Silver Belt Ninja Guide
Activity 04: Jungle Escape

Activity 4

Jungle Escape

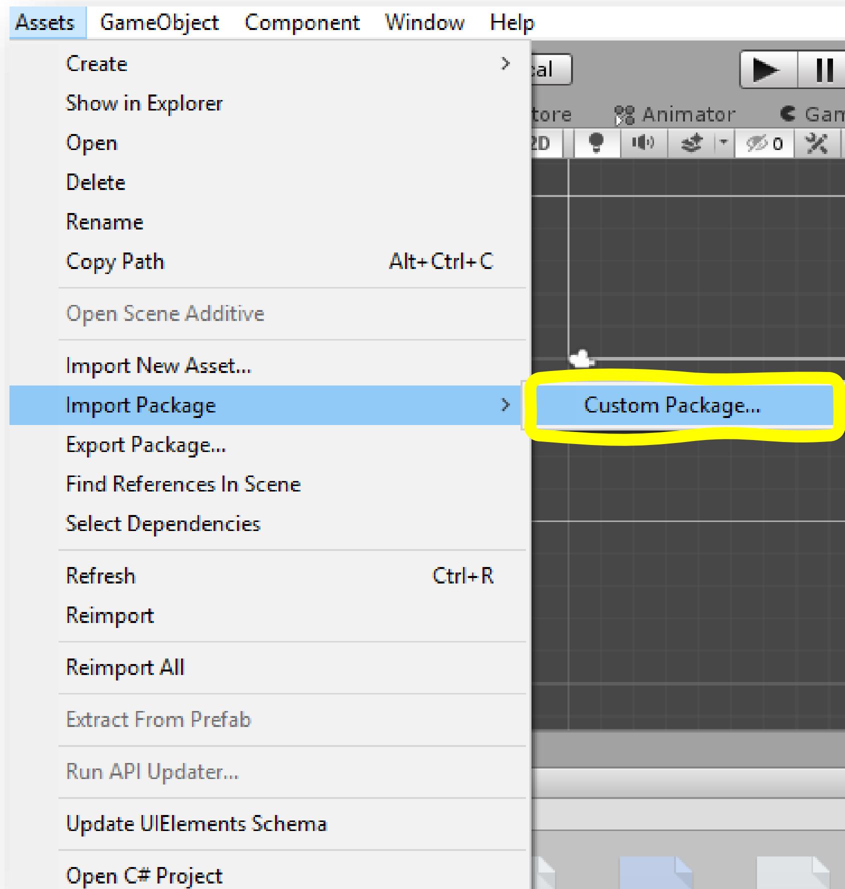
Through this game, you will learn about **raycasts** and use it as another way to check if the player is grounded. You'll also make jumping more realistic by adjusting gravity as the player falls. Lastly, you'll learn about the **Animator Controller**, including how to control the Animator's parameters in your code.

Today's mission: Navigate the wooden boards toward the glowing portal that will teleport you to your escape: the beach! Be careful though – some boards may disappear on you and others will try to trick you!

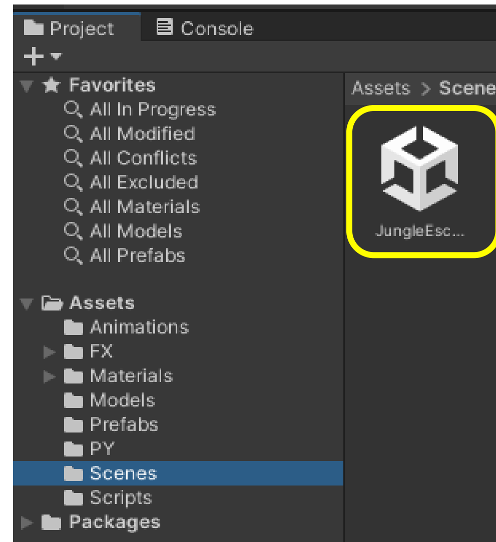
If you press play, you will notice that Codey's animations do not play. Can you figure out the second problem with Codey?

