Collisions 2D in Unity

[Understanding Collisions in Unity](#_wdxap2d4ruq9)

[Collision Resolution](#_dhf4ytix4q5e)

[Collision Detection](#_clv7jpi408gm)

[Detecting Collisions](#_li28l8tehnng)

[Creating Tags](#_2has8btrv88q)

[Using Tags](#_h1ky1cmnlr70)

[Trigger Colliders](#_387x3zsadljn)

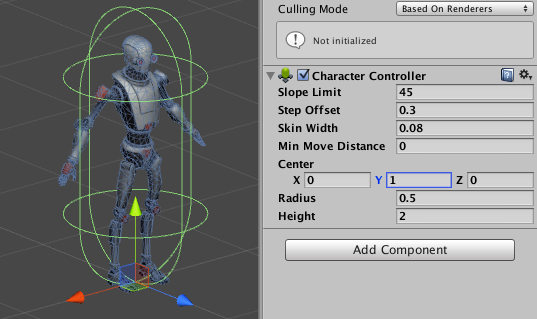
[Collision Layers](#_mtkfjo3v3pcu)

# 

# Understanding Collisions in Unity

In Unity, it is very common to test whether two **gameobjects** are colliding with each other. This behaviour is mediated via a family of components called **colliders**. Colliders are added to an object to define its shape. Most 3D models are high resolution; using them to calculate the shape of an object is often impractical. Unity provides a series of simple colliders (boxes, circles, polygons, cubes, sphere, cylinders…) that can be used to model simpler shapes.

Capsules and cylinders, for instance, are often used to as basic player colliders in FPS games. This can be seen in the image below, where a high poly robot (in blue) is surrounded by a cylindrical collider (in green).



Collisions in Unity are used for two separate gameplay aspects: *resolution* and *detection*.

* **Collision resolution**: when two physical objects collide with each other, a desirable behaviour is to have them bouncing off in a realistic manner. The *resolution* of their collision might require to change their position, rotation and velocity according to *Newton’s Laws of Motion*.
* **Collision detection**: in this scenario, we are interested in detecting when two gameobjects are colliding with each other. This might be in the case of two physical objects (for instance: two pool balls hitting each other), one physical and one non-physical object (for instance: a player entering a specific area), or even two non-physical objects (for instance: the mouse is over a button).

## Collision Resolution

Since colliders in Unity are *passive* components, they cannot change the state of the gameobjects they are attached to. If you want your objects to responds to collision in a realistic fashion, a **Rigidbody2D** will be necessary for collision resolution. During a collision, only the gameobjects fitted with a rigidbody can be moved by the physics engine.

When a gameobject has a rigidbody and a collider, no other action is generally required to have physical collisions working. However, it is important to configure the components properly. The rigidbody **mass**, for instance, is critical during collisions. Likewise, make sure that both objects involved in a physical collisions have a **physics 2D material** attached to them.

## Collision Detection

While *collision resolution* happens by default on rigidbodies fitted with colliders, the same is not true for *collision detection*. Detecting collisions is something that can only be via code, and there are two techniques to do it: *passive detection* and *active detection*.

* **Passive collision detection**: this is the most common scenario, and it requires to add special *methods* (called *events*) to your script, which the physics engine will call in the event of a collision.
* **Active collision detection**: in the case of passive collision detection, it is the physics engine to decide when to call the collision events. In certain situations, it is convenient to test *immediately* if an object is colliding with something else. Unity allows to do that with a series of methods available in the Physics2D class, which can be used in the Update method.

## 

# Detecting Collisions

Gameobjects which have a collider component attached to them can, under the right circumstances, be notified about **collision events**. This is possible by adding a script which contains one of these special method:

* OnCollisionEnter2D: called in the first frame during which a collision occurs.
* OnCollisionStay2D: called during each frame while the collision occurs.
* OnCollisionExit2D: called when the collision stops.

The following code can be used to detect the three stages of a collision:

|  |
| --- |
| public class Spaceship : MonoBehaviour  {  void **OnCollisionEnter2D** (Collision2D collision)  {  Debug.Log("Collision started!");  }  void **OnCollisionStay2D** (Collision2D collision)  {  Debug.Log("Collision in progress!");  }  void **OnCollisionExit2D** (Collision2D collision)  {  Debug.Log("Collision ended!");  }  } |

All the three OnCollision- methods are called with a Collision2D parameter. The physics engine will use that to provide information about the collision. For instance, it is possible to retrieve the other collider involved in the collision using collision.collider, and even a list of the contact points with collision.GetContacts.

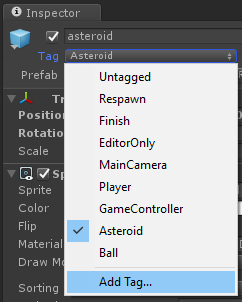
|  |
| --- |
| **⚠ Moving colliders**  If you change the position of an object manually (assigning a new value to its transform.position), there is no guarantee that the OnCollision- methods will be called as expected.  This is particularly problematic with *trigger colliders*. Suddenly moving an object from inside a trigger collider might prevent the OnTriggerExit2D and OnTriggerEnter2D method to be called.  If you need to move a gameobject manually, use Rigidbody2D.MovePosition instead. If you need to teleport it to a different location, you can still use transform.position, but be aware of the consequences. |

## Creating Tags

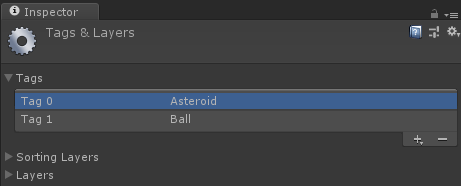
By default, the physics engine will notify the gameobjects of *all* collisions. Being able to differentiate behaviours based on which object you collided with is essential. There are several ways to this this in Unity. The easier way is to use a feature called **tags**.

