photon

Search

This document is about: **QUANTUM 2**

SWITCH TO

# Callbacks

## Introduction

Collision and Trigger callbacks in Quantum are handled through ***system signals***. There are two steps required to have callbacks executed for any specific entity:

- enabling the specific type(s) of callback to the entity(ies).
- implementing the corresponding signal(s).

Before learning how to do enable callbacks and how to write code, it is important to first understand the different callback types and what causes them to be executed.

## Callback Types

Collisions and triggers start as either a two-entity or an entity-static pair generated by the collision detection step in the physics engine. Depending on the combination of physics components attached to entities, and the value of the trigger property (true/false), the tables below illustrate the types of callbacks that will be executed.

### Entity vs Entity

According to their component composition, physics-enabled entities can be classified as either:

- **Non-Trigger Collider**: Entity with a non-trigger Physics Collider and, optionally, a kinematic Physics Body.
- **Trigger Collider**: Entity with a trigger Physics Collider and, optionally, a kinematic Physics Body.

When a collision pair is composed of two entities A and B, these are the possibly executed callbacks (depending on the group each entity belongs to):

| Entities A x B | Non-Trigger Collider | Trigger Collider | Dynamic Body |
|---|---|---|---|
| Non-Trigger Collider | None | OnTrigger | OnCollision |
| Trigger Collider | OnTrigger | None | OnTrigger |
| Dynamic Body | OnCollision | OnTrigger | OnCollision |

## Entity vs Static Collider

Static colliders, on the other hand, can be either Trigger or Non-Trigger, according to their `IsTrigger` property. These are the possible combinations when the collision pair is composed of an entity and a static collider:

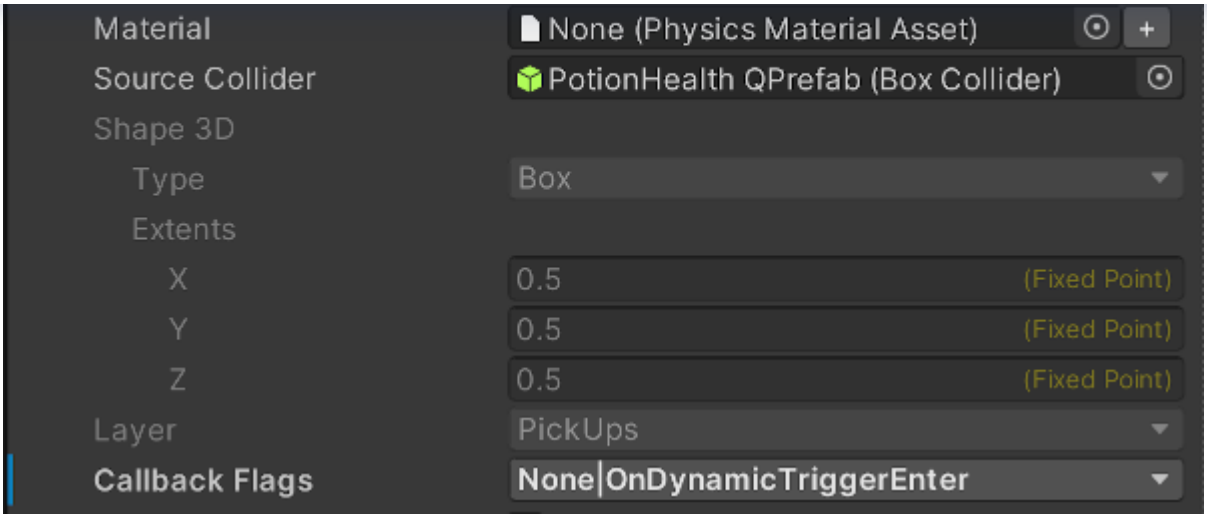
| Components (Entity and Static) | Non-Trigger Static Collider | Trigger Static Collider |
|---|---|---|
| Non-Trigger Collider | None | OnTrigger |
| Trigger Collider | OnTrigger | None |
| Dynamic Body | OnCollision | OnTrigger |

# Enabling Callbacks on Entities

It is possible to control which callback types (and against which kinds of other collider) are enabled for each individual entity. This is done via the *Entity Prototype* in Unity or in code via the *SetCallbacks* function in the physics engine API, which takes the entity and a collision callbacks flag.