1 Start a new Unity Project and name it *YOUR INITIALS – JungleEscape*. Select **3D core**.

2 Import the Jungle Escape- starter Unity Package by going to **Assets > Import Package > Custom Package > All > Import**



3 Double-click on the **JungleEscape** scene. You can find this in the **Project** tab under **Assets > Scenes**.



4 Go to the **Game** tab and change the Game Resolution to 16:9.

5 In your **Project** tab, open the C# Script named **Jump**. Notice the code adding force is already there. This is the same code we used in Cloud Hop. Do you see anything missing?

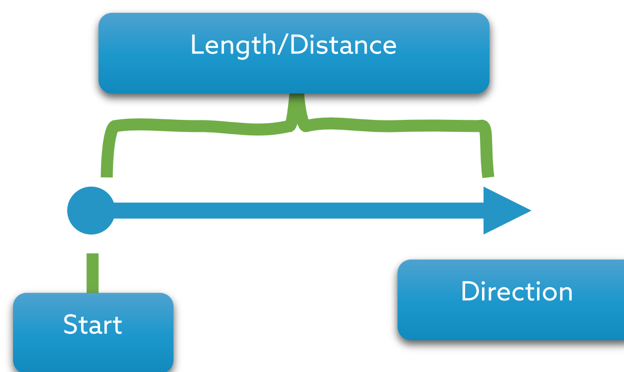
```
    Unity Script (2 asset references) | 0 references
7   public class Jump : MonoBehaviour
8   {
9       Rigidbody rb;
10
11      float jumpForce = 5.7f;
12
        Unity Message | 0 references
13      void Start()
14      {
15          rb = GetComponent<Rigidbody>();
16      }
17
        Unity Message | 0 references
18      void Update()
19      {
20          if(Input.GetButtonDown("Jump")){
21              rb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
22          }
23      }
24  }
```

6 There is no grounded condition yet! What happens when there is no grounded check? This should help you figure out what, besides the animation, is wrong with the game. Press play and test out your theory!

Did you repeatedly press spacebar? The second problem is that the player can infinitely jump, without ever touching the ground!

7 Cloud Hop taught you how to check if the player is grounded using the player's vertical velocity. In Jungle Escape, you will learn another way to check if the player is grounded. We will be using a raycast with the function `Physics.Raycast()`.

8 Like the name suggests, a **raycast** casts a ray. What does that mean? It creates a line with a specific start point, length, and direction.



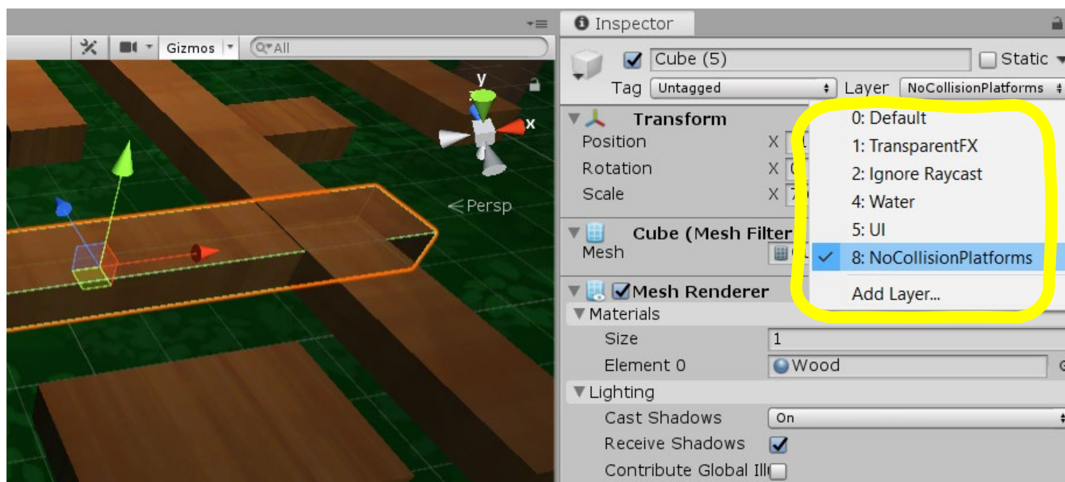
9 The main parameters for `Physics.Raycast()` are as follows:

Physics.Raycast(Vector3 **origin**, Vector3 **direction**, float **distance**, int **layer**)

What is a Layer?

