ROS-Based Autonomous Driving Simulation with BeamNG.tech: From Setup to Control Algorithms

Developed for PSD 2025 & Robot's Mechanics

Professor: Marco Gabiccini

Students: Davide Serpi, Stefano Cimmino

Citations and Acknowledgements

- S. Macenski, T. Foote, B. Gerkey, C. Lalancette, W. Woodall, "**Robot Operating System 2: Design, architecture, and uses in the wild**," Science Robotics vol. 7, May 2022.

- Pascale Maul, Marc Mueller, Fabian Enkler, Eva Pigova, Thomas Fischer, Lefteris Stamatogiannakis, "**BeamNG.tech Technical Paper**"

- Thanks to all the BeamNG.tech Team for the **help with the forum's topics**

- Racerguy24, "**RG R/C Pack**" mod that helped for the structure

Index

Introduction

Objectives, Tools Overview and Requirements

BeamNG

Objectives

The goal of this guide is to set up a simulation environment for testing autonomous driving algorithms. We will use the **BeamNG.tech** simulator integrated with **ROS 2** for sensor data acquisition and vehicle control. This setup allows for flexible **scenario** design, giving users full control over its elements and customization.

Algorithms can be developed in **Python** using ROS 2 **nodes**, without requiring a virtual machine.

The provided vehicle is ready to use immediately after installation. It is a 1:1 scale reproduction of the real vehicle used in the PSD project, specifically created in BeamNG.tech to ensure realistic and reliable testing.

Softwares and Libraries Used

• BeamNG.tech	Environment	0.34.2
• Blender	3D modeling	4.2.2
• WSL 2	for Ubuntu Linux kernel	
• Ubuntu	for ROS 2 installation	22.04
• PyCharm	Python IDE and terminals usage	2024.2.3
• Visual Studio Code	IDE for code editing	1.92
• ROS 2 Humble Hawksbill	for running the bridge with BeamNG.tech	
• Python	for scripting and algorithm development	3.10
• BeamNGpy	Python API for BeamNG.tech	1.31

Requirements

Operating Systems:

- Windows 10/11 (for BeamNG.tech and development tools)
- Ubuntu 22.04 (via WSL 2 for ROS 2 environment)

CPU: Quad-core processor (Intel i5/i7 or AMD Ryzen 5/7 recommended)

RAM: 16 GB (for smoother simulation and parallel processes)

Storage: SSD with 70 GB of free disk space

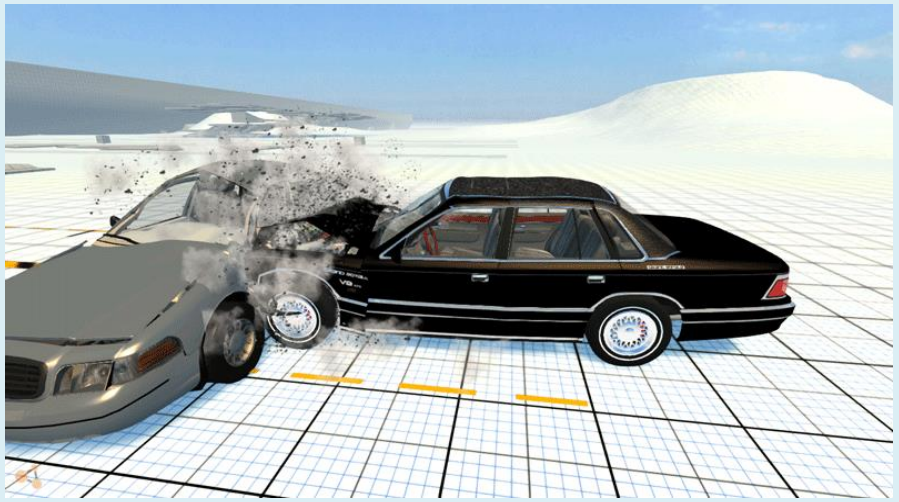
Graphics Card: NVIDIA or AMD dedicated GPU with at least 2GB VRAM (tested with NVIDIA GeForce GTX 1650)

