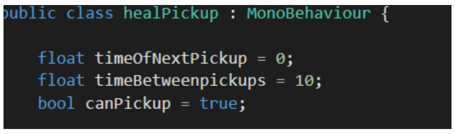
Interaction

At it's basis you can use the same methods to interact, this is interacting in a way, except you interact and things go bang. I'll show you two examples of how you can use these methods to interact with things. The first, a basic collect and pickup script, we'll make an object that can be used as a collectable and when we hit it, it'll disappear and we get some health back.

Create a new cube, shrink it down and give it a new material colour, something that makes it stand out. On the attached collider click the is trigger property, a trigger doesn't provide any "physical" boundary, we can walk through it but it still generates an event.

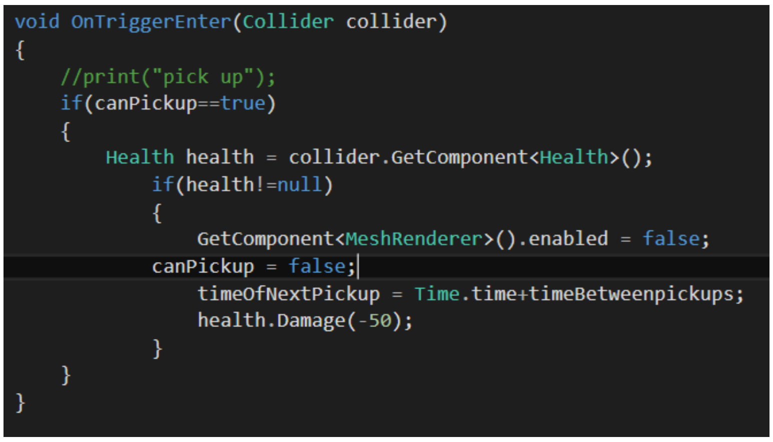
Create a new script, call it "heal Pickup", similar to the enemy attack we use a built in function, this time "OnTriggerEnter" which takes a parameter of which Collider was hit.

It looks like this;



```
public class healPickup : MonoBehaviour {

    float timeOfNextPickup = 0;
    float timeBetweenpickups = 10;
    bool canPickup = true;
```

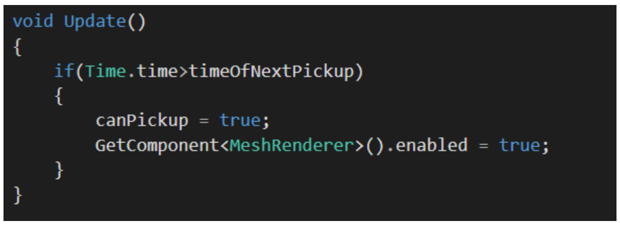
We first set a few variables, a float for the time we can next pick this up and the time between pickups (to stop people standing on it to recover health) and a Boolean to check if it's able to be picked up.



```
void OnTriggerEnter(Collider collider)
{
    //print("pick up");
    if(canPickup==true)
    {
        Health health = collider.GetComponent<Health>();
            if(health!=null)
            {
                GetComponent<MeshRenderer>().enabled = false;
            canPickup = false;
                timeOfNextPickup = Time.time+timeBetweenpickups;
                health.Damage(-50);
            }
    }
}
```

We check to see if we are able to pick this up, then we get the health script from whatever we've hit and store it. If we've been able to grab a health script then we disable the meshrenderer making it invisible and unable to be picked up before setting the time of next pickup to be now + time between pickups.

Finall, this time we "damage" it -50, effectively healing 50. You can do all sorts of things here, pick things up, open doors, trigger speech, the limit is imagination.



```
void Update()
{
    if(Time.time>timeOfNextPickup)
    {
        canPickup = true;
        GetComponent<MeshRenderer>().enabled = true;
    }
}
```

Now, in update we check to see if enough time has passed and if so update "canPickup" and make the pack visible again.

Put the script on the pickup item we created earlier and try it.

Raycast interaction

Raycast interaction is similar too, we'll show two examples, one where the ray hits a door and if a button is pressed we open, one where a camera changes when you approach. Create another new box, scale it (0.5,4,4) to make it a more door like shape and maybe change the colour to something appropriate and clearer.

