Documentation - Easy Bike System

1 - Overview Introduction to the Package

This free package provides everything you need to set up two-wheeled vehicles (bicycles, scooters, motorbikes) with highly customizable setups, and with the new addition of inverse kinematics (IK) support for Rigged characters

- Works using Unity's built-in components.
- Highly customizable handling for bicycles, scooters, and motorbikes.
- Includes example vehicles, prefabs, and a demo scene.
- Fully compatible with Unity's Animation Rigging package (Dependency for IK).

Simply add your own vehicle and character models to the system, and you'll have custom 2-wheeled rides up and running in minutes.

Documentation regarding all the **provided assets**, **settings**, **configurations**, **extensions**, and **usage** can be found below.
The 3D assets provided in the pack are usable. for private and commercial projects. based on the
CC Sharealike 4.0 International license by **RayznGames** (Package creator) Rayzn - Sketchfab

Thanks for downloading the Easy Bike package!

2. Package Contents

- **BicycleSystem**– Complete 2 wheeled vehicle handling system.
- **Friction Curve Visualiser.**
- **IKBikeRig Script** – Inverse Kinematics for characters (Animation Rigging package).
- Playable **Demo Scene** – With working prefabs and test environments
- Free **Example Models & Prefabs** – Bicycle, scooter,and motorbike with rigs.
- Free **Example Shaders and FX materials.**
- Simple **3rd-Person Camera Controller** – With collision detection to prevent clipping.
- Greyboxing Tiles Shader – For prototyping environments.
- Compatible with **URP** and **HDRP** render pipelines

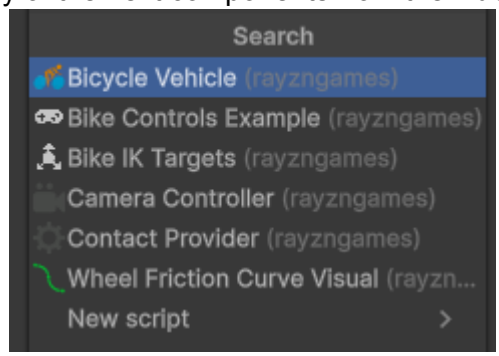
3. Dependencies

- **Unity's Animation Rigging Package** (required for IK functionality).

4. Quick Start

- Import the package into your Unity project.
- Will prompt you to Install dependencies automatically
- Open the Playable Demo Scene. (URP or HDRP)
- Press Play - You'll find a ready-to-drive 2 wheeled vehicle.

Add any of the next components from the **AddComponent** menu



Inputs: (Accessible through the `BicycleVehicle` script)

- For steering and acceleration, `horizontalInput` and `verticalInput` are used as axes (`-1 to 1` range), allowing compatibility with both Legacy (Old) and the new InputSystems. So you can control the vehicle with your own custom inputs
  e.g. **bike.horizontalInput / bike.veticalInput**

- Braking is handled via the public `Braking` boolean – applies `brakeForce` to both wheels, while cutting off power.

- Access to the current **GroundedState** of the vehicle through the accessible method **OnGround()**

- Modify access to the vehicle control state with the method **InControl(bool state):**
  - **InControl True:**
    - Sets **RigidBody** Constraints to **Freeze rotation** in the **Z** axis
    - Allows the bicycle to respond to the Inputs provided
  - **InControl False:**
    - Frees all Rigidbody Constraints. giving full control of the notation of the object to the rigidbody and the physics system, or the developer.
    - Does not allow the bicycle to respond to the inputs provided

    Access the current inControl state by reading directly from the public **isInControl** property getter.

*(**Input Examples** can be found under **Code Examples** below)*

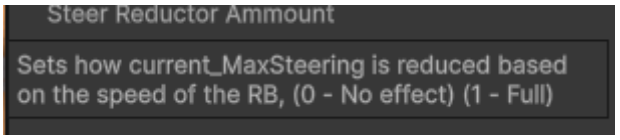
5. How It Works

5.a - Bicycle System - Inspector, Runtime Modifiable, and accessible Read-Only properties.

Inspector Properties

These next set of properties are accessible through the inspector, that let you customize the handling, physics, and feel of your bike. By adjusting them, you can simulate anything:
(**e.g.** a rusty old bicycle, mountain bicycle, kids-Bike, a scooter, miniBike, motocross Bike, motorbike, high-performance superbike, or any 2 wheeled vehicle)

(Hover on top of any property to get ToolTip information regarding its setup)



Power & Braking

`motorForce (float)`

What it does:

The forward driving force applied to the rear wheel.
Power is cut off whenever braking is active.

Examples:

- `Low MotorForce` → Bike accelerates slowly (good for old rusty bikes or children's bicycles).
- `High MotorForce` → Bike accelerates more quickly (motorbike, racing bike). (*Use with higher RB mass values)*

`brakeForce` (`float`)

What it does:

Definess the overall braking strength applied to the wheels.
Independent of `MotorForce` — higher values = stronger braking, or wheel locking

Examples:

- `Low BrakeForce` → Bike takes longer to slow down (coasting feel, casual ride).
- `High BrakeForce` → Bike stops quickly (racing bikes, arcade handling).

`frontBrakePower / rearBrakePower` (`float`)

What it does:

Defines the brakeforce applied to each of the wheels.(*acts as a scaling factor)*
Independent of `Brakeforce` — higher values = stronger braking, or wheel locking

Examples:

- `Low BrakePower` → Wheel applies less **brakeForce** (coasting feel, casual ride, reduced pad contact).
- `High BrakePower` → Wheel applies higher **brakeForce** up to brakeforce value (racing bikes, oil brakes).
- Can be dynamically adjusted at runtime for custom effects and behaviours (Dynamic Brake forces.., brake pads wear, individual custom brake performance)

`COG` (Center of Gravity Offset) *(`Vector3`)*

What it does:

Adjusts the center of mass of the bike's Rigidbody.
Used to make the bicycle more stable or more "tippy."

Examples:

- `COG lowered (negative Y)` → Bike feels more stable, less likely to tip.
- `COG raised (positive Y)` → Bike feels less stable, tips more easily (useful for stunt bikes).

Steering

`MaxSteeringAngle` *(`float`)*

What it does:

The maximum angle (in degrees) the handlebars can rotate/turn.

Examples:

- `Low MaxSteeringAngle (e.g., 20°)` → Bike turns gently, cannot turn sharply (good for racing at speed).
- `High MaxSteeringAngle (e.g., 50°+)` → Bike can make sharp turns (good for casual cycling or trial bikes).

