Physics 2D in Unity

[Moving a GameObject via its Transform](#_p0q2pfubrbc4)

[Setting up a Rigidbody2D](#_tmku9jjfr4ut)

[Physics Material](#_7sjzc1bkd61)

[Understanding Physics](#_2my93hqxb6qb)

[Velocity](#_x8kgsvj0vpk)

[Acceleration](#_w63sqawkzutw)

[Accessing the Rigidbody](#_6t3z5466gzrr)

[GetComponent](#_x51akqzbhprw)

[Manual Linking](#_w0pzxrhx2h63)

[Moving a Rigidbody](#_3qb4lbvzd01)

[Rigidbody2D.velocity](#_fcperxjhas41)

[AddForce](#_g8at20faryva)

[AddForce (ForceMode2D.Impulse)](#_yh8n2m5l61kf)

[Rotating a Rigidbody](#_ufkrff9vzqf)

[AddTorque](#_b0ack0g82l3q)

[AddForceAtPosition](#_i5g34sj0f79l)

[MovePosition](#_hj39gl15a5en)

[MoveRotation](#_6o6oyvz2z477)

[Understanding Rigidbodies](#_krk16bw1f08)

[Update vs FixedUpdate](#_7arqosa57y5g)

[The Rigidbody Body Type](#_n6q4gsb9n038)

# 

# Moving a GameObject via its Transform

In the previous lectures, we have seen how to move a **gameObject** by changing its **position** in a script. This is done accessing transform.position and replacing it with a different value:

|  |
| --- |
| public class Move : MonoBehaviour  {  [Tooltip(“Horizontal speed, in units/second”)];  [Range(-1,+1)]  public float X;  [Tooltip(“Horizontal speed, in units/second”)];  [Range(-1,+1)]  public float Y;  void Update ()  {  // Get inputs from the keyboard/controller  float h = Input.GetAxis("Horizontal");  float v = Input.GetAxis("Vertical");  // Updates the position of the transform  **transform.position += new Vector3(h\*X, v\*Y, 0) \* Time.deltaTime;**  }  } |

Such a method works well, but only allows for very simple moving behaviours.

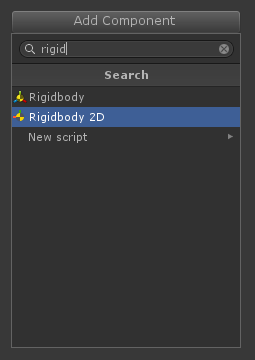
# Setting up a Rigidbody2D

If you need more realistic movements, you can delegate all the movements to a **physics engine**. Its job is to simulate realistic physics, using physical properties such as *velocity*, *acceleration*, *torque* and *friction*. Using a physics engine you will also have access to realistic collisions between objects.

Unity comes with two separate physics engines, one for 2D and one for 3D, accessible using the Rigidbody2D and Rigidbody components, respectively. Despite the similar name, they are different implementations developed by different companies. Some of the features you might have in 2D, for instance, might not be available in 3D.

|  |  |  |
| --- | --- | --- |
| **Type** | **Component Name** | **Physics Engine** |
| 2D | Rigidbody2D | Box2D |
| 3D | Rigidbody | PhysX by NVIDIA |

To delegate the movement of a **2D sprite** to the physics engine, you need to add a component called Rigidbody2D to your game object:



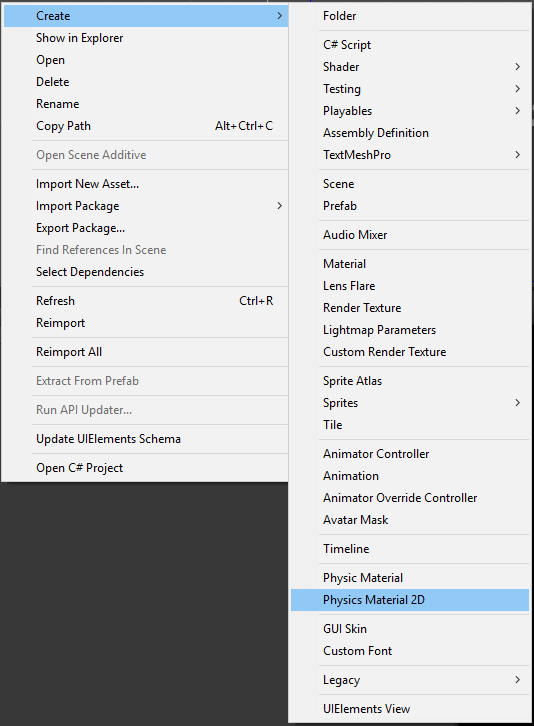
From now on, it will be the Rigidbody2D’s task to update the game object’s transform.position. This means that if your intention is to use realistic physics, you should not move an object *manually* (which means, changing its transform.position directly). All changes to a game object’s position and rotation should be mediated by its Rigidbody2D, for instance using AddForce (for movements), AddTorque (for rotations) or similar methods.

|  |
| --- |
| **⚠ Rigidbody2D and Transform**  When an object is moved using a Rigidbody2D (or Rigidbody), changing its position using transform.position will lead to unpredictable results.  This is especially problematic if you are trying to detect collisions, as moving objects manually could potentially break them. |

## Physics Material

Some properties of the rigidbody, such as its mass and drag, can be controlled the Rigidbody2D component. Other ones, such as its bounciness and friction, passes through an additional asset called a **physics material**.