So, we need two scripts for this one, one on the door, one on the player. The one on the player sends a signal to the one on the door which then does all the work. Let's start with that first, create a new script called "DoorOpenScript" and attach it to our door. There's loads of different ways we could do this but we'll keep it simple, using another of Unity's built in functions, "OnEnable and

OnDisable" these both carry out the actions set when the script is either enabled or disabled respectively. It looks like this;

```
void OnEnable()
{
    this.transform.position =
        new Vector3(transform.position.x, transform.position.y + 3, transform.position.z);
}
```

inside OnEnable we set the transform position of the thing it's attached to to be equal to a new Vector3 which has the values of the current position plus 3 more in the y direction. This effectively just makes the door rise by 3. I'll leave you to do the same for "void OnDisable()" it's exactly the same but instead you minus 3 to make it go down.

If you now look at your door in the inspector and disable the door script by clicking the little tick next to it, then press play, what you should see is that when you click the tick again whilst it's playing the door goes up. Untick it and the door goes down. Try it.

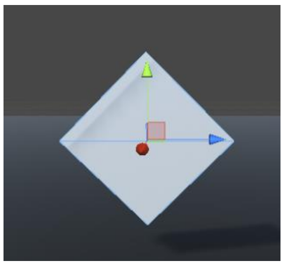
Finally, we need the script for the player, create a new script called InteractionScript, here, in Update we're going to add a button that when pressed near to a door will open it. This is very similar to the weapon above, the script is below, see if you can follow it.

```
public class InteractionScript : MonoBehaviour
{
    // Update is called once per frame
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.F))
        {
            Ray mouseRay = GetComponentInChildren<Camera>().ViewportPointToRay(new Vector3(0.5f, 0.5f, 0));
            RaycastHit hitInfo;
            if (Physics.Raycast(mouseRay, out hitInfo))
            {
                DoorOpenScript door = hitInfo.transform.GetComponent<DoorOpenScript>();
                if (door)
                {
                    door.enabled = true;
                }
            }
        }
    }
}
```

Changing Camera view

Now for the last item, a raycast for a switch to change camera, for something different we're going to make this raycast from an inanimate object.

Add a new Camera to the scene and position it at 0,20,0 with rotation 90,0,0.



Add a cube to the scene, position it at (-2, , 1, 0) and rotate it (45,0,0) finally scale it down to 0.2,1,1. This should give you something like this.

We'll make the Raycast come out in the X axis, in the middle, right about where that red line is. When the player moves into it, it'll switch to the new camera, when a player moves out, It'll switch back.