`SteerReductorAmount` *(`float`)*

What it does:

Dynamically reduces the maximum steering angle as vehicle speed increases.
Prevents unrealistic sharp turns at high speeds

Examples:

- `Low Reduction` → Steering remains responsive even at high speeds (arcade-like handling, less realistic)
  Note. *Requires knowledge over WheelColliders frictions, tweaking the parameters, and wheel friction management is expected to be handled by the developer, and can lead to unexpected behaviours.*
- `High Reduction` → Steering becomes stiffer at high speeds (realistic handling, prevents wipeouts).

`TurnSmoothing` *(`float`)*

What it does:

Controls how quickly the handlebars reach the target steering angle.

Examples:

- `Low TurnSmoothing` → Handlebars move slowly, steering feels heavy (like a rusty, stiff bike).
- `High TurnSmoothing` → Handlebars snap quickly, steering feels sharp (sports bike feel).

Lean / Tilting

`MaxLeanAngle` (`float`)

What it does:

The maximum tilt (in degrees) the bike can reach while turning.

Examples:

- `Low MaxLeanAngle (e.g., 10°)` → Bike barely leans, feels stiff and heavy.
- `High MaxLeanAngle (e.g., 40°+)` → Bike leans aggressively into turns (sporty feel).

`LeanSmoothing` (`float`)

What it does:

Controls how quickly the bike reaches its max lean angle.
Higher values = faster leaning response, lower = slower, smoother lean.

Examples:

- `Low LeanSmoothing` → Bike takes time to lean into turns, feels sluggish but stable.
- `High LeanSmoothing` → Bike snaps into leans quickly, feels agile but twitchy.

Input & modifiable - Methods & Runtime Properties

The following properties and methods can be accessed at runtime for input handling, and extending the system with custom logic and custom inputs.
These values are not updated by the **BicycleSystem** script. and require the developer to use their functionality in order to provide the inputs and extend the system as needed.

`horizontalInput : (float)`

Description:
Represents the steering axis of the bicycle. Values range from **-1 to 1**.
Negative values correspond to left steering, positive values to right steering.
This is a direct input value provided by your custom input system.

Usage Example:

- Mapping joystick or keyboard input to bicycle steering.
- Driving AI logic for automated steering behavior.

`verticalInput : (float)`

Description:
Represents the acceleration axis of the bicycle. Values range from **-1 to 1**.
Negative values represent reverse or backward pedaling (if implemented), positive values represent forward acceleration.

Usage Example:

- Feeding throttle or pedaling controls to the vehicles.
- Adjusting acceleration dynamically.

`Braking : (bool)`

Description:
A toggle that determines if braking force is applied to both wheels.
When enabled, it cuts motor/acceleration power and applies the configured ***`brakeForce`*** multiplied by each wheels, ***BrakePower***

Usage Example:

- Binding to a button or trigger input for brake control.
- Implementing automatic braking under scripted conditions (e.g., cutscenes, collisions).

OnGround() : Method (bool : getter)

Description:
Returns whether the bicycle wheels are currently in contact with the ground.
Useful for detecting airborne states such as jumps or falls.

Usage Example:

- Restricting acceleration and steering to when the bicycle is grounded.
- Triggering special effects when the bike becomes airborne (e.g., enabling air controls).

InControl(state) : Method (bool : setter)

Description:
Modify the vehicle control state with the method **InControl(bool state):**

- **InControl True:**
  - Allows the bicycle to respond to the Inputs provided
- **InControl False:**
  - Does not allow the bicycle to respond to the inputs provided

Usage Example:

- Toggling control during cutscenes, scripted events, or when switching AI/player control.
- Enabling ragdoll-like behavior by relinquishing input control.

isInControl : (bool : getter)

Description:
Exposes the current control state of the bicycle.
Returns **true** if the system is actively listening to inputs, **false** if left to raw physics.

Usage Example:

- Checking control state before applying new inputs.
- Knowing what vehicle is currently in control.

WheelNormalizedSlipInfo() : Method (Debug)

Description:
Logs to console the normalized slip info of the WheelColliders front and rear, for debugging friction parameters using a normalized slip info from 0-1, (0 to 1 max grip, above 1 slip is being generated)

Usage example:

- Use it to understand the limits of friction and when it starts to slip to get feedback regarding what's needed to change in the friction Curves.

`ConstrainRotation(state) : Method (Bool : setter)`

Description:
Constraints the rotation of the RB in the Z component, or frees all the constraints if set to false.

- **ConstrainRotation True:**
  - Sets **RigidBody** Constraints to **Freeze rotation** in the **Z** axis
- **ConstranRotation False:**
  - Frees all Rigidbody Constraints. giving full control of the notation of the object to the rigidbody and the physics system.

Read-Only Properties (Not in the inspector)

The following properties can be accessed at runtime through scripting for debugging, UI display, or for extending the system with custom logic. These values are updated continuously by the **BicycleSystem** script and cannot be modified by the developer.

`currentSteeringAngle : (float)`

Description:
The actual steering angle currently applied to the vehicle's front handles.

- Expressed in **degrees**.
- Positive and negative values correspond to left/right steering.
- This is the **final steering value in use**, after any dynamic reductions have been applied (see ***`current_maxSteeringAngle`***).

Usage Example:

- Creating on-screen UI indicators (handlebar angle).
- Feeding custom animations or procedural visuals (e.g., rider leaning into turns).

`current_maxSteeringAngle : (float)`

Description:
The **dynamic maximum steering angle** allowed at the current speed.

- Calculated using the vehicle's **Rigidbody linear speed**.
- Reduced automatically by the `steerReductorAmount` setting in the inspector
- Ensures vehicle stability and realism at speeds by clamping `currentSteeringAngle`.

Usage Example:

- Visualizing steering limits in debug tools.
- Adjusting rider animation intensity relative to maximum possible steering.
- Controlling when steering is equal to max steering during long periods for custom behaviour.

`currentLeanAngle : (float)`

Description:
The lean angle (tilt) currently applied to the bike during turns.

- Expressed in **degrees**, with `-45°` to `+45°` as the typical range.
- Negative values = leaning left, positive values = leaning right.
  Updated dynamically as the bike responds to steering and speed.

Usage Example:

- Driving Rider Character tilt animations.
- Visual feedback (HUD lean angle display).
- Physics or VFX extensions (e.g.: sparks if lean angle exceeds a threshold safe range).

`currentSpeed : (float)`

Description:
The current forward speed of the bike's Rigidbody in **meters per second (m/s)**.