Objects are put into different layers so that you can select which functions affect it.

If you include the layer parameter in a raycast, then the raycast only returns true if it hits an object in that layer.



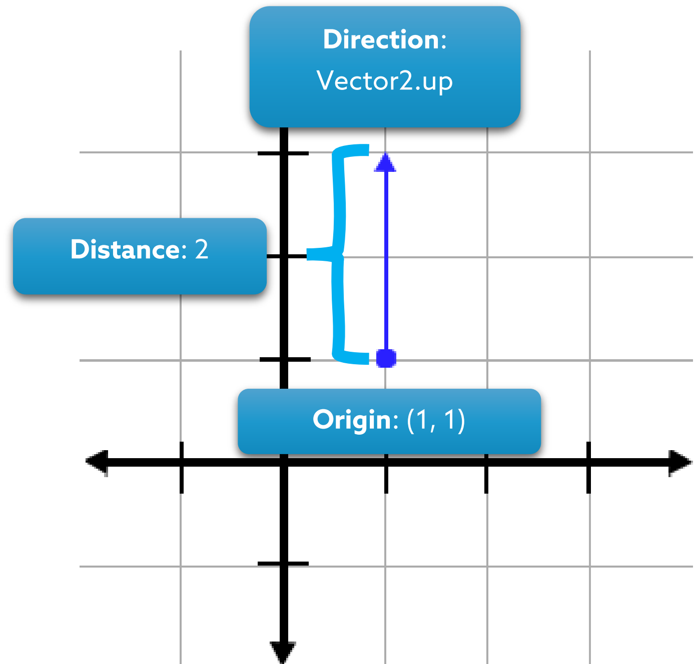
In this picture, the selected platform is in layer 8, which we've named **NoCollisionPlatforms**. To reference a layer, you must use a very specific format. We cannot use the number 8.

Instead, format it like this: 1 << layer. For a **raycast** only on layer 8, it would read: **Physics.Raycast(origin, direction, distance, 1 << 8).** If you want the raycast to look at all layers, simply leave out the layer parameter.

You can also create a **layermask variable** to make it a bit easier to read.

EXAMPLE:

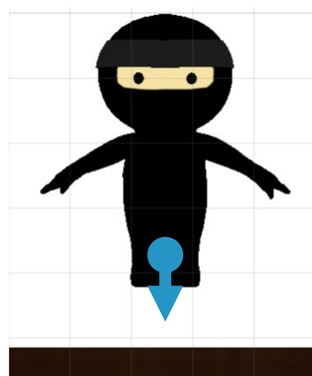
What do you think the function looks like for the ray below? Assume it looks at all layers.



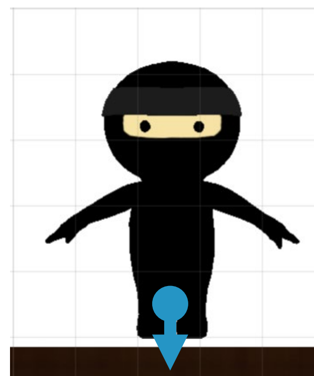
The function for this ray is: `Physics.Raycast(new Vector2 (1, 1), Vector2.up, 2)`.

10 How does using a ray help check if the player is grounded?
`Physics.Raycast()` returns true when it touches another collider! So, if we cast a ray down from our player for a specific distance, then we know the player is grounded when the raycast is true.

Ray NOT touching ground
Physics.Raycast() = false



Ray touching ground
Physics.Raycast() = true



11 How do we get the origin to follow our player? A new Vector is just an unchanging location; we need to get the constantly changing location of our player. We can make the vector origin change using `transform.position`, which always gets the position of the GameObject that the script is attached to. In this case, it's our player.

