Creating Enemies that chase you

As per everything we're doing there are lots of ways of creating enemies, I'm going to show you just a limited number. 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.

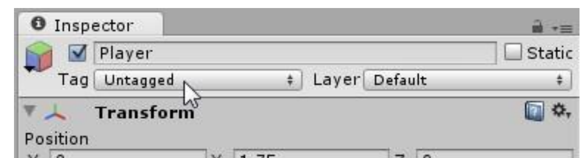
Enemies

We're going to use spheres as the enemies, creepy, huh, create a new sphere and call it "Enemy" if you can't remember how to do this, check out the last tutorial in the series. Create and add a new material called "Blue" and change the enemies to that. Blue spheres huh? Scary. Add a character controller (Component > Physics > Character Controller) to them like the player.

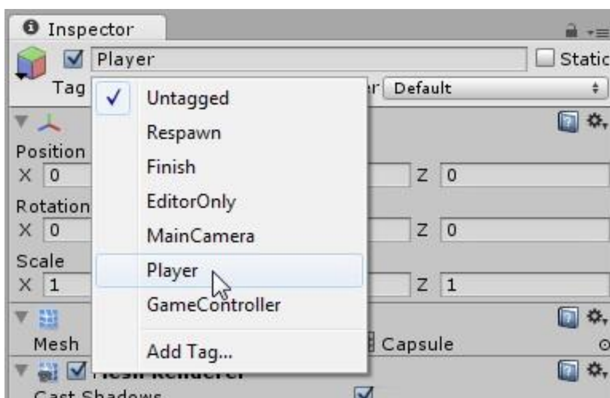
Make a new C# script. Name it "EnemyMovement".

First, before we can make the enemy chase the player, the enemy has to continually know where the player is. It needs a handle, or a reference, to the player's transform (remember Transform holds the position of a game object).

Using Tags



Here we'll make use of what Unity calls "tags"; go to your Unity Editor window. Select the Player game object and in the Inspector, you'll see a property called Tag. The value should be "Untagged" right now.



Click on the "Untagged" and it'll show you a selection of other values to choose from. Among them should be one that says "Player". We'll use that so go ahead and click on it.

So your Player game object should now have the Player tag.

Basic Enemy Movement Script

"playerModel" is a variable that will hold a reference to the Player game object's transform. How will we give it a value? There are a lot of ways to do it. Copy the code into EnemyMovement script, by this point you should be able to follow some of it, consider commenting code to help you better understand it (single line comments begin // multi line begin and end with **).

```
public class EnemyMovement : MonoBehaviour {

    Transform playerModel;

    // Use this for initialization
    void Start () {
        GameObject playerGameObject = GameObject.FindGameObjectWithTag("Player");
        playerModel = playerGameObject.transform;
    }
```

A new area here is FindGameObjectWithTag, this is slow but at the start that's not an issue, it's not something you should use continually throughout the scene. It searches through the scene to find the gameObject with the correct tag.

The next line finally retrieves the transform component of the player game object.

Now we need to add to the script, add the lines for CharacterController, the same way our PlayerMovement script makes use of the character controller component, our EnemyMovement will be using its character controller. Add to your code to get it to the point below;

```
Transform playerModel;
CharacterController controller;

// Use this for initialization
void Start () {
    GameObject playerGameObject = GameObject.FindGameObjectWithTag("Player");
    playerModel = playerGameObject.transform;
    controller = GetComponent<CharacterController>();
}
```

Like the PlayerMovement script, we'll be calling the Move function of our character controller. But the enemy won't be controlled by keyboard input. We'll compute it using a formula instead:

```
Transform playerModel;
CharacterController controller;

// Use this for initialization
void Start () {
    GameObject playerGameObject = GameObject.FindGameObjectWithTag("Player");
    playerModel = playerGameObject.transform;
    controller = GetComponent<CharacterController>();
}

// Update is called once per frame
void Update () {
    Vector3 direction = playerModel.position - transform.position;
    controller.Move(direction * Time.deltaTime);
}
```

So how come this simple formula in line 19 made the enemy follow us? We are actually computing the delta (difference) between the enemy and the player:

$$\Delta x = x_2 - x_1$$

In our case, this would be:

$$direction = destination - source$$

Using this formula gives us the proper direction to move to, in order to chase the player.

Compare line 19 of our code with the formula. "source", in this case, is our own position

(transform.position), and “destination” is the player's position (playerModel.position). These positions are Vector3 values (it has x, y, and z coordinates), so likewise, we need to store the subtraction result in a Vector3.