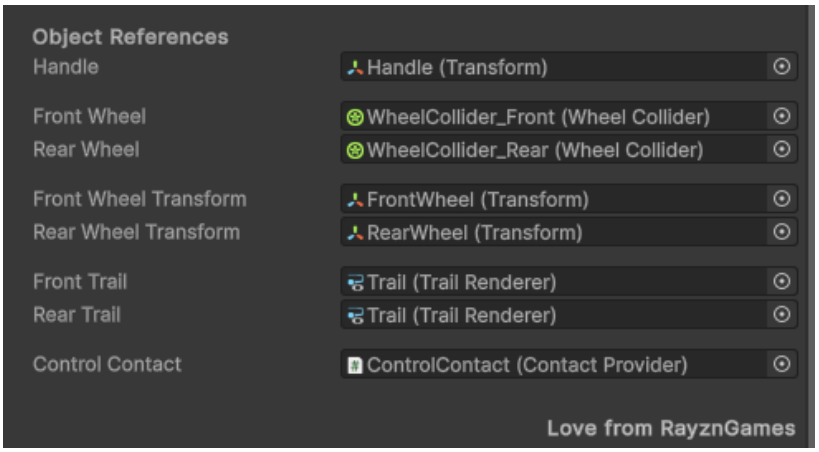
- This is the speed value extracted and exposed by the system.
- Can be converted to km/h or mph for UI.
  Km/h: multiply by `3.6`
  Mph: multiply by `2.23694`

Usage Example:

- Speedometer UI.
- Triggering effects or gameplay logic at certain speeds (e.g., camera shake, wind VFX).
- Adjusting rider animation.

Object references:

All the important aspects and references for the vehicle are located at the bottom of the system.



Handle: Holds a Transform reference to the handle holder (**Handle)** of the entire front fork that holds: handles, and the front (**visual**) tire 3D model.

Front Wheel Reference to front wheel collider
Rear Wheel Reference to rear wheel collider

Front Wheel Transform: Reference to front wheel visual (*Located under handle)*
Rear Wheel Transform Reference to rear wheel visual

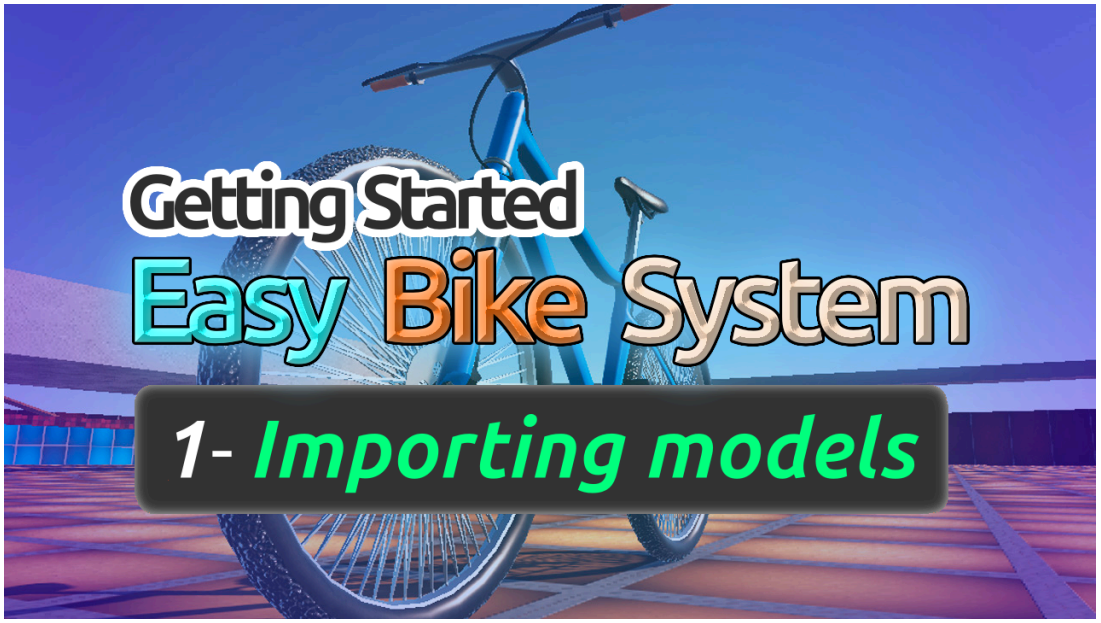
Front Trail Contact: The trail component of the front wheel
Rear Trail Contact: The trail component of the back wheel
*(**TrailsContact** prefab hold the contact providers, and the smoke ParticleSystems **TrailsContact** are children of the WheelColliders)*

Control contact - Trigger at the wheelbase of the bike, returning general contact with ground.

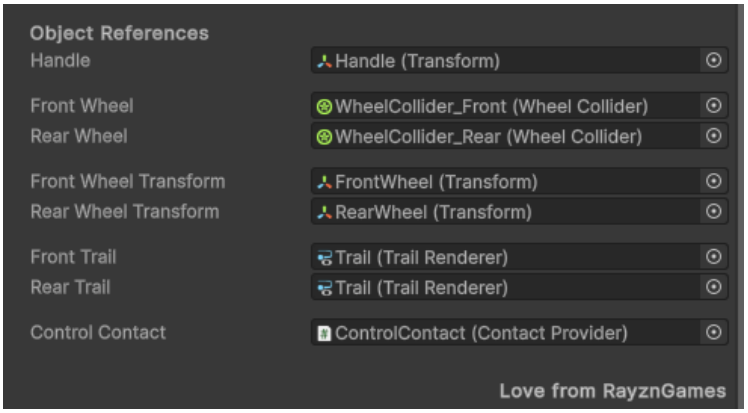
Model Hirearchy: (Applies to any 2 wheeled vehicle)

Adding your 2 Wheel Vehicle Models:

Easy Bike System - Tutorial 1- Getting Started + Importing Models

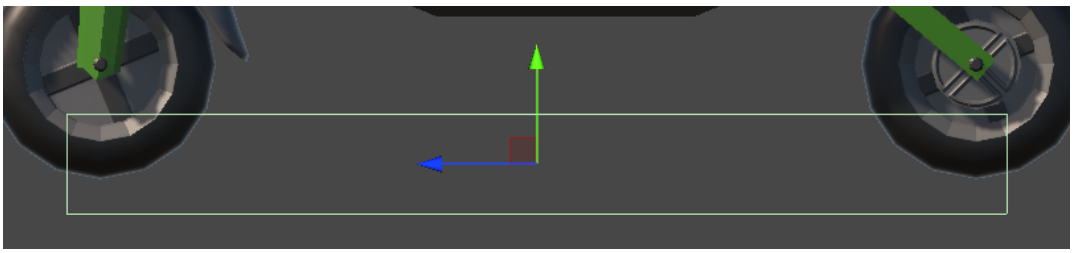


FrontWheel (3D model) should be parented under **Handle.**
Handle should be the assigned transform in **BicycleSystem** < **Handle**
*(See **Handle Pivot** position and rotation in **Example Prefabs**)*