Now you'll get to see it all come together in our Jungle Escape game.

12 Let's get coding! In your **Project** tab under the **Scripts** folder, open the **Jump** script. First declare a `public bool isGrounded`. This will serve the same purpose as the `canJump` bool used in Cloud Hop.

```
public class Jump : MonoBehaviour
{
    Rigidbody rb;

    float jumpForce = 5.7f;

    public bool isGrounded;

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

`isGrounded` will be assigned to the `Physics.Raycast()` so we can easily access the ray's return. Start by adding the following line to your `Update()`:

```
void Update()
{
    isGrounded = Physics.Raycast();

    if(Input.GetButtonDown("Jump")){
        rb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
    }
}
```

13 Now using what you've learned about `Physics.Raycast()` create a function in `Update` that casts a ray from the player for a short distance. Remember the three parameters: (origin, direction, distance).

Add the parameters as shown here:

```
void Update()
{
    isGrounded = Physics.Raycast(transform.position, Vector3.down, .15f);

    if(Input.GetButtonDown("Jump")){
        rb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
    }
}
```

14 Before we press play, Unity has a handy function that lets us see the ray we just created: `Debug.DrawRay()`!

```
Unity Message | 0 references
void Update()
{
    isGrounded = Physics.Raycast(transform.position, Vector3.down, .15f);
    Debug.DrawRay(transform.position, Vector3.down * .15f, Color.red);