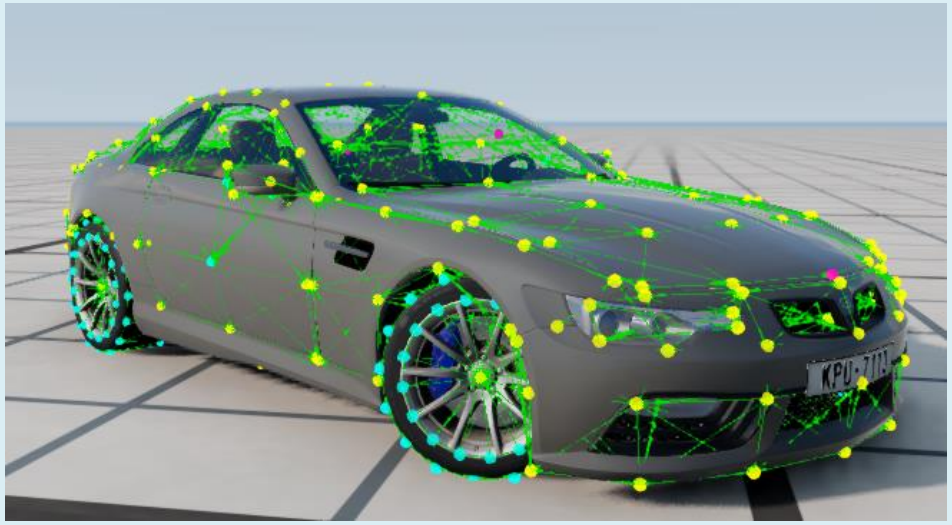
Other requirements: Basic programming skills in Python, familiarity with ROS 2, basic knowledge of installing software on Windows and using Linux bash commands, familiarity with videogame car simulators.

What is BeamNG.tech?

BeamNG.drive is a highly realistic driving simulator known for its advanced soft-body physics and detailed vehicle dynamics.

BeamNG.tech is its R&D-focused version, offering tools for AI testing, autonomous driving, and physics simulations.





BeamNG's physics engine is based on a node and beam structure:

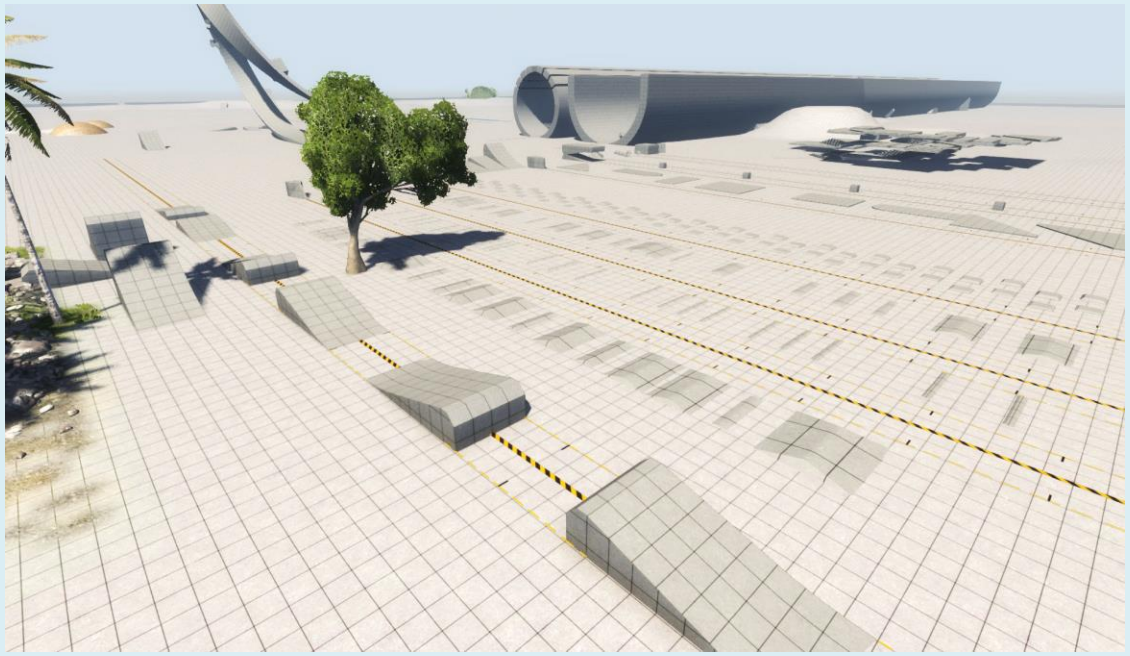
- **Nodes** are mass points that form the vehicle's frame and components.
- **Beams** are elastic connections between nodes, acting like springs and dampers.

