Game Programming Using Unity 3D

Lesson 6: Basic Enemy AI

Ferdinand Joseph Fernandez

Chief Technological Officer, Dreamlords Digital Inc.

Admin and Co-founder, Unity Philippines Users Group

September 2011

Overview

Again, we'll be continuing where we left off from the previous lesson. If you haven't done Lesson 5, you need to do it before you can continue here.

We'll move away from doing player code in the meantime and start working on our enemy AI. AI means artificial intelligence, and it simply means having computer-controlled opponents that try to mimic a human level of skill and behaviour as much as possible.

For now, we'll just do something basic: making enemies that chase the player wherever he goes.

Placeholder Art For The Enemies

For now, we'll make boxes that represent enemies.

Make a cube (GameObject > Create Other > Cube). Scale it to (1, 2, 1).

Just like the player, we'll also make use of a character controller here to make the enemy move. Add a character controller (Component > Physics > Character Controller). Click “Replace” when prompted about the “Replace existing component?” warning.

Finally, rename the cube to “Enemy”.

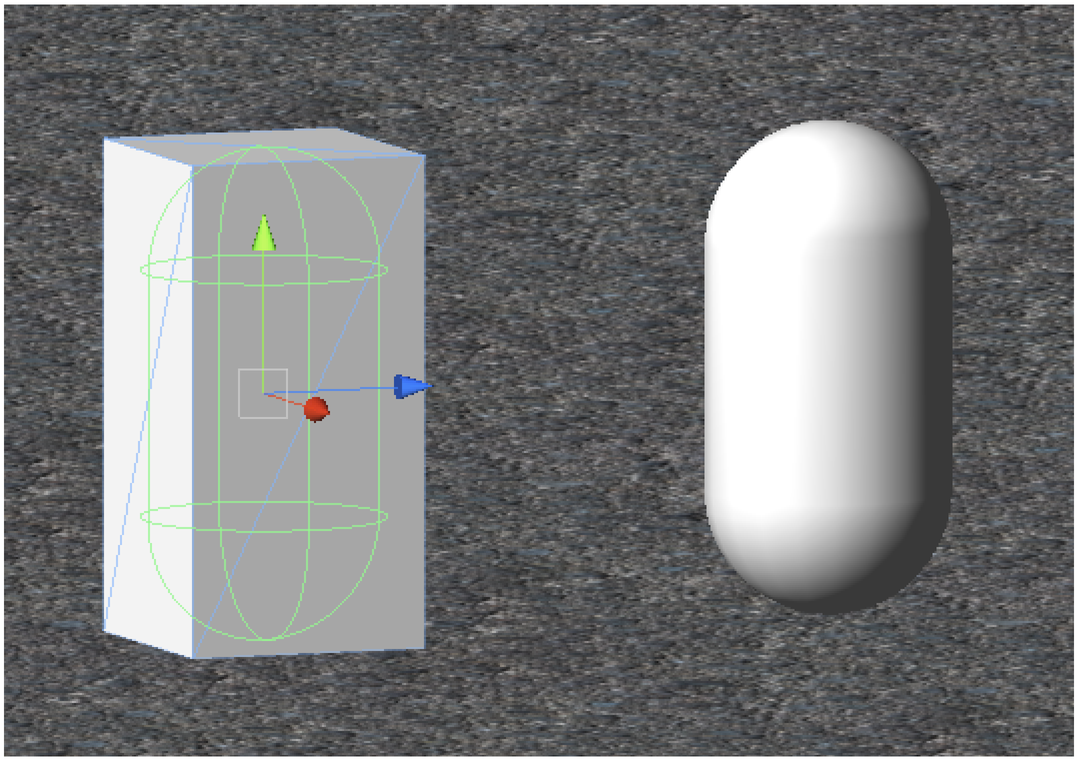


Illustration 1: The enemy box should be roughly about the same size as the player.

Creating The Script

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).