    if(Input.GetButtonDown("Jump")){
        rb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
    }
}
```

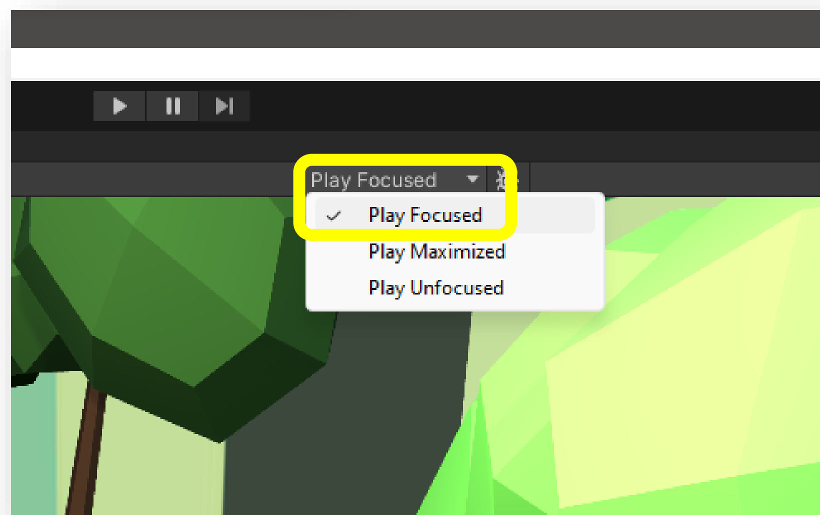
15 **Save** your script.

16 Let's see what the **raycast** looks like!

Note: `isGrounded` is not yet included in the condition for jump; this means you can still infinitely jump.

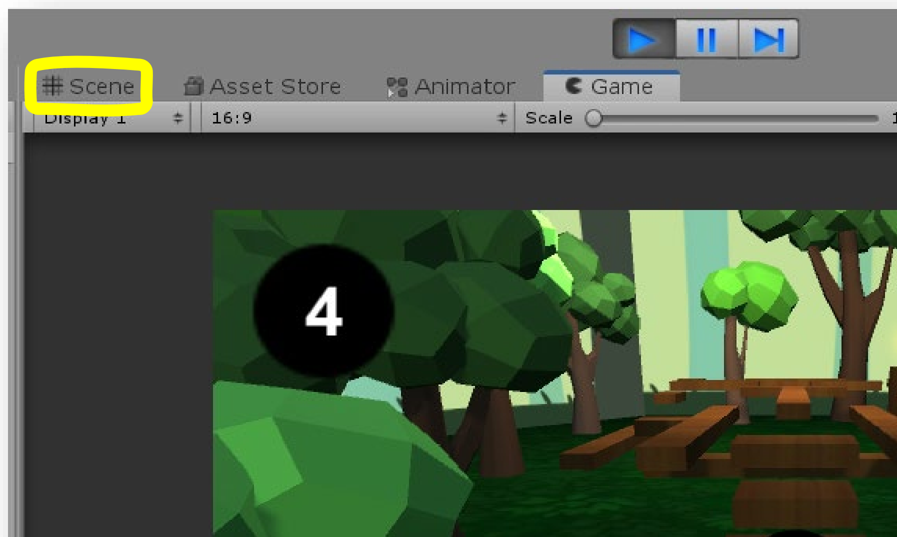
Press **play**.

17 If your game is full screen, click on **Play Focused** in the game window (circled below):

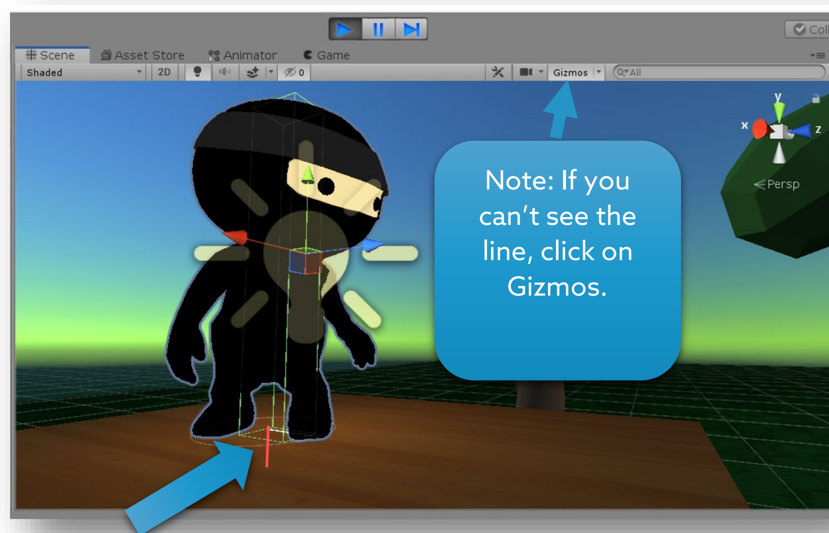


CODE NINJAS®

Next, click the **play** button to exit play mode. Then press **play** again. Click on the **Scene** tab.



18 Zoom in and rotate around Codey to see the **raycast** you made. Remember in the `Debug.DrawRay`, you included `Color.red`? Can you find the **raycast**?



Note: The ray goes through the wooden platform, so it may be hard to see. If you are having trouble finding it, press space bar to get Codey to start his jump, then pause the game. Press the skip button until you can see the ray.

19

Awesome job, ninja! All the pieces are coming together!

Let's do a quick review. We have two separate functions: an add force and a physics raycast stored in `isGrounded`. What is the final step we need to finally get rid of the continuous jump?

Stop and think about what we need to do before you move on to the next step.

20

That's right! Add `isGrounded` to the add force's ***if*** condition:

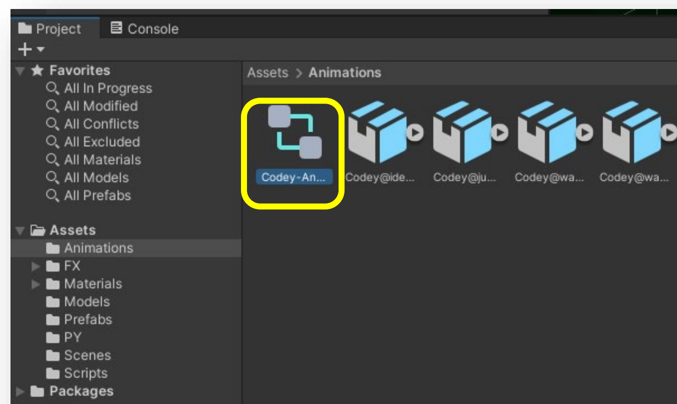
```
Unity Message | 0 references
void Update()
{
    isGrounded = Physics.Raycast(transform.position, Vector3.down, .15f);
    Debug.DrawRay(transform.position, Vector3.down * .15f, Color.red);

