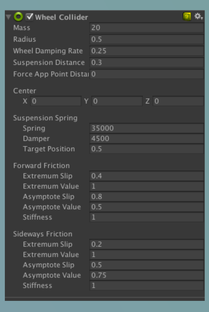
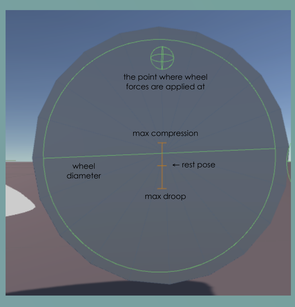
This is how the Unity 5.0 WheelCollider works

(learning vehicles simulation essentials in 7 minutes)

To get a car working in Unity it's useful to understand how the WheelCollider component works internally. Here we attempt to explain that in just a few minutes.



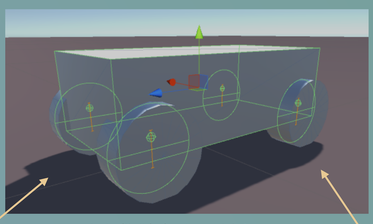
WheelCollider in inspector view, with default settings



WheelCollider in scene view, with gizmo enabled

We'll start from the Editor and then dive right into technical details. So let's say we want to create a regular four wheel car. We will only care about physics in this post, visuals are left aside.

To create this shiny car…



…start with this simple model. Anyway this is how physics engine sees your car.

To get the car moving in Editor check out WheelCollider API:

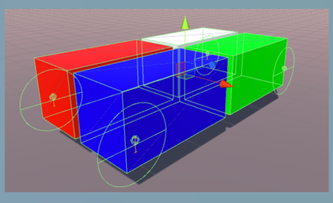
- set motorTorque to accelerate
- set breakTorque to decelerate
- set steerAngle to corner
- there is no engine nor gearbox, you just set raw values

In the inspector view:

- carRoot has Rigidbody with mass 1500
- Cube represents car body, scale it as you like
- 4 WheelCollider objects with default settings

Unity 5 uses the PhysX 3.3 Vehicles SDK that is raycast based. That means a wheel **does not** actually have any shape at all. Instead, there is a raycast done per each wheel along the suspension travel distance (rigidbody local -Y). This is a tradeoff between performance and accuracy. Raycast cars are known to require a really **smooth collision geometry** for a pleasant simulation. Later Unity 5.x will likely add custom casts to raycasts, and that would allow to describe wheel shape better, but that's still a direction for future improvement.

⁂

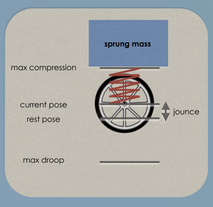


Now let's see what happens under the hood.

PhysX Vehicles simulation is based on a **sprung mass model**. That means, PhysX computes the sprung mass for each wheel (=the portion of the car's total mass it supports). Obviously all sprung masses sum up to the total mass of the car.

← Here we see a visualisation of the sprung mass concept for a typical four wheel car.

The sprung mass model lets to work out the forces individually for each wheel, ignoring the other wheels for the moment. This is physically equivalent to dealing with all wheels at once, but allows to avoid unnecessary complications. Note, this is a **sprung mass but not a sprung weight** model. There is still a weight redistribution effect when some of the wheels are in air or overloaded\underloaded.



(In the Editor, the distance between "max compression" and "max droop" is set by "suspensionDistance". "rest pose" is set by "targetPosition".)

PhysX vehicles' suspension is configured relative to the rest pose. This means we explicitly set the position of the wheel where the spring force is exactly balanced by the sprung weight (while the car is resting on a horizontal flat surface).

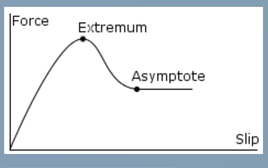
a/ Each frame we try to put the wheel on the ground. If in air, the spring force is zero.
b/ Then, the suspension force is computed:

F = sprungMass * g + jounce * stiffness - damper * jounceSpeed

*(**jounce** is the offset of the wheel relative to the rest pose, **jounceSpeed** is the speed the wheel moved with along the suspension travel direction, **stiffness** and **damper** are spring parameters)*

***sprungMass * g** is the force the spring produces at rest on horizontal surface; if not, a raycast would move the wheel along the suspension travel direction and that would compress or elongate the spring thus affecting **jounce * stiffness** and **damper * jounceSpeed** components.*

Tire forces are also calculated each frame after the corresponding spring force is computed.

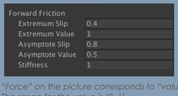


Tire simulation does not use dynamic and static friction from PhysicMaterial. The slip based model is used instead. *(Ground surface is typically emulated by changing friction "stiffness"; that would simply scale the friction.)*

Why using slips but not regular physics friction? Turns out this way we get better tire simulation: with a real tire, grip seems to depends on how big the slip values are.

Increase forward friction to get more grip.

Increase sideways friction to reduce drifting in corners.



"Force" on the picture corresponds to "value" in inspector. The range for the value is (0, 1)

After all forces have been computed, we simply apply them to the car's body. One important thing to remember is that forces are applied at "forceAppPointDistance" up the suspension travel distance **relative to the base of the resting wheel**. There is no default value that works for all configurations, but avoid having forceAppPointDistance = 0, as that can cause unstable simulation.

⁂

One more thing… After you got your car working reasonably well, you might want to make it so that the visual wheels actually follow physical wheels. Use WheelCollider.GetWorldPose to get the wheel pose the simulation is using. That includes position as well as rotation in world space, ready to assign to your visual wheel's transform.

```
Quaternion rotation;
Vector3 position;
physicsWheel.GetWorldPose(out position, out rotation);

visualWheel.transform.position = position;
visualWheel.transform.rotation = rotation;
```