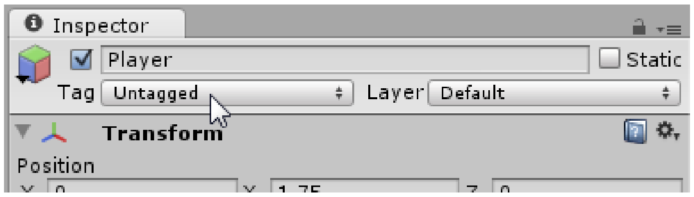
Add this code to the EnemyMovement script:

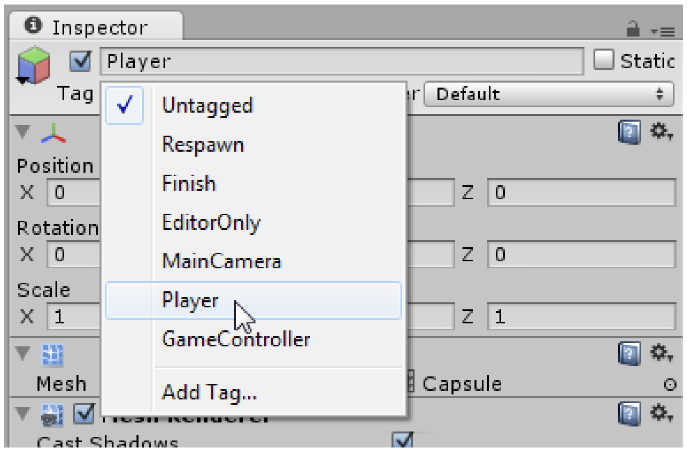
```
using UnityEngine;
using System.Collections;

public class EnemyMovement : MonoBehaviour
{
	Transform _player;

	void Start()
	{
	}

	void Update()
	{
	}
}
```

Here you'll see us making use of a transform that's not our own. “_player” 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. Here we'll make use of what Unity calls “tags”.

Using Tags

Go to your Unity Editor window. Select the Player game object. 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.

Go back to the EnemyMovement script. Add this code:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class EnemyMovement : MonoBehaviour
05 {
06        Transform _player;
07
08        void Start()
09        {
10               GameObject playerGameObject = GameObject.FindGameObjectWithTag("Player");
11               _player = playerGameObject.transform;
12        }
13
14        void Update()
15        {
16        }
17 }
```

We're able to find the Player game object by letting Unity search it for us. In line 10, FindGameObjectWithTag searches through the current scene for a game object that has the tag that we specify. And we specify here “Player”. Since we gave the “Player” tag to our Player earlier, we're able to get the Player game object this way.

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

Initializing Character Controller

The same way our PlayerMovement script makes use of the character controller component, our EnemyMovement will be using its character controller.

```
using UnityEngine;
using System.Collections;

public class EnemyMovement : MonoBehaviour
{
       CharacterController _controller;
       Transform _player;

       void Start()
       {
              GameObject playerGameObject = GameObject.FindGameObjectWithTag("Player");
              _player = playerGameObject.transform;

              _controller = GetComponent<CharacterController>();
       }

       void Update()
       {
       }
}
```

Chasing The Player

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:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class EnemyMovement : MonoBehaviour
05 {
06     CharacterController _controller;
07     Transform _player;
08
09     void Start()
10     {
11         GameObject playerGameObject = GameObject.FindGameObjectWithTag("Player");
12         _player = playerGameObject.transform;
13
14         _controller = GetComponent<CharacterController>();
15     }
16
17     void Update()
18     {
19         Vector3 direction = _player.position - transform.position;
20
21         _controller.Move(direction * Time.deltaTime);
22     }
23 }
```

Line 19 does the work needed so the enemy knows where to go to. Let's test the script first to verify its working. Go ahead and test it by attaching the EnemyMovement script to your Enemy game object.

Start the game then run anywhere. The cube enemy should follow you around.

So how come this simple formula in line 19 made the enemy follow us? We are actually computing the delta 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, to chase the player.