    if(Input.GetButtonDown("Jump" && isGrounded){
        rb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
    }
}
```

21

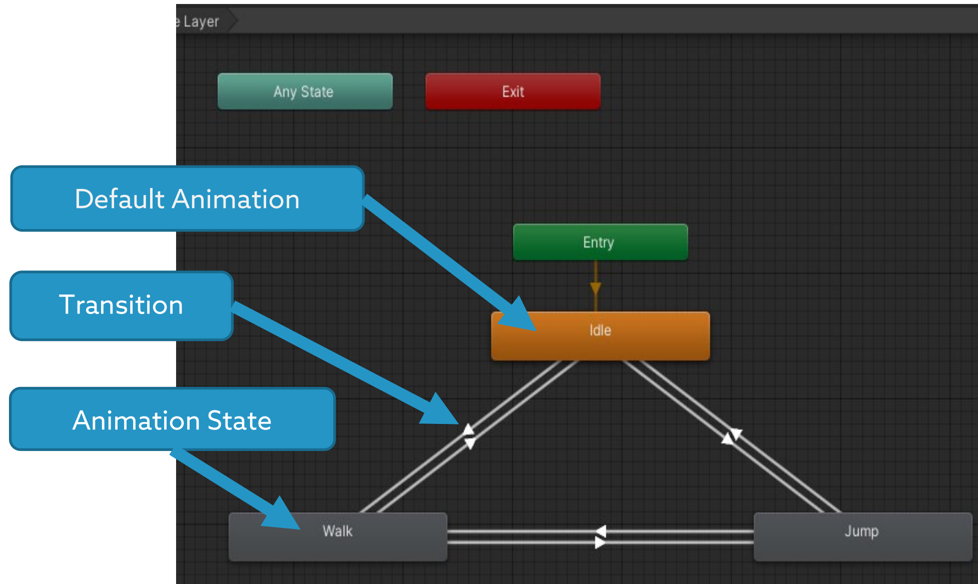
Save your script and test out your jump!

22

Now that you've mastered movement code, you must now master animation. If you recall from your lessons in Bronze Belt, animations are controlled in the animator controller. In your **Project** tab under the **Animations** folder, you will find the **animator controller** for Codey. Double-click on the controller to open it.

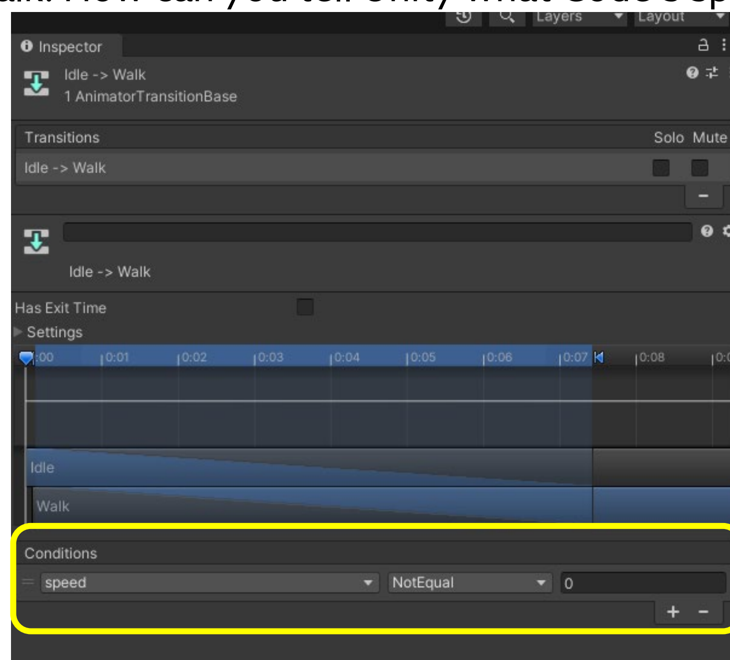


23 Let's review the animator controller interface. Each rectangle is an **animation state**, so in this case Codey can either be idling, walking, walking backwards, or jumping. The arrows connecting the states are **transitions**.