Tags are labels that can be attached to gameobjects. Each gameobject can only have one tag at a time. Imagine a game in which a spaceship can collide with both asteroids and fuel packs; you can use two tags to to mark these two classes of objects.

In Unity, you can create as many tags as you want, by clicking on the **Tag** drop-down menu in the *Inspector* and selecting **Add Tag...**.



Then, expanding the **Tags** foldout and creating a new one by pressing on the **+** button.



When a tag is created, you can assign it to an object from the **Tag** drop-down menu used before in the Inspector.

## Using Tags

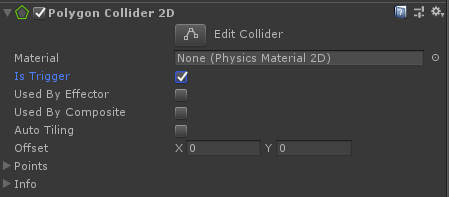
A gameobject can be tested to see if it has been marked with a certain tag, using the method CompareTag. This is often done in the OnCollision- methods.

The following snippet checks for collision with gameobjects tagged as Asteroid, and destroys the spaceship if such an event occurs.

|  |
| --- |
| public class Spaceship : MonoBehaviour  {  void **OnCollisionEnter2D** (Collision2D collision)  {  if (collision.gameObject.CompareTag("Asteroid"))  {  Destroy(gameObject); // Destroys this gameobject  }  }  } |

## Trigger Colliders

Collision resolution is not always a desired behaviour. Sometimes colliders might represent non-physical objects, such as *areas* the player have to cross. In this case, the colliders can be marked as as **triggers**.



A trigger collider passes through other physical objects, but still responds to both passive and active collision detection.

If you are using passive collision resolution, the name of the events change whether you have a trigger collider or not.

|  |  |  |
| --- | --- | --- |
| **Type** | **2D** | **3D** |
| Collider | OnCollisionEnter2D (Collision2D)  OnCollisionStay2D (Collision2D)  OnCollisionExit2D (Collision2D) | OnCollisionEnter (Collision)  OnCollisionStay (Collision)  OnCollisionExit (Collision) |
| Trigger | OnTriggerEnter2D (Collider2D)  OnTriggerStay2D (Collider2D)  OnTriggerExit2D (Collider2D) | OnTriggerEnter (Collider)  OnTriggerStay (Collider)  OnTriggerExit (Collider) |

|  |
| --- |
| **⚠ Rigidbodies and trigger colliders**  Trigger events (OnTriggerEnter2D, OnTriggerStay2D, OnTriggerExit2D, ...) are only called if *at least* one of the two gameobject has a rigidbody attached to it.  This is because collision events (including triggers) are dispatched by the physics engine. |

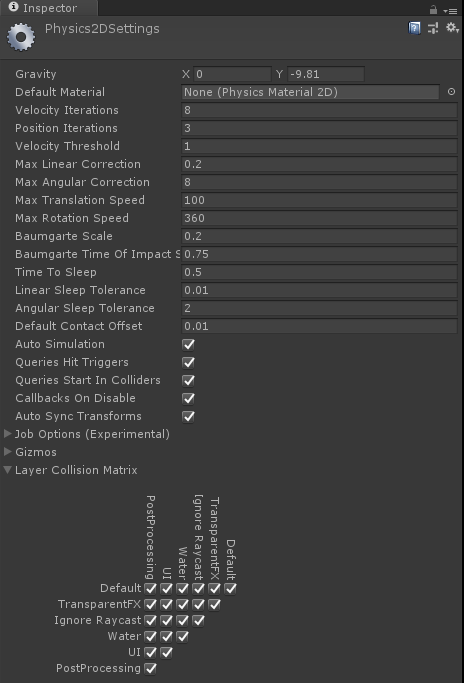
# 

## Collision Layers

Sometimes there might be classes of gameobjects that you do not work to collide with each other. For instance, the player and the bullet it spawn do not typically collide. Unity has a concept of **collision layer**, which can be used to determine which layers can collide with each other.

The process to create a collision layers is similar to the one required to create a *tag*. A gameobject can only be on a single collision layer at once.

Deciding how layers collide is possible using the **Layer Collision Matrix**, which Unity exposes in **Edit > Project Settings… > Physics 2D**.



If the checkbox between layer X and layer Y is checked, it means that gameobjects marked in those two layers can collide with each other. If not, the physics engine will not resolve nor detect their collisions.

Collision layers are extremely helpful when doing active collision detection, in particular with **raycasts** and **overlaps**, which will be explored in a future lecture.