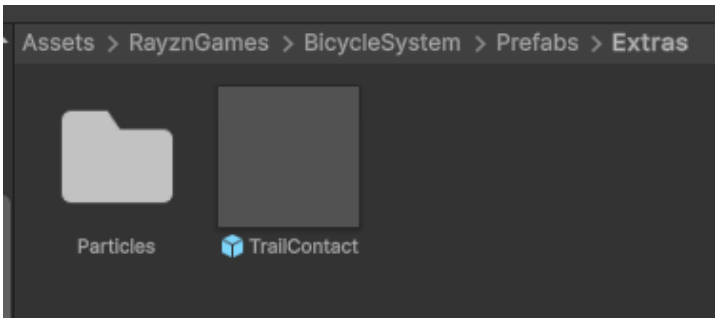


Modify: values like **motorforce**, **brakeforce**, **maxSteeringAngle**, **maxleaningAngle**, **turnSmoothing**, or **leanSmoothing**, modify the **weight of the rigid body,** and adjust the **size** and **wheelbase**, to achieve different bike configurations and behaviors. Allowing you to make multiple vehicles be unique

Control Contact: A Collider trigger that encompasses the entire wheelbase of the vehicle, and provides information regarding the current contact state with the ground. (Less janky than WheelCollider contacts, or ContactProviders) Useful for many reasons. Requires direct handling for it to have an effect



TrailsContact Prefab
The system comes with a prefab that contains: the Trails, a ContactProvider, and a Particle System**. (*For the system to display slip related FX*)**
Now trails are slip dependent, which means that you're required to lose traction and skid, to generate rubber trails and smoke particles. Find them at:
Assets < RayznGames < BicycleSystem < Prefabs < **Extras**



Wheel colliders:

Should be placed in a separate game object from the wheel's visual transforms.
Make sure the orientation of the vehicle models are facing **Forward** (**+Z**)
WheelColliders are required to have at least a bit of suspension travel, (***Suspension distance)*** to correct;y work, if not forces will not be properly calculated and transferred to the RigidBody.
When placing the wheelcolliders make sure they are parented to the vehicle

Keep in mind when tweaking frictions or suspensions, both affect the overall grip of the vehicle equally since both are responsible for the transfer of forces to the rigidbody.

To avoid strange or impossible suspension behaviours it is recommended to use a cylindrical collider approximating 75-80% of the circumference of the wheel, to avoid extreme suspension punches and contact inconsistencies. Careful use of Collision Layers with these Cylinder colliders is important.
*(**Technical limitations of wheelcolliders** are detailed in an **Appendix** at the end in this same document.)*

Suspensions:

*(*WheelColliders suspensions in Unity can be easily tweaked using what is known as:
[Spring & Damper to mass ratio] multiplied by the weight of the vehicle)
What this means is that we can perform a set of calculations to find the usual or the average suspension settings for one axle. example:

Extract Ratios (Default) Unity
FInd unity's WheelColliders parameters (Configured for cars) and extract the ratios of damper and spring for a car of 1500 Kg of mass
Spring- ratio (default):
35000 / 1500 = **23.3335000 per KG**

Damper ratio (default):
4500 / 1500 = **3.04500 per KG**

Sport bike example (Rear Suspension):
Find the mass of the system

- Mass = 200 kg(Vehicle) + 80 kg (Rider) = **280 Total System KG**

FInd the weight distribution bias of the system (Front / Rear weight)

- Front supports 40% of the mass while rear stays for 60% of the load
- Average the weight based on the system bias:
  Example :60% of 280 = 280 kg * 0.60 = **168kg**

FInd the Average spring rate for this kind of vehicle axle suspension (Rear)

- (Average) Rear spring rate = 100 N/mm = **100,000** N/m (since 1 mm compression = 100 N).

Now simply compute to find the spring needed to support that mass:

- Spring-to-mass ratio:
  100,000 N/m / 168 kg **≈ 357** Spring Coeff

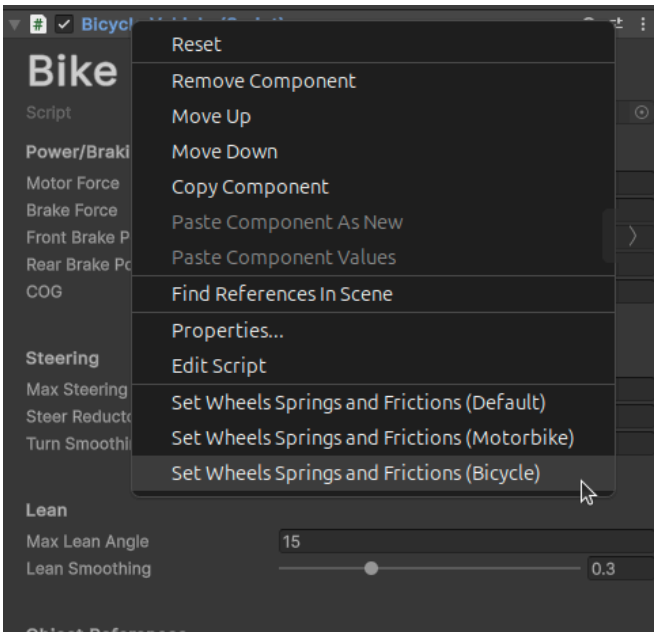
Spring force and damper should be adjusted depending on the RigidBody mass and weight bias of the system..
Make sure to configure **proper Dampers**, to ensure no pogo or jumpy behavior on the suspension.

Damper-to-mass ratio:
Harder to define exactly, but usually tuned around **10–20% of critical damping**.
Unity's car ratio (3 damping coefficient. per kg) would be underdamped for a motorcycle.
Realistic bike setups often land closer to **15–30 Coefficient. per kg**, depending on the use case (dirt vs. road).
However lots of tweaking is expected to make sure no pogoing happens and the stability of the ride.