Challenge 1

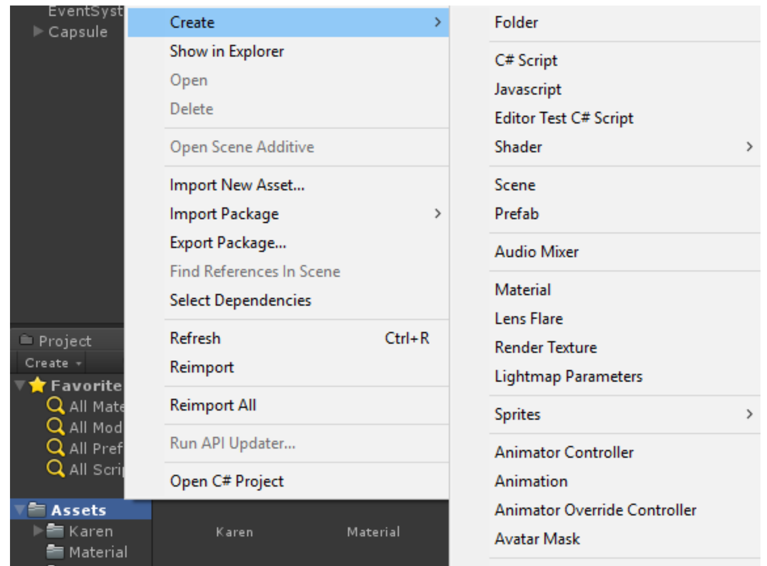
Add a “moveSpeed” variable to the EnemyMovement script, use it similarly to the one in PlayerMovement.

Prefabs

We now have an enemy that chases our player, now how about making more of that enemy?

Unity features an easy way to create templates out of any of your game objects called a “prefab”.

Think of a prefab as a template, or a master copy of one of your game objects. It’s that master copy that you keep on duplicating if you want more copies of it.



Making a Prefab Folder

Prefabs exist in the Project View, first let's make a folder to store all our prefabs, inside the root game folder, make a new folder. Name it “Prefabs”.

Creating A Prefab In Unity

Creating a prefab out of a game object is easy, just drag the “Enemy” game object from the Hierarchy View into the “prefabs” folder in the Project View you just created.
When you drop it there, the “Enemy” prefab will be created. Prefabs are indicated by a blue box icon and the text in the inspector will turn blue.
Think of a prefab instance as a “live” copy, whereas the prefab is the master copy that doesn't exist anywhere in the scene. A prefab's job is only to be duplicated into live copies to the scene.
Prefabs are still the same game objects that you know. You can add or remove components from them. You can move them and edit their values like usual.

Creating Prefab Instances

To create new instances of a prefab, drag the prefab from the Project View into anywhere in the scene; alternatively, you can drag the prefab from the Project View into the Hierarchy View.
Go ahead and drop at least two more enemies onto the scene.

Modifying Prefabs

Now, if we wanted duplicates of our Enemy, actually, we could have just copy-pasted it without bothering with prefabs. Why are we bothering with prefabs then?

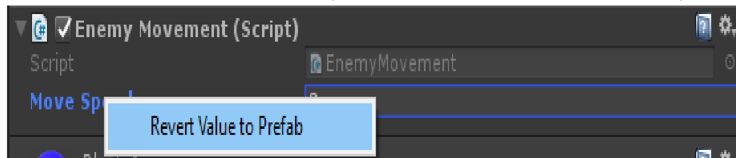
The convenience of a prefab is that you can change its master copy, and all live copies will have the changes propagated to them. (The most important reason you'll want prefabs is that you can create live copies of a prefab on-the-fly during runtime, but we'll tackle that later.)

Now, we'll try modifying the prefab. Click on the Enemy **prefab** in your Project View. The Inspector will show the Enemy prefab's properties, just like a regular game object.

Change the “Move Speed” to 5.25. If you select any of your Enemy game objects in the scene, you'll see it now has 5.25 in its Move Speed also!

Overriding Prefab Values

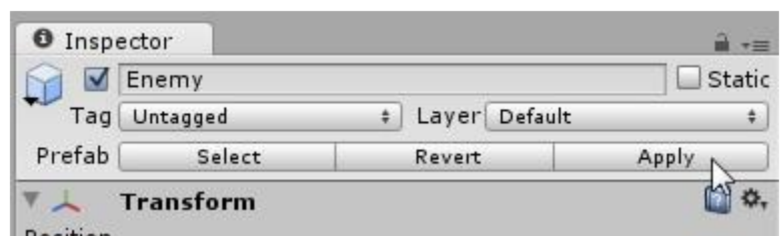
How about if you want only one of the enemies to have a different speed? You can still edit each Enemy game object individually. When you modify only one of them, the value will show in bold letters. That indicates that you've overridden its default prefab value.



If you ever want to revert a property back to its default value, right-click on it’s name and choose “Revert Value to Prefab”.

Turning an Override Value into Its Default Value

If you ever decide that the override value you gave to one of your prefab instances should be applied to all copies of that prefab, you can do so by pressing the “Apply” button found at the top of the Inspector.



You'll also find other prefab-specific buttons there:

“Select” highlights the prefab (the master copy) in the Project View.

“Revert” reverts all overridden values in all components of that prefab instance to the defaults.

So we got our enemies chasing us, it’s not the best way of doing it, but it works, later on we’ll look at a more advanced way of having our AI move around but that’s still to come. The next tutorial will deal with ways of damaging and interacting with enemies and objects.

Challenges

Challenge 2

Add a new enemy type that is faster than the others, change it’s colour to a new colour that isn’t being used. Save this as a new prefab.

Challenge 3

Stop the AI from bashing into the player, this is quite a bit harder, you need to check if the magnitude of playerMode.position- transform.postion is greater than 2 then move the player. This means that if the distance between the two (the magnitude of the vector) is not more than 2 we won’t move.