Modding on BeamNG.tech

BeamNG is designed to facilitate modding, allowing users to create and customize a wide range of content. The game supports modifications for:

- Vehicles
- Maps
- Scenarios

For this project, we will develop a vehicle mod and a scenario mod. Additionally, we will modify an existing map mod to integrate our customized track.



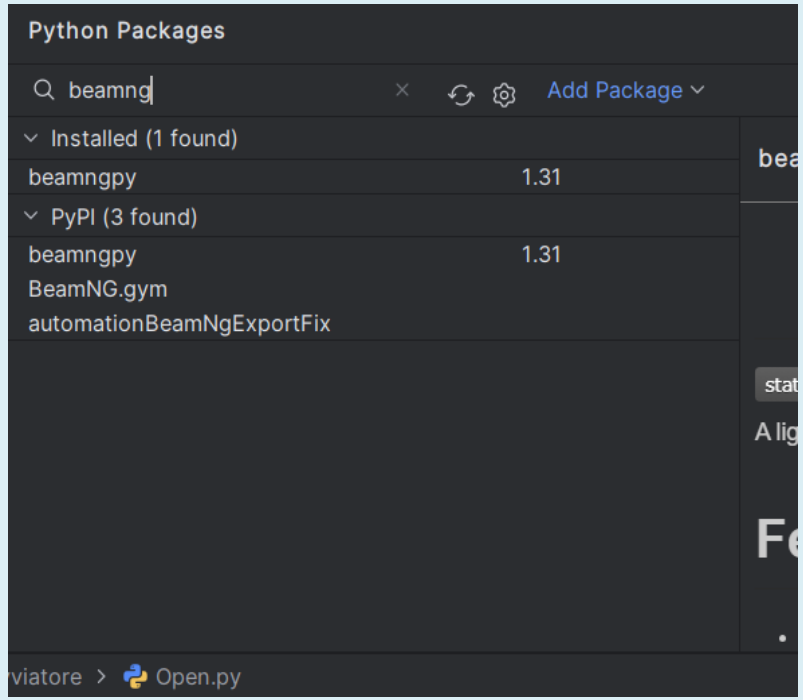
What is BeamNGpy?

BeamNGpy is a Python library that allows interaction with BeamNG.tech and it enables vehicle control, real-time telemetry data retrieval and game environment modifications.

BeamNGpy

We will use BeamNGpy to communicate with the game and for sending command messages in the example Python script.

The following chapters describe how to install the library in a **Python environment** and how to use it. If you are using **PyCharm** as your Python IDE, you can install the library directly from the Python package manager. However, we recommend following the instructions provided later for a more complete setup.



Overview of the Setup Process

Before proceeding with the setup guide, here is an overview of the steps we will follow in the next chapters to set up the vehicle and environment:

- **Installing BeamNG.tech**
- **Launching the game**
- **Installing the vehicle**
- **Setting up the ROS2 workspace**
- **Setting up the ROS2 bridge**
- **Starting a scenario with the ROS2 bridge**
- **Creating a level**
- **Creating a track (optional)**
- **Creating a scenario**
- **Creating a ROS2 publisher node**
- **Developing an algorithm for vehicle control**

These steps will guide you through setting up a functional simulation environment.

BeamNG.tech Installation and Vehicle Files Setup

How to download and setup BeamNG.tech and the vehicle

BeamNG

Chapter index

- **how to install BeamNG.tech**
- **how to lunch the simulator**
- **how to install the vehicle**
- **test the vehicle**

How to install BeamNG.tech

You can install **BeamNG.tech** after your license request has been accepted by the BeamNG team. You can compile and submit your request here:

BeamNG.tech request

You will typically receive a response via email within a few hours. They will inquire about your intended use of BeamNG.tech and will request a brief introduction about yourself. Academic requests for study and research purposes are highly likely to be approved.