I have taken the time to ensure we have a broad spectrum of example valid spring and Damper Ratios *(Keep in mind this information has been **extracted** using **averages** values **of***

suspension *spring Kn/m forces, travel Distances ,rebound expectancy for dampers,, system weights, and weight bias for each kind of vehicle (front / rear).*

Vehicle Suspension	Spring	Damper
Car (Unity Default)	23.33 * Mass (KG)	3 * Mass (KG)
Motorcycle Rear	300–400 * Mass (KG)	15–30 * Mass (KG)
Motorcycle Front	450–800 * Mass (KG)	25–40 * Mass (KG)
Bicycle Rear	219–400 * Mass (KG)	11.7–30 * Mass (KG)
Bicycle Front	256–500 * Mass (KG)	16–30 * Mass (KG)



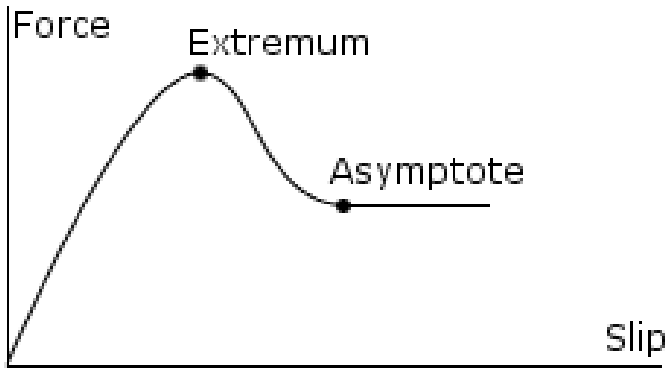
You can **Auto-Setup** proper (Weight Calculated) Suspensions and decent default friction values from the inspector right clicking in the **Bicycle Vehicle Component** and choosing the type that fits your vehicle suspensions.

These are similar defaults you'll find here. You can later tweak at will.

Friction parameters and Friction curves.

In order to simplify a relatively complex concept like friction curves I'm going to outline the key points and takeaways for understanding friction management, which is one of the most important of the aspects handling wise.

Here's the example curve that will allow us to understand how do the friction parameters like **ExtremiumSlip, ExtremiumValue, AsymptoteSlip, AsymptoteValue** and **Stiffness** relate to one another



In order to more easily visualise these parameters and see them change the curve in realtime Check The ***Wheel Friction Visualiser Script,*** which allows you to see each of the friction curves change while you edit the parameters.

The curve works the next way, it uses 2 values to define each point in the graph, and is composed of 3 points,

The Initial point - (Base) at 0 Slip and 0 Force.
It is the Default initial evaluation of the FrictionCurve at rest

The second Point - (Extremium) At **extremiumSlip** ,and **extremiumValue.**
Is the point at the Peak of friction, at this point you have reached the absolute grip limit,
If the curve evaluates higher slip rates, friction starts to decrease as more slip is generated and falls towards asymptote

The Third Point (Asymptote) At **asymptoteSlip,** and **asymptoteValu**e.
Is the point of lowest friction when slip rate is high, this is a useful value pair to control the grip level at high slip values. Represents when the rubber gives up all grip and wheels slide. (Keep in mind when sliding there's still frictions involved, but they are lower)

Stiffness:
Works as a multiplier for the frictionCurve, and I personally recommend using it really carefully since it can lead to solver inconsistencies if values are slightly high. I recommend to better modify **asymptote** and **extremium** and keep stiffness at 1 if possible as a value to reduce and scale friction properly to simulate other surfaces

ExtremiumSlip / ExtremiumValue

ExtremiumSlip should always be below **asymptoteSlip**,
ExtremiumValue should be greater than **asymptoteValue**.

A higher **extremiumSlip**: Means that it takes a lot of slip to reach peak of friction force
A high **extremiumValue** Defines a high grip or high friction when slip rate is at that Extremium threshold.

Combining a **High ExtremiumSlip** with a **High ExtremiumValue**, creates what could be considered Really Grippy tyres, when slip forces are within (Base and Extremium)
This is because we are setting the limit of really high grip (**extremiumValue**) at a really High slip rate wich means that it requires a lot of external slip forces to reach the peak of grip which is really high, creating a **really smooth transition** from base to extremum and **keeping a consistent grip** even at high external slip forces **for longer**

AsymptoteSlip / AsymptoteValue

AsymptoteSlip should always be greater than **ExtremiumSlip**
AsymptoteValue should always be below **ExtremiumValue.**

A high **asymptoteSlip** creates a smoother loose of grip transition from the peak of grip (Extremum) to the lowest grip (asymptote)
A high **asymptoteValue**, provides more grip when the vehicle wheels are technically losing traction because of high external slip forces
A low **asymptoteValue** generates less grip or friction when external slip forces are high or reaching **asymptoteSlip**.

Combining a **high asymptoteSlip** with a **medium asymptoteValue** (Compared to extremium) creates what could be considered a **powerslide mechani**c,when Slip exceeds Extremium and is between (**Extremium and Asymptote)** This is because when slip forces exceed extremum the curve instead of plummeting towards the minimum grip at asymptote sharply (what could be considered Drifting) instead creates a more smooth curve where friction falls smoothly within a longer range towards the lowest grip **Asymptote**.

Providing a range of smoothly lessening grip or a powerslide range that allow for some control since it still provides reasonable lessening grip at high slip rates..What can make for great arcade experiences.

Forward and Sideways frictions

Forward : Is the longitudinal friction parameters , and refers to the forward acceleration or deceleration friction created by the tyres when rotation changes are applied, This friction is helpful for **Acceleration**, or **braking** performance.

Sideways Is the lateral friction parameters applied to the wheel, when the tyre scrubs the