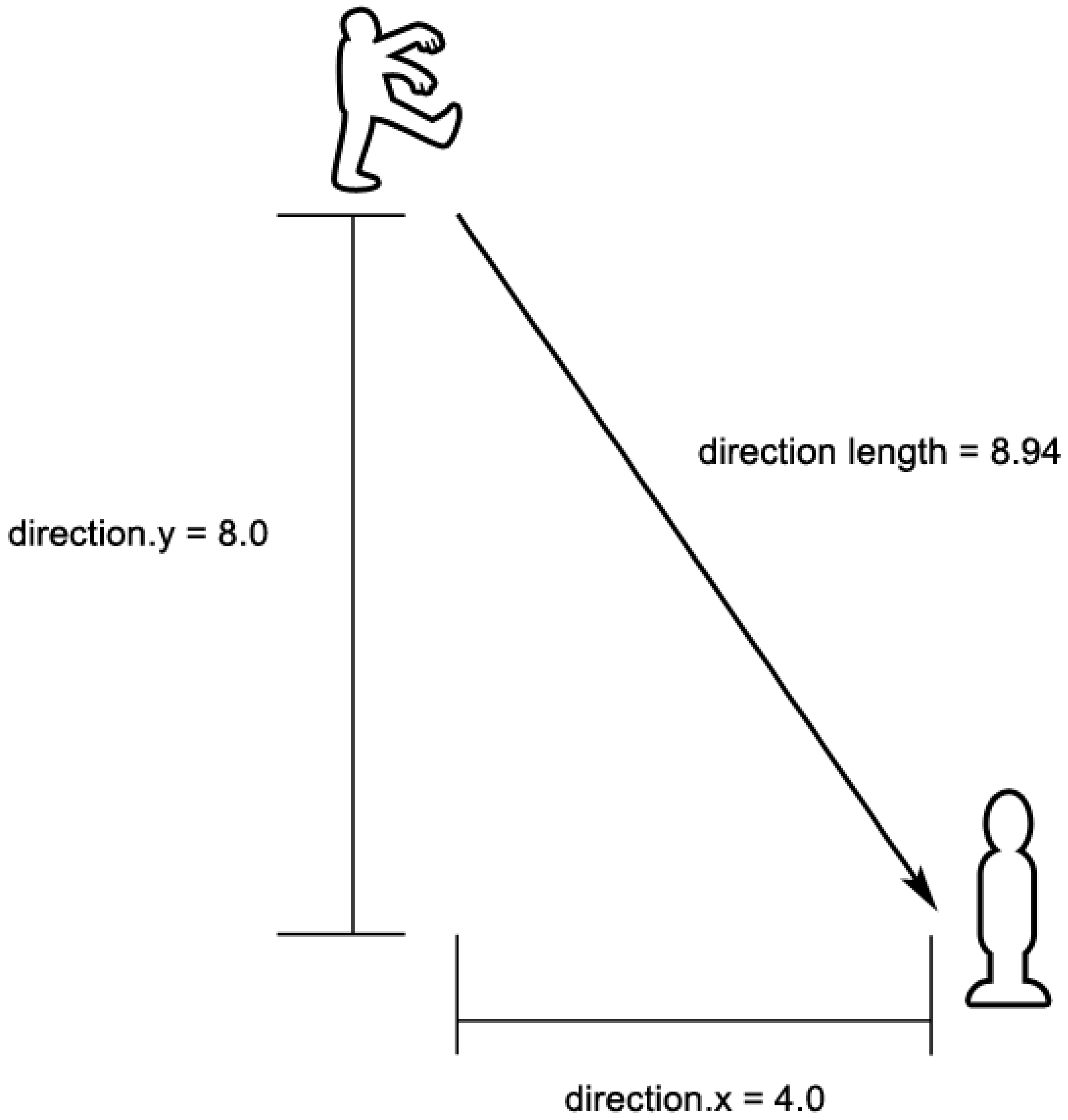
Compare line 19 of our code with the formula. “source”, in this case, is our own position (transform.position), and “destination” is the player's position (_player.position). These positions are Vector3 values (it has x, y, and z coordinates), so likewise, we need to store the subtraction result in a Vector3.

Normalize!

However if you notice, the enemy slows down as it gets near the player. This makes sense, since the more our destination and source get near each other, the lesser the result will be when you subtract them from each other.