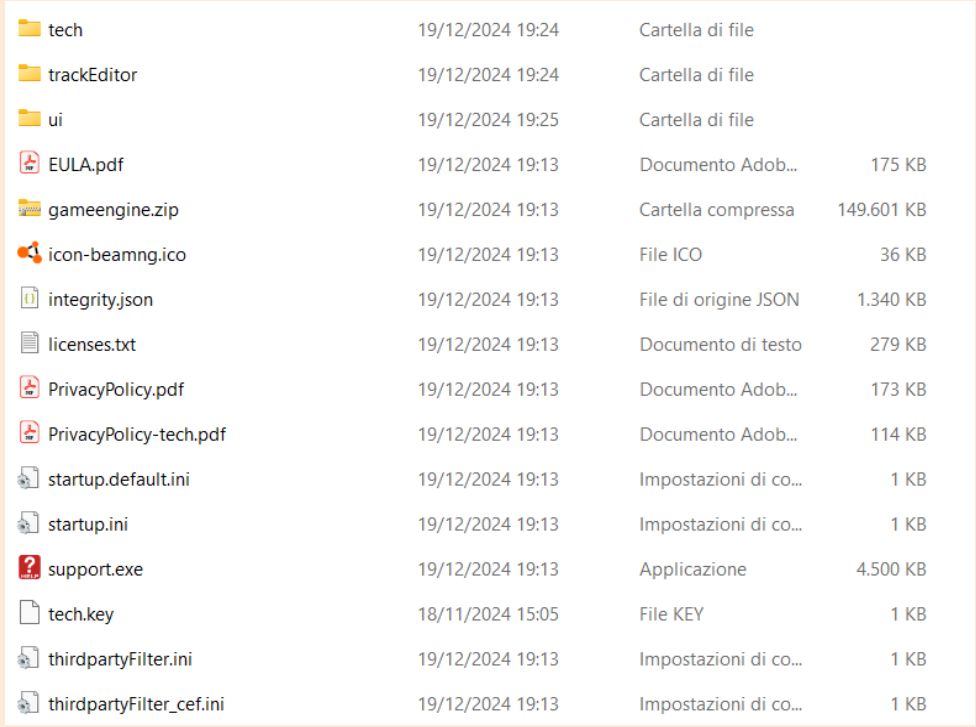
You will receive an email containing a link to download the BeamNG.tech zip file and the key to be placed in the game's folder, along with all relevant information regarding the license.

How to install BeamNG.tech

After that, you can download the zip file from the provided link and extract the folder on the Desktop or any preferred location, as long as it's a Windows folder.
This guide was created using **version 0.34**. If you plan to use a more recent version, please check for compatibility, as this was the latest version available at the time of writing.

Remember that after extracting the folder, you must place the key file inside the game's directory. If a key file already exists, simply replace it with the new one.

After these steps, everything is ready to launch the game.