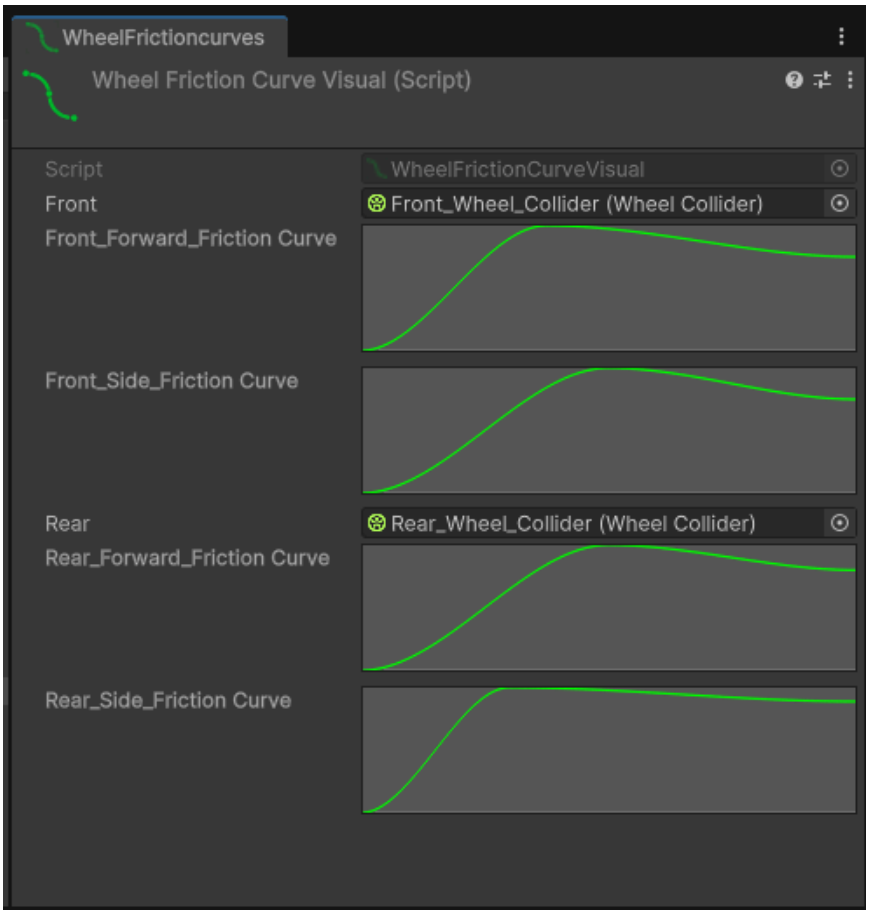
ground with any steering input., A high grip sideways friction Curve will avoid understeering in the front wheel, and give more stability on the rear wheel at fast speed turns.

Wheel Friction Curve VIsualiser

The wheel Friction curve Visualiser is a simple script that executes in Edit Mode,when assigned to an object in the scene.

It gathers the friction parameters from the WheelColliders that are assigned to the script And generates visual representations in editor and at runtime of each of the gathered Forward and Sideways FrictionCurves, for each wheel.

These curves serve as a display reference to understand what changes are made to the friction curves and the effects of each change visually.



The package includes a property attribute **[CurveSize(Height)]** to display Individual curves with more height inside the inspector for easier visualisation. You can use this ***Property attribute*** on AnimationCurves in your own scripts like: **[CurveSize(100)]** if using rayzngames namespace

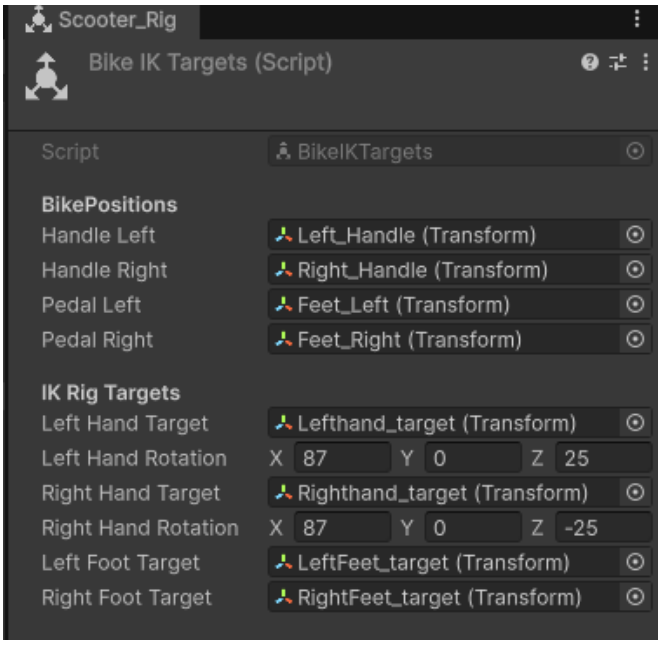
5.b - IK Bike Rig (Bike IK Targets Script)

Easy Bike System - Tutorial 2- Inverse Kinematics Rig & Advanced Features

Requires the **AnimationRIgging** package for you to use it. Controls IK targets for rigged character hands and feet.

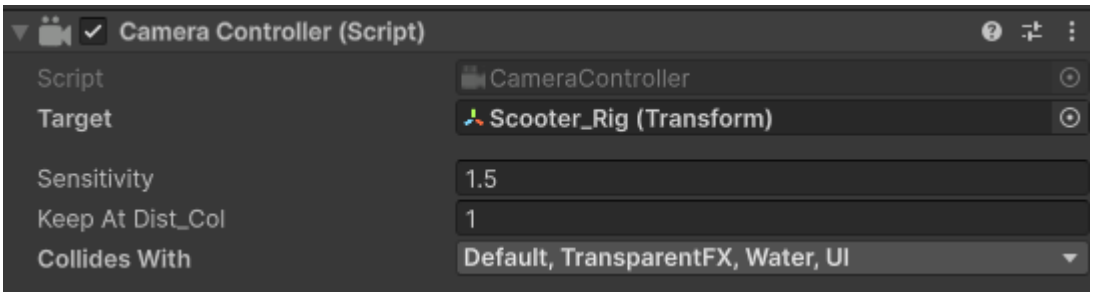
- Targets:
  - **BikePositions** → Predefined bike positions (left & right handles) and pedal positions (left & right pedals) for the IK targets to track.
  - **IK Rig Targets** → The rigged character's IK bone targets: (Created by *Animation RIgging Package*)
    (e.g: **leftHand Target, rightHandTarget)**
- When pedals are animated, the foot targets move accordingly, keeping the rider in sync with the bike.

(The IK rig System is a unity native component, and the IK rig setup is completely dependant on you the developer)



5.c - Camera Controller

- Simple third-person camera with built-in collision detection.
- Works by raycasting backward and adjusting position to prevent geometry clipping.
- Select collision layers in the **CameraScript inspector**.



6 - Code Examples:

Make sure you're **using rayzngames;** namespace in the top of your script when interacting with the vehicle inputs and outputs.

```
using rayzngames;
using UnityEngine;
```

When gathering the vehicle component, you can assign it to a public field in the inspector , grab it from the **Awake**() method, or gather it at **Start**().

```
0 references
public class BikeControlsExample : MonoBehaviour
{
    4 references
    public BicycleVehicle bicycle;
    // Start is called once before the first executio
    0 references
    void Start()
    {
        bicycle = GetComponent<BicycleVehicle>();
    }
```

For Controlling the vehicle Inputs you can do so with any input methods you have at your disposal, the modularity it comes with offers the possibility of using the ***new InputSystem,*** the (Legacy) ***Old InputSystem***, or your own ***custom InputSystem.***
(Here i am using the Old input System, by using the **Horizontal** and **Vertical** Axis accordingly, and as well the **GetKey** for brake handling)

```
0 references
void Update()
{
    bicycle.verticalInput = Input.GetAxis("Vertical");
    bicycle.horizontalInput = Input.GetAxis("Horizontal");
    bicycle.braking = Input.GetKey(KeyCode.Space);
}
```

You can extend the functionality of the system like so: with your input system of choice, completely modular and scalable to your project needs.