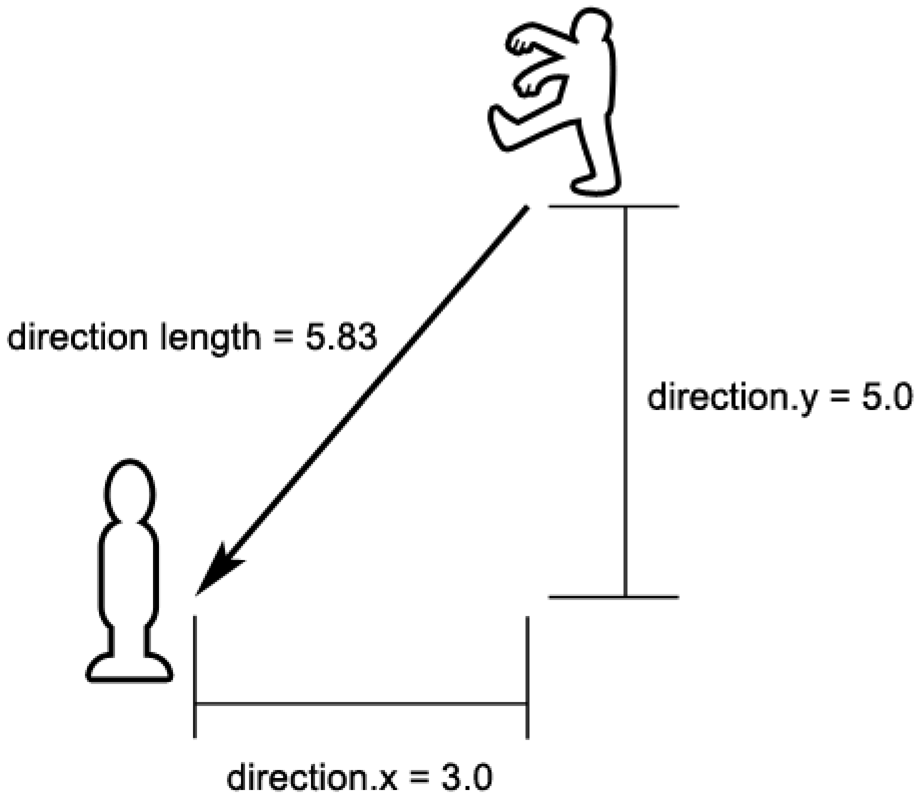
Consider this example:

Here we show the direction variable as an arrow.



In this example, the distance between the enemy and the player is 8.94 units long. The direction variable's length then, is the same 8.94 units.

Now look at the next diagram. In a situation where the enemy is closer to the player, the direction's length becomes smaller. Remember, in line 21 of our code, its this direction variable that we feed to the Move function. Since the direction length is lower, the enemy slows down.