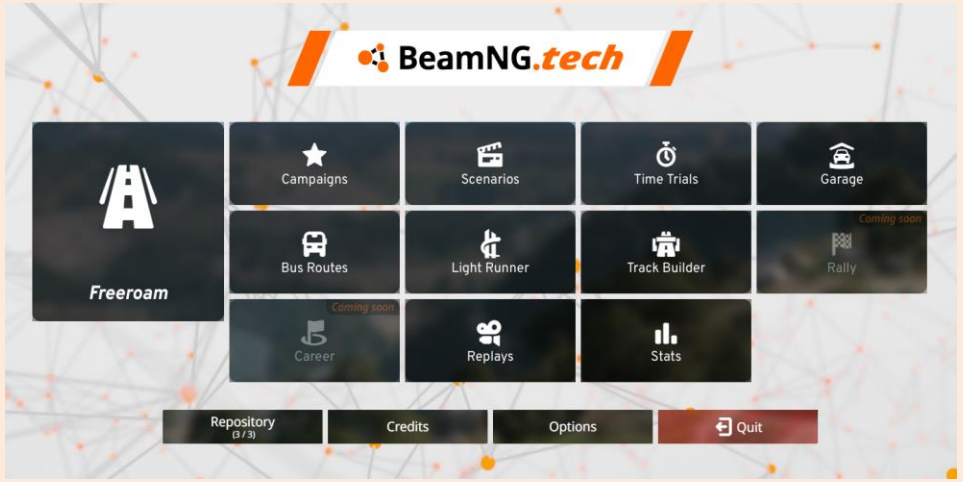
tech	19/12/2024 19:24	Cartella di file	
trackEditor	19/12/2024 19:24	Cartella di file	
ui	19/12/2024 19:25	Cartella di file	
EULA.pdf	19/12/2024 19:13	Documento Adob...	175 KB
gameengine.zip	19/12/2024 19:13	Cartella compressa	149.601 KB
icon-beamng.ico	19/12/2024 19:13	File ICO	36 KB
integrity.json	19/12/2024 19:13	File di origine JSON	1.340 KB
licenses.txt	19/12/2024 19:13	Documento di testo	279 KB
PrivacyPolicy.pdf	19/12/2024 19:13	Documento Adob...	173 KB
PrivacyPolicy-tech.pdf	19/12/2024 19:13	Documento Adob...	114 KB
startup.default.ini	19/12/2024 19:13	Impostazioni di co...	1 KB
startup.ini	19/12/2024 19:13	Impostazioni di co...	1 KB
support.exe	19/12/2024 19:13	Applicazione	4.500 KB
tech.key	18/11/2024 15:05	File KEY	1 KB
thirdpartyFilter.ini	19/12/2024 19:13	Impostazioni di co...	1 KB
thirdpartyFilter_cef.ini	19/12/2024 19:13	Impostazioni di co...	1 KB

How to launch the simulator

You can either launch the executable `BeamNG.tech.x64.exe` in the `Bin64` folder located in the main folder of the game or launch it through **PowerShell** in **administrator mode** and open the console for debugging and visualizing warnings and errors:

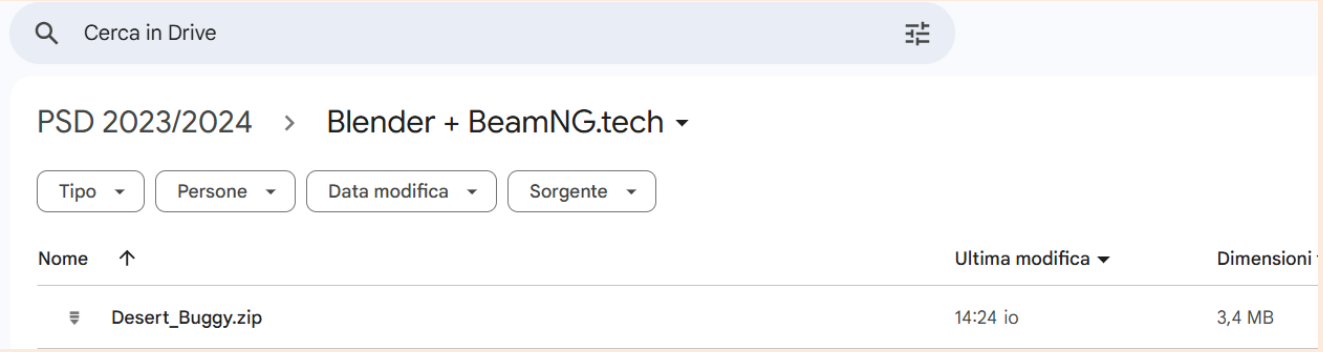
```
E:\PSD\BeamNG.tech.v0.34.2.0\Bin64\BeamNG.tech.x64.exe -console -nosteam
```

If everything has been set up correctly, when you launch the game, you will see the text "BeamNG.tech."

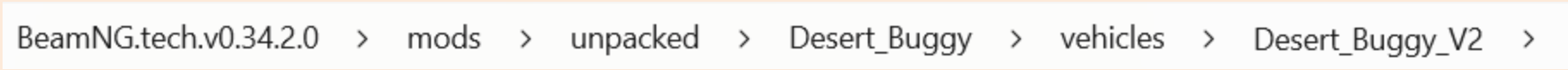


How to install the vehicle

You can find the vehicle's mod in the BeamNG.tech PSD 2023/2024 drive:



It is necessary to create several folders: within the game's directory, create a folder named "mods", and inside it, create a subfolder called "unpacked". Then, extract the mods into this folder. Afterward, delete the zip file and properly configure the folder structure:

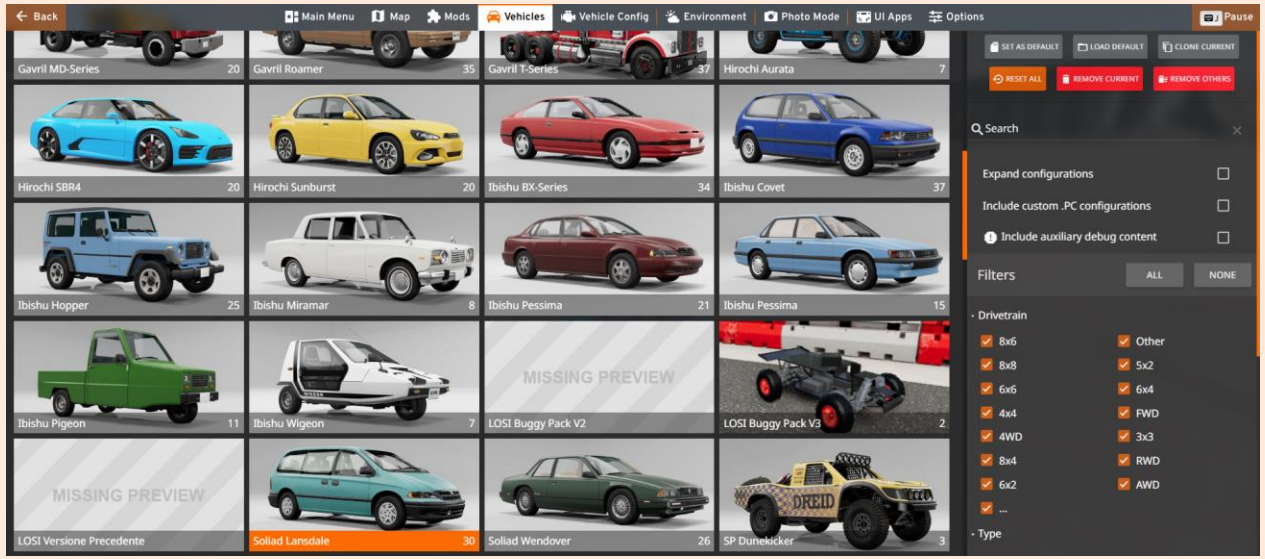


There aren't any zip files so be careful about this.
For any problem you can read the documentation here: [Mods documentation]

Test the vehicle

After these steps you can drive and test the vehicle in game.

Lunch the game and select the **garage** mode, then search the vehicle into the vehicle list:



Select the package and one of vehicle configuration.

To try the vehicle click the “Test” button located in the bottom-left corner.

BeamNG-ROS2 Bridge

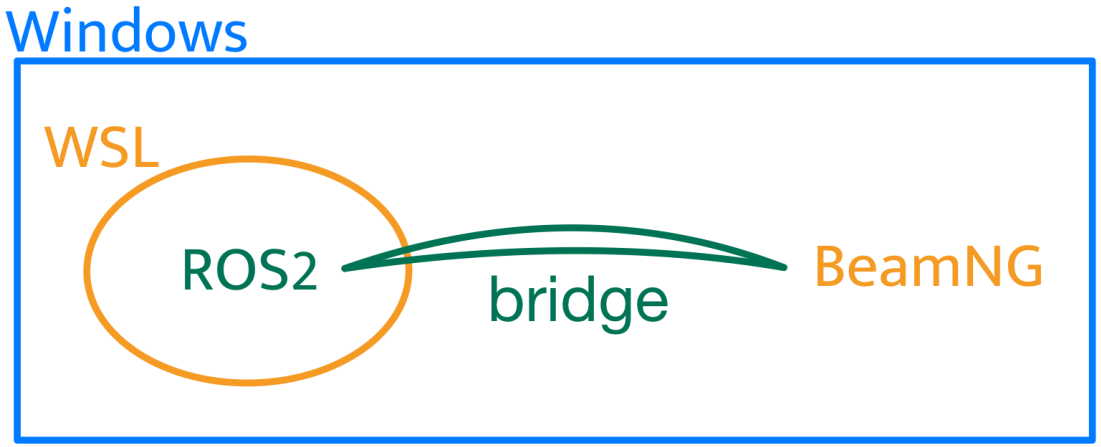
How to connect BeamNG.tech with ROS 2

Chapter index

Configuration

Among the possible configuration alternatives, and as recommended by the developers of BeamNG and the bridge, the following setup has been chosen.