A physics material can be created by right-clicking in the **Asset window**, then selecting the appropriate option:



Once created, it can be dragged into the **Material** slot of a Rigidbody2D component. Multiple rigidbodies can safely shader the same physical material.

|  |
| --- |
| **💡 Default Physics Material**  If no physics material is selected, Unity will use the *default* one, which can be configured in the **Physics 2D Setting**s (accessible from **Edit>Project Settings>Physics 2D**). |

# Understanding Physics

In order to make the most out of the Unity’s rigidbodies, it is important to have a basic understanding of physics. A gameobject’s position and rotation are stored in its transform, in position and rotation, respectively. All other physics properties are stored and accessible through its rigidbody.

## Velocity

The **position** of a gameobject is updated continuously by its **velocity** (Rigidbody2D.velocity). Velocity is, in fact, a measure of how fast the position changes over time. This is why velocity itself is measured in *metres per second* (m/s)

Following *Newton’s First Law*, the velocity of an object is a property that, without an external interaction (forces, gravity, collisions, friction, …) remains unchanged. Unity simulates air drag and ground friction using **linear drag** (Rigidbody.drag), which slowly reduces the velocity until it reaches zero. Setting linear drag to zero causes rigidbodies to move frictionless.

The speed at which an object rotates is given by its **angular velocity** (Rigidbody2D.angularVelocity), which is measured in *degrees per second* (deg/s). The angular velocity is consumed by the **angular drag**, which plays the same role that linear drag plays for velocity.

## Acceleration

Changing a rigidbody’s velocity directly is possible, although is not advisable as it will result in unrealistic behaviours. Velocity itself is updated using an additional property, called **acceleration**. Acceleration measures how fast the velocity is changing, which is why its unit of measurement is *metres per second, per second* (m/s/s=m/s^2).

Following *Newton’s Second Law*, the acceleration of a rigidbody should be altered using a **force**. A force causes an *instant* change in acceleration, which in turns causes a *permanent* change in velocity. In reality, virtually all interactions between objects are mediated by forces.

The ability of a force to alter an object velocity depends on its **mass** (Rigidbody2D.mass). The mass of an object measure how much “stuff” it contains, and is measured in *kilograms* (Kg). Mass is the property that opposes to acceleration. The same force, applied to two objects with different mass, will result in different changes to their final velocity.

The amount of force an object is subjected to over a period of time is called an **impulse**. Pushing an object with a certain force for one minute, or with half the force for two minutes, yield the same impulse. In Unity, forces are delivered *continuously*, over time, while impulses are single events that affect an object’s velocity *instantly*.

The diagram below shows the main property of moving objects (**position**, **velocity** and **acceleration**) and what is affecting them.



In Unity, you can read and change those properties using the following methods from the following table. Please, note this is not a complete list.

|  |  |  |
| --- | --- | --- |
| **Property** | **Read** | **Change** |
| Position | Rigidbody2D.position | Rigidbody2D.position |
| Rotation | Rigidbody2D.rotation | Rigidbody2D.rotation  AddTorque |
| Velocity | Rigidbody2D.velocity | Rigidbody2D.velocity  AddForce(impulse, ForceMode2D.Impulse) |
| Acceleration |  | AddForce(force)  AddForce(force, ForceMode2D.Force) |

# Accessing the Rigidbody

When a rigidbody is attached to a gameobject, you should change its state using the dedicated methods provided in the Rigidbody2D class. This means that your script has to access the Rigidbody2D component attached to the game object. There are two ways of doing it.

## GetComponent

The first one is to use a method called GetComponent, which allows retrieving another component attached to the same game object:

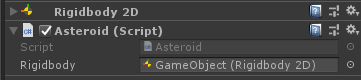
|  |
| --- |
| public class Asteroid : MonoBehaviour  {  void Start ()  {  Rigidbody2D rigidbody = **GetComponent<Rigidbody2D>()**;  }  } |

## Manual Linking

The second method is to expose a public property in the inspector, which can be later connected manually:

|  |
| --- |
| public class Asteroid : MonoBehaviour  {  **public Rigidbody2D Rigidbody;**  } |

In the same way public float Speed; creates an **input field** in the inspector that can hold a number, public Rigidbody2D Rigidbody; creates an **input slot** that can be attached to rigidbody. The connection is done by simply dragging the rigid body into the input slot (below).