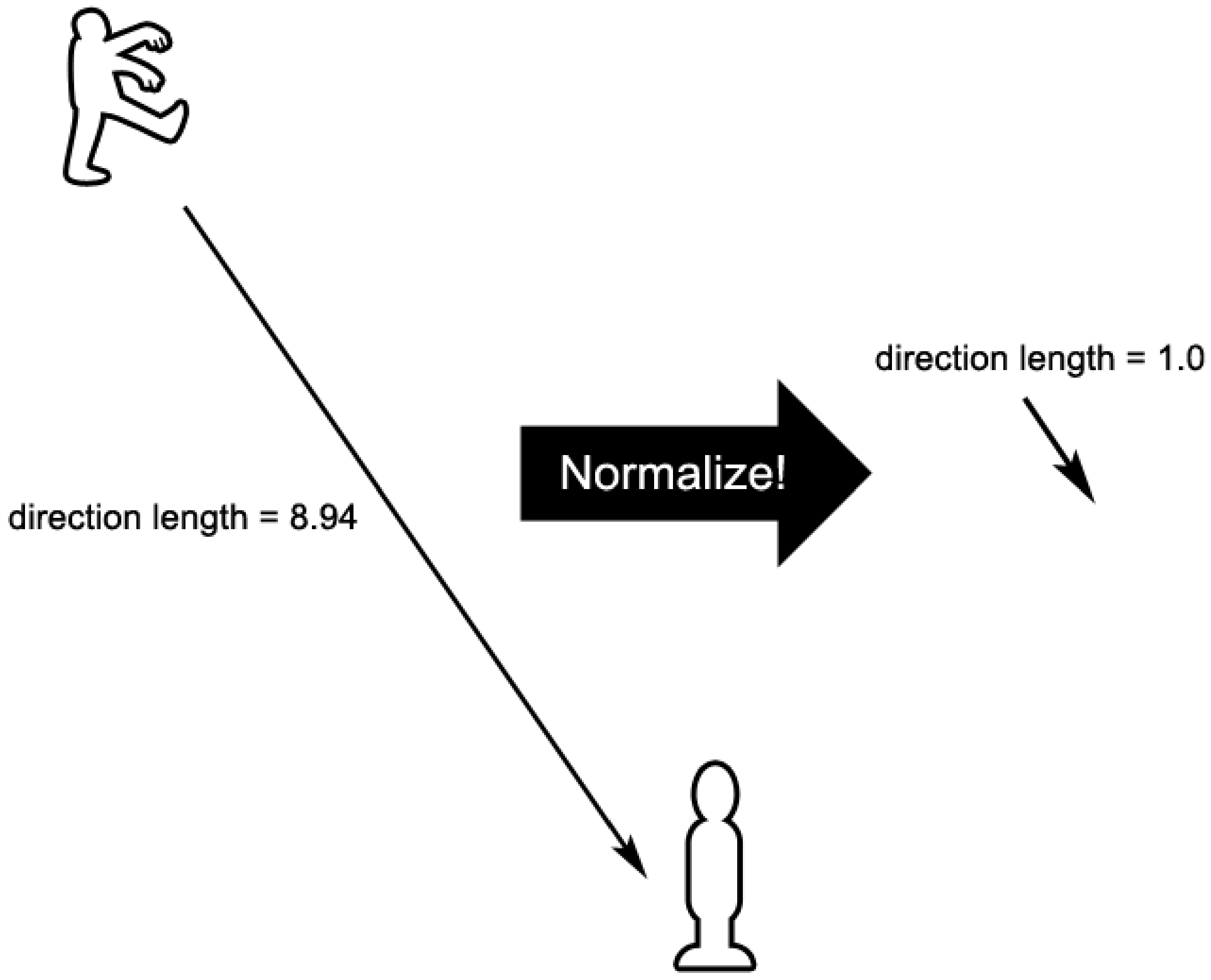


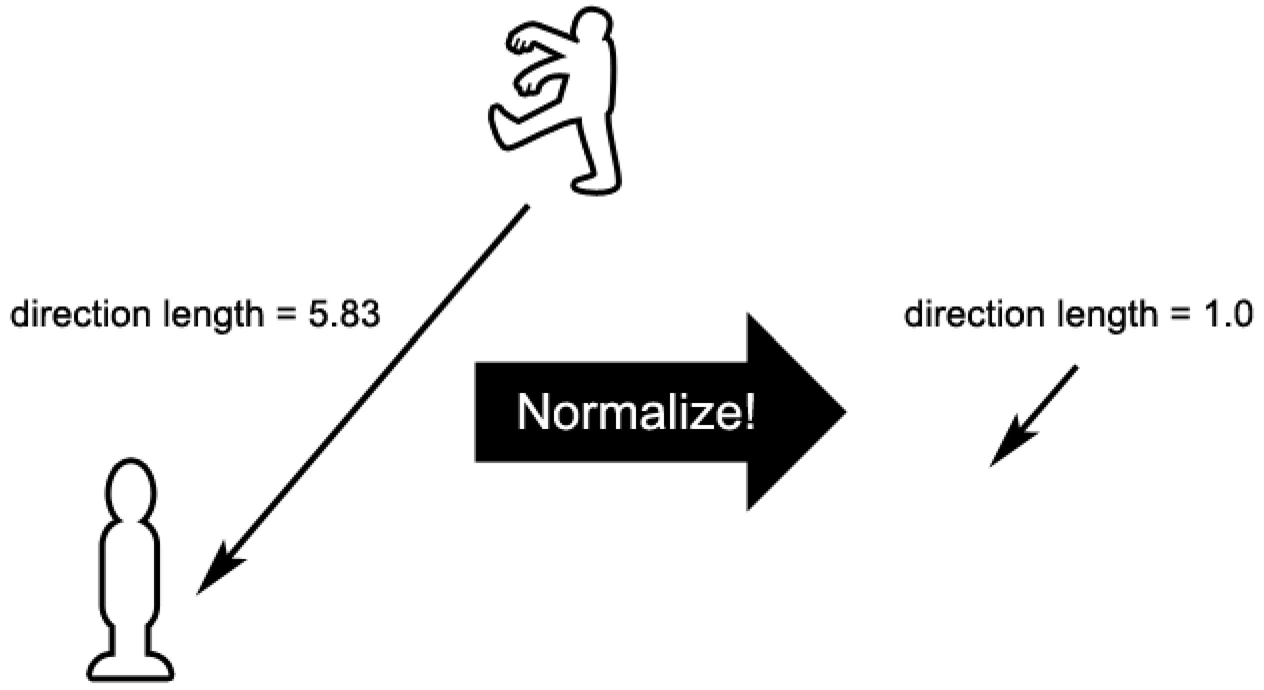
Let's say this isn't the behaviour that we want. Our enemies are zombies, and we don't want them to slow down as they get near the player.

The problem is our direction variable gets a lower value as it gets near the player. This is what we need to fix. We need to convert our direction so its length is a constant value. We do this by normalizing the vector.

Normalizing a vector changes its length to 1.0.

Consider what happens to the two examples above when their directions are normalized.


direction length = 8.94
Normalize!
direction length = 1.0


direction length = 5.83
Normalize!
direction length = 1.0

You'll notice that normalizing changes the direction length to 1.0, but in each situation, the arrow still points the same way. **Normalizing only changes the length, not the direction of a vector.**

Fortunately, normalizing is an easy process in Unity. See the code below:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class EnemyMovement : MonoBehaviour
05 {
06        CharacterController _controller;
07        Transform _player;
08
09        void Start()
10        {
11               GameObject playerGameObject = GameObject.FindGameObjectWithTag("Player");
12               _player = playerGameObject.transform;
13
14               _controller = GetComponent<CharacterController>();
15        }
16
17        void Update()
18        {
19               Vector3 direction = _player.position - transform.position;
20               direction.Normalize();
21
22               _controller.Move(direction * Time.deltaTime);
23        }
24 }
```