24 An arrow means Codey can transition from that animation to the other if it meets the conditions. Click on the **transition** arrow from Idle to Walk and look in the **Inspector**.

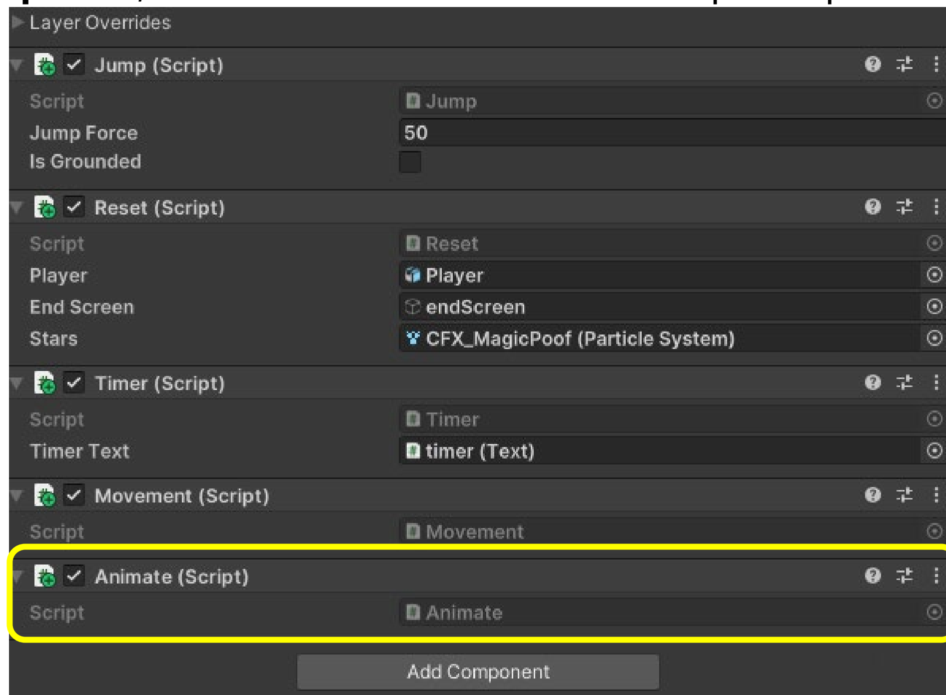
25 Under **Conditions**, speed must be greater than 0 for Codey to change from idle to walk. How can you tell Unity what Code's speed is?



26 Let's get coding! The condition speed is a whole number, or integer (int for short!). Therefore, we must set up if statements that will say when each animation condition is true or false. First, let's open up the Animate script.

27 Click on the **Player** in the **Hierarchy**.

In the **Inspector**, double click on the **Animate** script to open it up.

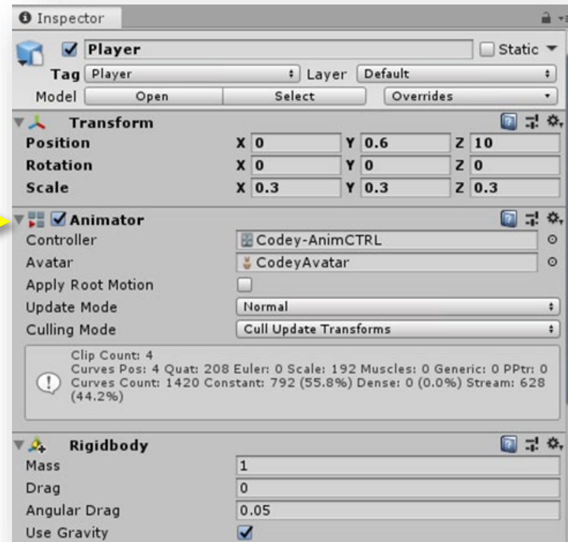


28 Declare `Animator animator`. This will let Unity know that whenever we use the name Animator, we are referencing the type Animator (for Animator Controller).