The property called Rigidbody in the script now refers to the Rigidbody2D component that is attached to the game object. The second method is faster, but require each slot to be manually linked in the inspector.

|  |
| --- |
| **⚠ None**  If you forget to drag the Rigidbody2D component into the Rigidbody slot, it will appear as **None (Rigidbody 2D)**. That means that the property is not linked to any rigidbody.    Attempting to access the Rigidbody2D will result in an error, and might stop the execution of your script. |

# Moving a Rigidbody

There are several ways in which you can change the state of a rigidbody.

## Rigidbody2D.velocity

The simplest way to initialise the movement of a rigidbody is to change its velocity manually. This is often done in the Start function of a script.

|  |
| --- |
| public class Asteroid : MonoBehaviour  {  public Rigidbody2D Rigidbody;  void Start ()  {  **Rigidbody.velocity = new Vector2(2,0);**  }  } |

Changing the velocity of rigidbody is something that should only be done in rare circumstances, as it “breaks” the physics, by creating unrealistic behaviours. Accessing velocity is often used to bring a rigidbody to a halt, by setting it to Vector2.zero.

|  |
| --- |
| **💡 Velocity**  The velocity of an object is how fast it moves. More technically, velocity is defined as the distance travelled (typically *metres*, *kilometres* or *miles*) in a given unit of time (typically a *second* or an *hour*). This is why velocities are expressed in **metre per second** (m/s), which measure how fast the position of an object has changed over a second.  In Unity, the standard unit of measurement is **unit per second**, where a *Unity unit* generally corresponds to a metre. |

## 

## AddForce

The most common way to move a rigidbody is to apply a **force** to it. A force is a physical quantity that directly affects the **acceleration** of an object. As a result, forces are able to change the velocity of a gameobject. You can imagine a force acting on an object as a push in a certain direction.

|  |
| --- |
| **💡 Acceleration**  Velocities are expressed in **metre per second** (m/s) because they measure how fast the position (in metre) changes over time (one second).  Likewise, accelerations are measured in **metre per second, per second** (m/s/s), which measures how fast the velocity (in metre per second) changes over time (one second). |

You can add a force to a rigidbody by using AddForce. It takes a 2D vector which indicates the direction and strength of the force. To avoid issues with the physics engine, AddForce should be invoked in a method called FixedUpdate.

|  |
| --- |
| public class Asteroid : MonoBehaviour  {  public Rigidbody2D Rigidbody;  void **FixedUpdate**()  {  **Rigidbody.AddForce(new Vector2(2,0));**  }  } |

While moving an object continuously via its transform.position requires Time.deltaTime, adding a force over time does not.

|  |
| --- |
| **💡 Newton**  While velocity is expressed in **unit per second** (m/s), forces are expressed in **Newtons**. A Newton is the “amount of force” (how “hard” you have to push) required to accelerate an object of 1 Kg by 1 m/s. |

A typical use case for AddForce is rocket propulsion:

|  |
| --- |
| public class Rocket : MonoBehaviour  {  public Rigidbody2D Rigidbody;  public float Speed;  void **FixedUpdate**()  {  // Pushes the rocket forward while pressing the spacebar  **if (Input.GetKey(KeyCode.Space))**  Rigidbody.AddForce(**transform.up \* Speed**);  }  } |

The vector transform.up can be used to get the current **heading** of your object. If you multiply it by a certain speed (let’s say 10), you obtain a vector of length 10 that points in the same direction of the rocket’s heading. Feeding it to AddForce will push the rocket forward with a force of 10 Newtons.

|  |
| --- |
| **⚠ GetKey** vs **GetKeyDown**  While GetKey can be safely used in FixedUpdate, GetKeyDown and GetKeyUp cannot! |

## AddForce (ForceMode2D.Impulse)

Forces need to be applied over a long period of time; if you apply a force for a single frame, you might barely see any change. Certain interactions (explosions, kicks, …) require a huge amount of force to be delivered instantly. In that case, it is better to use an **impulse**, rather than a force. While forces act on the acceleration, impulses act directly on the velocity:

Weirdly enough, impulses are delivered using AddForce (with ForceMode2D.Impulse):

|  |
| --- |
| public class Rocket: MonoBehaviour  {  public Rigidbody2D Rigidbody;  public float BoostImpulse;  void Update ()  {  if (Input.GetKeyDown(KeyCode.Space))  Rigidbody.AddForce(transform.up \* BoostImpulse**,**  **ForceMode2D.Impulse**);  }  } |

Because impulses are delivered instantly (and not over a longer period of time), AddForce needs to be called only once. Hence it is generally safe to do this outside FixedUpdate (more on this later).