Here's a simple naive example of how you can extend its functionality (Commented)

```
//Extending functionality
if (bicycle.OnGround() == false) { controllingBike = false; }

//Landing Controls
if (Input.GetKey(KeyCode.E)) { controllingBike = true; }
bicycle.InControl(controllingBike);
}
```

(*What this does is ensure that the system is not controllable when Airborne, and expects the player to press the landing Key (E) to keep it under control,,when the bike touches the ground during this frame, or the next frame*)

This example input script can be found inside: -
Assets < RayznGames < BicycleSystem < Scripts < **Examples&Extras**

7 - Simple System Extensions:

The system remains open to be extended with functionality around it, I will go over some of the possibly more demanded features and the reason the system will not implement those is that it remains important to keep it as simple as is, while open for any extra functionality.

As mentioned the system has been designed for new and experienced developers to be able to use it in the broadest possible amount of scenarios. And therefore to be extended and blended with custom functionality that is not built-in with the system. making it as versatile as a plug and play asset, and a deeply extendible feature with more advanced mechanics

Gearing System:

MotorForce is what drives forward the traction wheel.

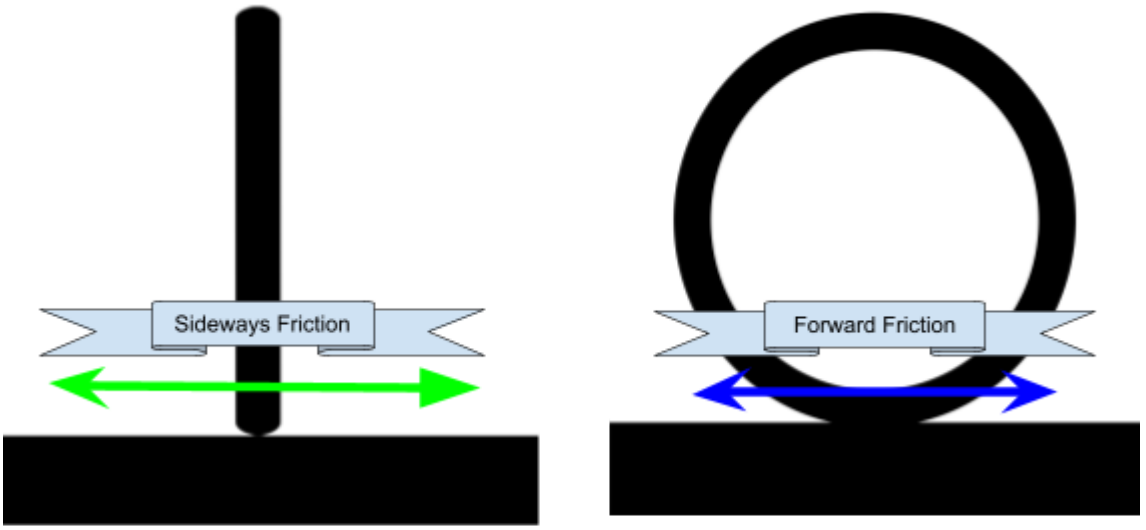
If your goal with the system is to create a high performance motorbike, you can create an **external "Gearing module"** with a float,representing a global gear ratio, and an array of floats, the array size representing the number of gears, and the array values representing the ratio of each gear.
You can then create an "Engine Power Curve" using an AnimationCurve and evaluate the curve at runtime using a HorsePower float, and RPM as time, and your current and main gearing ratios to provide the exact motorforce at that gear and moment along the engine power curve.

Airborne Controls:

In order to allow the vehicle to roll and make stunts while on air, it is possible to add external controls to drive forces to the Rigidbody while on air, making it rotate with physics driven forces, or apply carefully controlled transform rotations to the object when **freeze rotation Z** is disabled.

8- Appendix: Technical Limitations of Unity's WheelCollider Physics

The **WheelCollider** in Unity is a specialized physics component designed to approximate the behavior of a tire in contact with a surface. While useful for rapid prototyping and general vehicle setups, it comes with inherent limitations due to its design, simplifications, and reliance on Unity's (Nvidia) PhysX integration. The following points summarize these technical constraints:



If the wheel is properly straight up, the friction values are calculated properly, and the shown forward and sideways axes are used in accordance as the pictures above explain. Anything outside of this is excluded from the simulation and not noticeable if you're making vehicles, with 4 wheels, whereas no wheel is going to tilt in any direction. Therefore firctions will stay consistent for the most part

1. Simplified Contact Geometry

- The **WheelCollider does not simulate an actual 3D tire**.
- Instead, it performs a **raycast** from the wheel's position toward the ground.
- Contact with the surface is represented by a single point, with no actual volume-based geometry, meaning effects like tire deformation, multi-point surface contact, or tread behavior are not captured.
- **Uneven or small-scale surface details** are not resolved accurately Leading to artifacts.(e.g., curbs, bumps, stairs, small ground details, small objects.)

---

2. Limited Suspension Model

- The suspension is modeled as a **single spring-damper system** acting along the wheel's local vertical axis.
- This restricts its ability to represent more advanced suspension behaviors such as **multi-link, torsion bar, or non-linear spring/damping curves**.

- Real-world effects like suspension compression affecting camber or caster angles are not simulated.

3. Approximate Friction Modeling

- Tire friction is calculated using a Friction Curve, a **Pacejka-like slip curve approximation** provided by (Nvidia's) PhysX . Learn more about them here: [Pacejka curves](). [Comprehensive guide into Pacejka curves]()
- This results in a simplified slip ratio and slip angle response that does not match the full complexity of tire-road interaction.
- Important real-world effects like temperature, load sensitivity, and combined slip forces are absent. and usually handled by developers.
- The friction model often exhibits **instability at high speeds** or when forces rapidly change.