Line 20 does the normalizing. Vector3 has a function called Normalize that does the trick.

You can also write it this way:

```
20               direction = direction.normalized;
```

“normalized” on the other hand, returns the normalized version of a Vector3 but doesn't change its own value. So we have to feed it back to the direction variable if we use that.

If you run the game now, you'll see it moves slower. Since the direction length is now always 1.0, its now moving at a constant 1 meter per second. We can change this by adding a movement speed variable the same way we did for the player movement.

Adding Enemy Movement Speed

The code we add is similar to the PlayerMovement script:

```
using UnityEngine;
using System.Collections;

public class EnemyMovement : MonoBehaviour
{
	CharacterController _controller;
	Transform _player;

	[SerializeField]
	float _moveSpeed = 5.0f;

	void Start()
	{
		GameObject playerGameObject = GameObject.FindGameObjectWithTag("Player");
		_player = playerGameObject.transform;

		_controller = GetComponent<CharacterController>();
	}

	void Update()
	{
		Vector3 direction = _player.position - transform.position;
		direction.Normalize();

		Vector3 velocity = direction * _moveSpeed;

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

Don't give the enemy too much speed or the player won't have a chance to run away!

In Conclusion...

We've brushed over some vector math for something our game needs: making the enemy follow the player. You've also learned something about tags.

In the next lesson, you'll find an easy way to make more enemies using the feature that Unity calls “prefabs”.