## Via Entity Prototypes

The physics callbacks can be set on any *Entity Prototype* with a *PhysicsCollider* (2D/3D).




Setting Physics Callbacks via the Entity Prototype's Physics Properties in the Unity Editor.

Each entity can have several callbacks.

**N.B.:** Enabling the callbacks on an *Entity Prototype* only **sets the callbacks** for that particular entity. You still **have to implement** the corresponding *signals* in code. See the section on *Callback Signals* below for more information.

## Via Code

The callbacks flag is a bitmask and can specify multiple callback types by the use of bitwise operations as exemplified next.

The following snippet enables the full set of OnTrigger callbacks against another dynamic entity (OnDynamicTrigger, OnDynamicTrigger OnDynamicTriggerEnter and OnDynamicTriggerExit).

**C#**

```
CallbackFlags flags = CallbackFlags.OnDynamicTrigger;
flags |= CallbackFlags.OnDynamicTriggerEnter;
flags |= CallbackFlags.OnDynamicTriggerExit;

// for 2D
f.Physics2D.SetCallbacks(entity, flags);

// for 3D
f.Physics3D.SetCallbacks(entity, flags);
```

photon

is not being detected anymore by the physics engine):

- CallbackFlags.OnDynamicCollision
- CallbackFlags.OnDynamicTrigger
- CallbackFlags.OnStaticTrigger

And these are the corresponding Enter/Exit callbacks (that can be enabled independently from the above):

- CallbackFlags.OnDynamicCollisionEnter
- CallbackFlags.OnDynamicCollisionExit
- CallbackFlags.OnDynamicTriggerEnter
- CallbackFlags.OnDynamicTriggerExit
- CallbackFlags.OnStaticCollisionEnter
- CallbackFlags.OnStaticCollisionExit
- CallbackFlags.OnStaticTriggerEnter
- CallbackFlags.OnStaticTriggerExit

Having to enable callbacks on a per-entity basis is an intentional design choice to make the default simulation as fast as possible. Also notice that Enter/Exit callbacks incur in a bit more memory and more CPU usage (compared to basic callbacks), so for the leanest possible simulation you should avoid these whenever possible.

# Callback Signals

**N.B.:** Collision and Trigger callbacks for both *Entity vs Entity* and *Entity vs Static* pairs are grouped into a unified signals API.

These are the 2D Physics signals:

**C#**

```
namespace Quantum {
  public interface ISignalOnCollision2D : ISignal {
    void OnCollision2D(Frame f, CollisionInfo2D info);
```

```
    void OnCollisionEnter2D(Frame f, CollisionInfo2D info);
  }
  public interface ISignalOnCollisionExit2D : ISignal {
    void OnCollisionExit2D(Frame f, ExitInfo2D info);
  }
  public interface ISignalOnTrigger2D : ISignal {
    void OnTrigger2D(Frame f, TriggerInfo2D info);
  }
  public interface ISignalOnTriggerEnter2D : ISignal {
    void OnTriggerEnter2D(Frame f, TriggerInfo2D info);
  }
  public interface ISignalOnTriggerExit2D : ISignal {
    void OnTriggerExit2D(Frame f, ExitInfo2D info);
  }
}
```

And the 3D Physics signals:

**C#**

```
namespace Quantum {
  public interface ISignalOnCollision3D : ISignal {
    void OnCollision3D(Frame f, CollisionInfo3D info);
  }
  public interface ISignalOnCollisionEnter3D : ISignal {
    void OnCollisionEnter3D(Frame f, CollisionInfo3D info);
  }
  public interface ISignalOnCollisionExit3D : ISignal {
    void OnCollisionExit3D(Frame f, ExitInfo3D info);
  }
  public interface ISignalOnTrigger3D : ISignal {
    void OnTrigger3D(Frame f, TriggerInfo3D info);
  }
  public interface ISignalOnTriggerEnter3D : ISignal {
    void OnTriggerEnter3D(Frame f, TriggerInfo3D info);
  }
  public interface ISignalOnTriggerExit3D : ISignal {
    void OnTriggerExit3D(Frame f, ExitInfo3D info);
```

To receive the callback, you need to implement the corresponding signal interface in at least one active system (disabled systems do not get any signal executed in them):

**C#**

```
public class PickUpSystem : SystemSignalsOnly, ISignalOnTriggerEn
{
    public void OnTriggerEnter3D(Frame f, TriggerInfo3D info)
    {
        if (!f.Has<PickUpSlot>(info.Entity)) return;
        if (!f.Has<PlayerID>(info.Other)) return;

        var item = f.Get<PickUpSlot>(info.Entity).Item;
        var itemAsset = f.FindAsset<ItemBase>(item.Id);
        itemAsset.OnPickUp(f, info.Other, itemAsset);

        f.Destroy(info.Entity);
    }
}
```

The code above exemplifies yet another optimization one can use when implementing signals. Inheriting from *SystemSignalsOnly* lets the system handle signals while not needing to schedule an empty update function (that would unecessarily incur into task system overhead).

# CollisionInfo

The signals for `OnCollisionEnter` and `OnCollision` offer up additional information on the colliding entities via the `CollisionInfo` struct.

## Contact Points

The information on the contact points can be accessed through the `ContactPoints` API.

- `Length` : a buffer with all contact points
- `First` : returns the first contact point of the first Triangle

`First` can help saving computations if only one/any contact point is needed, and it does not have to be an averaged one.

`ContactPoints` is also an iterator. When colliding with a mesh, it can be used to iterate through all triangle collisions contact points.

**C#**

```
while(info.ContactPoints.Next(out var cp)) {
  Draw.Sphere(cp, radius);
}
```

*Sphere-Triangle* collisions have a single contact point; therefore, both `Average` and `ContactPoints[0]` will return the same contact. Other types of collision can have more contact points.

## Mesh Collision

When an entity is colliding with a mesh, `Count` and `Average` take all triangle collisions belonging to that specific mesh into account. Instead of receiving a callback for each triangle an entity collides with, these colliding triangles are grouped into a single `CollisionInfo` struct.

In the case of a mesh collision, `info.ContactNormal` and `info.Penetration` will return the average value of that mesh's triangle collisions; the same data available through `info.MeshTriangleCollisions.AverageNormal` and `info.MeshTriangleCollisions.AveragePenetration`.

In addition to the average normal an penetration, you can iterate through each triangle collision and access specific info such as the triangle data itself. `MeshTriangleCollisions` is also an iterator; it can be used to iterate through each triangle's collision data and retrieve mesh-specific collision data.

**C#**

```
    while(info.MeshTriangleCollisions.Next(out var triCollision)) {
      Draw.Ray(triCollision.Triangle->Center, triCollision.ContactN
    }
  }
```

# Misc and FAQ

Useful information when working with collision callbacks:

- If 2 entities have the same callback set, the callback will be called twice - once per entity it is associated with. The only difference between the two calls are the values for *Entity* and *Other* which will be swapped.
- Collisions with triggers and static colliders do not compute the Normal/Point/Penetration data.
- Static Colliders on Unity can have a Quantum DB asset attached to their "Asset" field. Casting it to your expected custom asset types is a good way to add arbitrary data to your static collision callbacks.

Back to top

photon

Gaming Circle

Industries Circle

## Support

Gaming Circle
Industries Circle
Circle Discord
Circle Stack Overflow

## Resources

Dashboard
Samples
SDK Downloads
Cloud Status

## Connect

Public Discord
YouTube
Facebook
Twitter
Blog
Contact Us

## Languages

English
日本語
한국어
简体中文
繁体中文