---

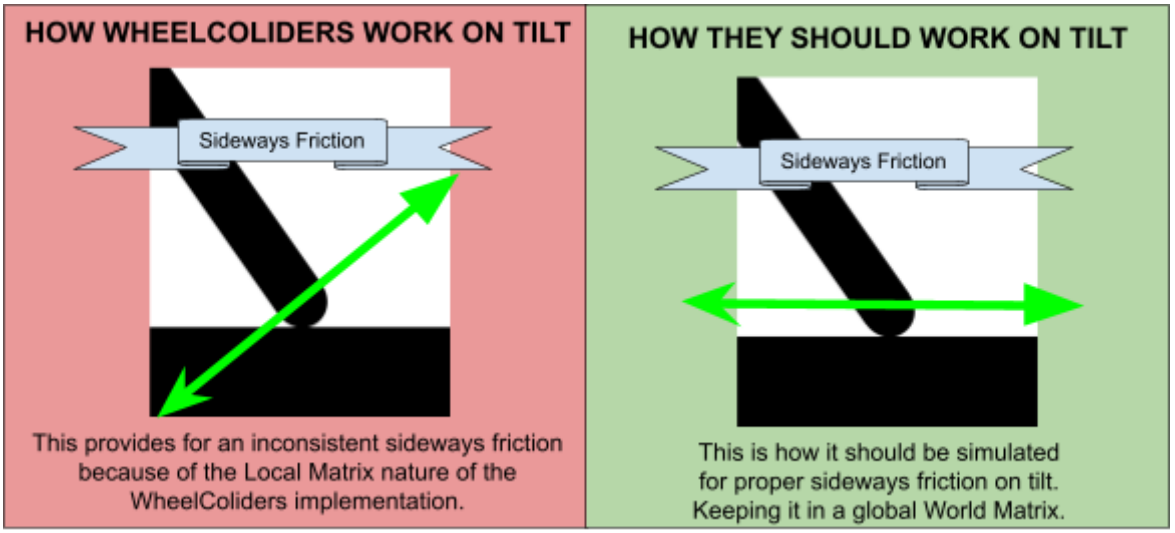
4. Unrealistic Tire Dynamics

- Tires are treated as **infinitely stiff in lateral and longitudinal response**, aside from what the slip curve allows.
- Effects like **tire flex, rolling resistance, or gyroscopic forces** from spinning wheels are not included.
- Wheel mass only affects inertia, not the actual tire-surface to ground contact interaction.

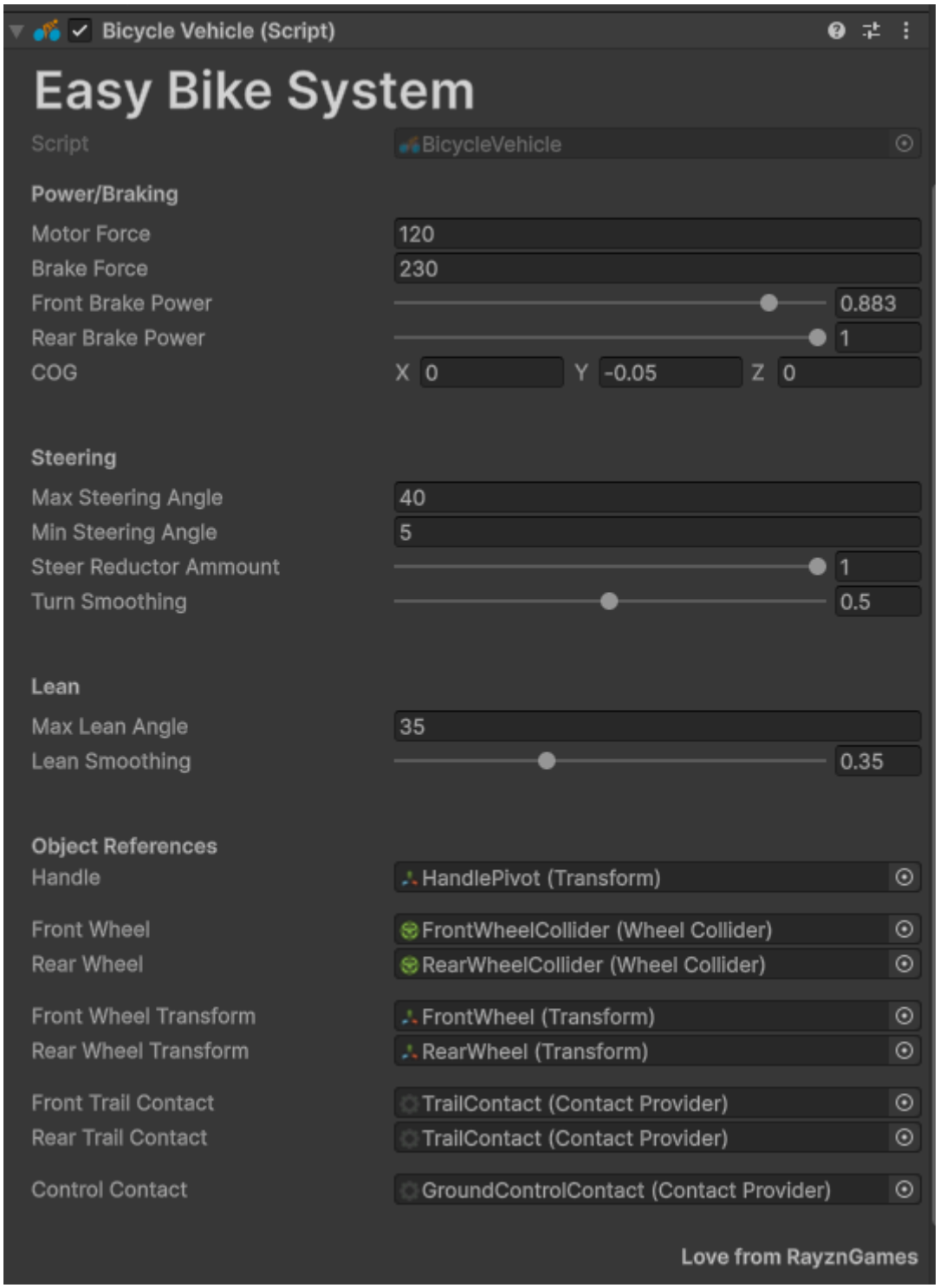
---

5. Dependency on PhysX Integration

- WheelColliders are tightly bound to Unity's **PhysX implementation**, meaning their accuracy and stability are directly limited by the physics engine version.
- Numerical instability can occur in scenarios with very high or very low mass ratios, extreme velocities, or non-standard time steps.



With all of that said these limitations are what makes the **performance to effectiveness** ratio of this component so high, since you can simulate lots of vehicles simultaneously in the physics engine without big inconsistencies, and not a big performance impact.



Thanks for reading.- Sincerely