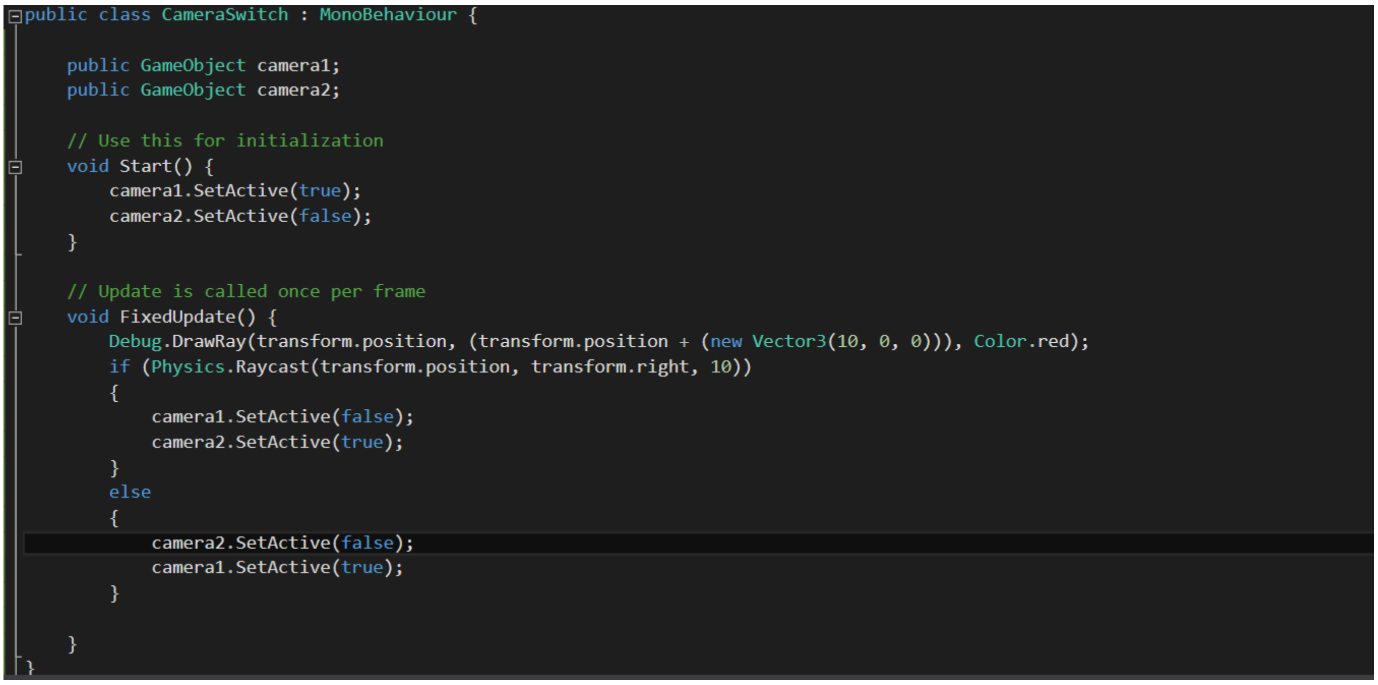
Create a new script “CameraSwitch” at the top, create two variables, camera1 and camera2 of type GameObject. This will be our cameras, remember, in Unity everything is a GameObject if it’s instantiated, they’re just collections of components, one of which is the Camera.

In start, using SetActive (camera1.SetActive(true) for example) turn camera 1 on and camera 2 off. Camera 1 will be our main camera attached to the player, camera 2 the camera looking down.

If you’re not sure the code is posted below, try it before hand though.

Finally, we need to actually make the logic, we do this using a slightly modified Update called FixedUpdate(). FixedUpdate doesn’t run every frame but runs a set number of times per second, which is run depending on how many steps are set in time settings.

The code is below, including what is needed for above.



```
public class CameraSwitch : MonoBehaviour {

    public GameObject camera1;
    public GameObject camera2;

    // Use this for initialization
    void Start() {
        camera1.SetActive(true);
        camera2.SetActive(false);
    }

    // Update is called once per frame
    void FixedUpdate() {
        Debug.DrawRay(transform.position, (transform.position + (new Vector3(10, 0, 0))), Color.red);
        if (Physics.Raycast(transform.position, transform.right, 10))
        {
            camera1.SetActive(false);
            camera2.SetActive(true);
        }
        else
        {
            camera2.SetActive(false);
            camera1.SetActive(true);
        }

    }
}
```

FixedUpdate, the first line is a debug statement, we create a visible ray with the similar properties to the raycast, this is so we can see where the rays are going.

This raycast is a little bit different, we create an if statement, inside we create a Raycast, the first parameter is the starting point of where we start the Ray, in this case, position of what it’s attached to. The next is the direction we want the ray to go to, transform.right which is the direction of the red x arrow and finally the distance we want it go, which is 10.

Now we’ve setup our enemies, some interaction and some fighting, we will add some more features and some more logic, mostly on improving our enemies, but the bulk is done.

Challenges

Challenge 1

Make sure to update all prefabs created previously to use this new attack scripts.

Challenge 2

Create a wall of boxes that can be used to block off a room for the door, use primitives here, you can also use cubes to .

Challenge 3

The Enemy that is faster should have less attack to compensate, reduce it's attack and update it's prefab.

Challenge 4

Create another pickup item, this time instead of healing the player it should damage whoever touches it like a mine. Either player OR enemy.

Challenge 5

Hardest now, modify a new enemy to use a ranged weapon based on the scripts found above, it needs to shoot out a ray from its centre repeatedly, if that ray hits a player, damage it.