# Rotating a Rigidbody

There are several ways to rotate a rigidbody.

## AddTorque

The most obvious one is adding **torque**. While force affects an object’s acceleration, torque affects its rotation. Torque needs to be added continuously, so AddTorque must be used in FixedUpdate.

|  |
| --- |
| public class Rocket: MonoBehaviour  {  public Rigidbody2D Rigidbody;  public float Torque;  void FixedUpdate ()  {  float horizontal = Input.GetAxis("Horizontal");  **Rigidbody.AddTorque(Torque \* horizontal);**  }  } |

## 

## AddForceAtPosition

Rotations can also be induced by forces that are applied at a specific point.

If you have a rocket which is rotated using lateral thrusters, you can use AddForceAtPosition.

The method takes a force, and the point where that force is applied (in **world coordinates**). A common solution is to add a public slot to link the script to the thrusters’ Transform. Then, the thruster's up property can be used to get its orientation (which indicates the direction of the force).

|  |
| --- |
| public class Rocket: MonoBehaviour  {  public Rigidbody2D Rigidbody;  public Transform LeftThruster;  public float ThrusterForce;    void FixedUpdate ()  {  float horizontal = Input.GetAxis("Horizontal");  if (horizontal > 0)  **Rigidbody.AddForceAtPosition(**  **LeftThruster.up \* ThursterForce \* horizontal,**  **LeftThruster.position);**  }  } |

# Understanding Rigidbodies

## Update vs FixedUpdate

The state of the game and the physics calculations are executed in two separate processes, which *may* run at different speeds. Scripts are executed using the Update method. Conversely, the physics engine is updated in a different method called FixedUpdate. The Update method is called once every frame. Loosely speaking, the FixedUpdate method is called every time the physics engine is updating the state of the rigidbodies.

It is *generally* safe to change the state of a rigidbody in Update, as long as it is a single, isolated change. For continuous changes (such as adding a force over a long period of time) you will need to use FixedUpdate instead.

In a similar way, it is safe to read the state of the keyboard or mouse in FixedUpdate. However, one-time events (such as *a button has just been pressed* (Input.GetKeyDown), rather than *a button is currently pressed* (Input.GetKey)) need to be executed in Update.

Use the following table to find out in which context inputs and physics methods are safe to be used. Please keep in mind there might be exceptions and special cases.

|  |  |  |
| --- | --- | --- |
|  | Update | FixedUpdate |
| **Input States**  Input.GetAxis  Input.GetKey  Input.GetButton | ✔️ | ✔️ |
| **One-time Events**  Input.GetKeyDown  Input.GetKeyUp  Input.GetButtonDown  Input.GetButtonUp | ✔️ | ❌ |
| **Continuous Interactions**  AddForce (Force) | ❌ | ✔️ |
| **One-time Interactions**  AddForce (Impulse)  velocity | ✔️ | ✔️ |

## The Rigidbody Body Type

When two objects are involved in a collision, their velocities and positions are both subjected to a change. If two rigid bodies collide with each other, both will be affected by the physics engine. However, particular attention should be paid if only one of them is a rigid body. In that peculiar case, resolving the collision in a realistic fashion might be impossible, as the physics engine has full control of only one of them.

One trivial solution would be to add a Rigidbody2D component to every object you can collide with. This is not just expensive, is often counterproductive, since you might want certain objects (such as buildings) to be immovable, but to still produce realistic collisions.

To solve the problem, Unity has introduced the **Body Type** properties, which can be changed to:

* **Dynamic**: the rigid body is controlled by the physics engine and will respond to collisions and external forces;
* **Kinematic**: used for game objects that are moved via code, but that have to interact with dynamic rigid bodies (more on this later);
* **Static**: used for immovable entities that have to interact with dynamic rigid bodies.

Use the following table to decide which body type your rigid body should use, and whether it needs a collider or not. Please, note that the table is just a starting point, and might not work for more complex scenarios. When in doubt, always refer to the Rigidbody2D manual: <https://docs.unity3d.com/Manual/class-Rigidbody2D.html>

|  |  |  |
| --- | --- | --- |
|  | **Interacting with** | |
| **nothing** | **rigid bodies** |
| **Not moving** (environment) | nothing | **Requires**  Collider2D |
| **Moved by code** (platforms, doors, ...) | nothing  **Move using**  transform.position | **Requires**  Collider2D  Rigidbody2D (Kinematic)  **Move using** MovePosition MoveRotation |
| **Moved by physics** (physical objects) | **Requires**  Rigidbody2D (Dynamic)  **Move using**  AddForce  AddTorque | **Requires**  Collider2D Rigidbody2D (Kinematic)  **Move using**  AddForce  AddTorque |