The **BeamNG.tech** runs on Windows since it requires *Vulkan* API to leverage the dedicated GPU.
Meanwhile, **ROS 2 Humble Hawksbill** runs on a Linux kernel (Ubuntu 22.04) installed via WSL (Windows Subsystem for Linux).



WSL installation

The requirement is to have Windows 10 version 2004 or later, or Windows 11. Open **PowerShell** or **Command Prompt** in **administrator mode** and enter the command:

```
wsl --install
```

Since the recommended Linux distribution is Ubuntu 22.04, you can use the command:

```
wsl --install -d Ubuntu-22.04
```

Then open **Ubuntu** as **administrator** from the Start menu and follow the steps to create the user.

After that we can update the packages using the command:

```
sudo apt update && sudo apt upgrade -y
```

Installing BeamNGpy

Both the ROS 2 and BeamNG.tech integration and sending commands to the simulator client require a Python library called **BeamNGpy**.
To install it, simply open Ubuntu from the Start menu with **administrator privileges** and run the following command:

```
pip install beamngpy
```

This installs the library **globally**, and the following guide will still work. However, to avoid potential conflicts with other installed library versions, it is recommended to create a virtual environment, as suggested by the developers:

```
mkdir -p ~/venv_beamngpy      # choose the name you prefer
python3 -m venv ~/venv_beamngpy
source ~/venv_beamngpy/bin/activate
pip install beamngpy
```

These are individual command lines; you can enter them one by one or copy and paste all the commands into the terminal at once.

This commands create a new directory for the virtual environment, create the virtual environment inside the new directory, activate the virtual environment and install BeamNGpy inside it.

Installing BeamNGpy

In case of issues due to the missing `ensurepip` module (required for managing the installation of `pip` within the virtual environment), you need to install the `python3-venv` package by running the following command and then try to recreate and activate the virtual environment again:

```
sudo apt install python3.10-venv
python3 -m venv ~/venv_beamngpy
source ~/venv_beamngpy/bin/activate
```

Every time you open a new terminal, activate the virtual environment before using BeamNGpy:

```
source ~/venv_beamngpy/bin/activate
```

And every time you need to deactivate it, you can use the command:

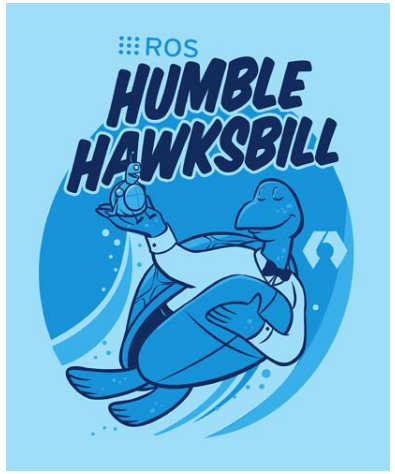
```
deactivate
```

At this point, let's move on to the installation of ROS 2. You can safely install it with the virtual environment active, but to be extra safe, we recommend temporarily deactivating it before installing ROS 2 and reactivating it afterward.

ROS 2 Humble Hawksbill setup guide

The following **integration** guide can be fully accessed at the following link:
https://github.com/BeamNG/beamng-ros2-integration

Meanwhile, the steps to install **ROS 2 Humble Hawksbill**, the tested distribution for the integration, can be found at:
https://docs.ros.org/en/humble/Installation.html
Under the "Binary packages" section in "deb packages".



ROS 2 Humble Hawksbill setup guide

Set locale

In the same Ubuntu terminal opened as administrator, insert these commands:

```
locale  # check for UTF-8

sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8

locale  # verify settings
```

These commands are used to set and verify the system locale, ensuring that it supports UTF-8 encoding. This is important because some applications, including ROS 2, expect a properly configured locale to handle text and special characters correctly.