```
public class Animate : MonoBehaviour
{
    Animator animator;

    Unity Message | 0 references
    void Start()
```

29

Remember that the Animator is a component. How did we access a game object's component before, like the rigidbody? We use `GetComponent<>()`.



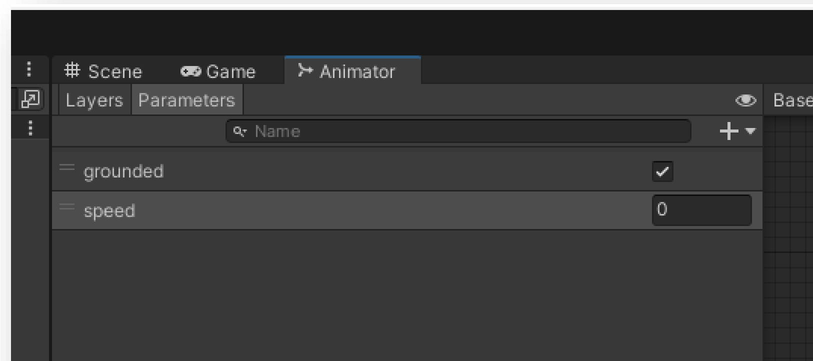
Add the following line to your `Start()` function:

```
void Start()
{
    animator = GetComponent<Animator>();
}
```

30

Now that we have the Animator Controller in our script, let's think about how to organize our Animate script! We need it to control when Codey stays in an animation or transitions between animations. Remember, in the **Animator Controller**, Codey needed the speed condition.

Before we think about the speed, let's first look at the other parameter, grounded.



31 Notice that we included "grounded" as a factor. For jump, the user input is the spacebar, but the player must also be grounded. If we just use `Input.GetButton("Jump")` as the condition for the jump animation, what happens when Codey is in the air and the spacebar is pressed? Codey does not jump again, but his jump animation will play again.

32 We need to use `isGrounded` as a condition for the jump animation, but the `isGrounded` variable was made in the Jump script. How do we access a variable that was made and defined in another script?

33 Just like other types, you can declare a script. This makes all declarations and functions from that script recognized in the current script. To declare a script, the type is the name of the script, then the name you would like to call it in the current script.

For example, to declare the Jump script, write the following line:

```
public class Animate : MonoBehaviour
{
    Animator animator;
    Jump jump;
```

34 Assign the Jump component to the variable using `GetComponent<>()`

```
void Start()
{
    animator = GetComponent<Animator>();
    jump = GetComponent<Jump>();
}
```

35 To use variables or functions from Jump all we need to do is include the script name with a period, like so: `Script.Variable` or `Script.Function()`.

36

Great! Now that we can access `isGrounded`, let's set the animation bool to be the same. In the **Animate** script under `Update()`, add the following statement:

```
void Update()
{
    animator.SetBool("grounded", jump.isGrounded);
}
```

Remember

An exclamation mark "!" in front of a variable means NOT. So **!Jump.isGrounded** means isGrounded equals false.

37

Now let's get the speed parameter. Just like how we created a reference to the Jump script, we are going to get a reference to the movement script. Insert the code as shown below.

```
public class Animate : MonoBehaviour
{
    Animator animator;
    Jump jump;
    Movement movement;

    Unity Message | 0 references
    void Start()
    {
        animator = GetComponent<Animator>();
        jump = GetComponent<Jump>();
        movement = GetComponent<Movement>();
    }
```

38 Now using `SetFloat`, get the speed from the movement script and set the animation parameter to match.

```
void Update()
{
    animator.SetBool("grounded", jump.isGrounded);
    animator.SetFloat("speed", movement.speed);
}
```

39 **Save** your script. Your game is ready to go! Press **play** and test it out.