ROS 2 Humble Hawksbill setup guide

Setup sources

These commands set up the ROS 2 package sources on your system, allowing you to install ROS 2 through `apt`. First ensure that the Ubuntu Universe repository is enabled (which contains community-maintained packages) because ROS 2 requires packages from this repository.

```
sudo apt install software-properties-common
sudo add-apt-repository universe
```

Install `curl`, a tool used to download files from the internet and then download the official ROS 2 GPG key, which is used to verify that the ROS 2 packages come from a trusted source.

```
sudo apt update && sudo apt install curl -y
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o
/usr/share/keyrings/ros-archive-keyring.gpg
```

ROS 2 Humble Hawksbill setup guide

Setup sources

Then add the repository to your sources list:

```
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-
keyring.gpg] http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo $UBUNTU_CODENAME)
main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
```

ROS 2 Humble Hawksbill setup guide

Install ROS 2 packages

Update the package lists, ensuring `apt` knows about the latest available versions of all software.

```
sudo apt update
```

And update all installed packages to their latest versions (ROS 2 packages are built on frequently updated Ubuntu systems. It is always recommended that you ensure your system is up to date before installing new packages).

```
sudo apt upgrade
```

We have three different installation options, depending on our needs as shown in the guide. We recommend downloading the *desktop install*: it has ROS 2 core components, plus RViz (a visualization tool), demos & tutorials (useful for learning ROS 2).

```
sudo apt install ros-humble-desktop
```

ROS 2 Humble Hawksbill setup guide

Environment setup

After installing ROS 2, you need to set up your environment so that your system recognizes ROS 2 commands and tools. This is done by sourcing the setup script.

```
# Replace ".bash" with your shell if you're not using bash
# Possible values are: setup.bash, setup.sh, setup.zsh

source /opt/ros/humble/setup.bash
```

By default, ROS 2 is installed in `/opt/ros/humble/`, but the system does not automatically recognize ROS 2 commands. Sourcing the script updates the system's environment so that it knows where ROS 2 is installed and how to use it. That's why every time you open a new terminal, you must manually source the setup script again.
To avoid this, you can add it to your shell's startup file:

```
echo "source /opt/ros/humble/setup.bash" >> ~/.bashrcsource ~/.bashrc
```

This automatically loads the ROS 2 environment every time a new terminal is opened.

ROS 2 Humble Hawksbill setup guide

Talker-listener example

Once ROS 2 is installed and set up (good job!), you can test it by running a simple communication example using the publisher-subscriber model. This example consists of:
- a **talker** (**publisher**) that sends messages
- a **listener** (**subscriber**) that receives those messages

Since ROS 2 supports both C++ and Python, this test ensures that both APIs are working correctly.
In one terminal, source the setup file and then run a C++ **talker**:

```
source /opt/ros/humble/setup.bash
ros2 run demo_nodes_cpp talker
```

In another terminal source the setup file and then run a Python **listener**:

```
source /opt/ros/humble/setup.bash
ros2 run demo_nodes_py listener
```

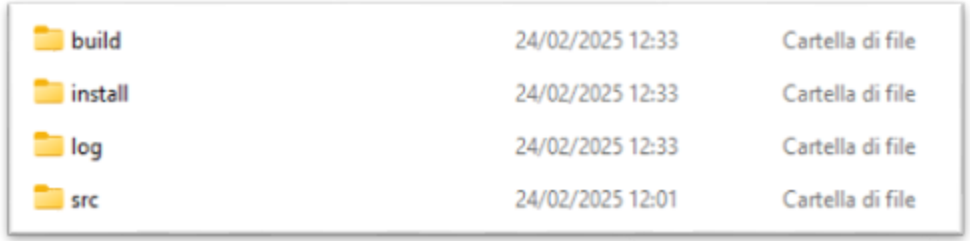
Press **Ctrl + C** to terminate the current process.

ROS 2 – Creating a workspace

Understanding workspaces

A **workspace** in ROS 2 is a directory that follows a specific structure to organize, build, and manage ROS 2 packages. It typically contains:

- **src/** → Stores the source code of ROS 2 packages
- **build/** → Holds intermediate build files
- **install/** → Contains the installed packages
- **log/** → Stores logs from builds



Workspaces allow developers to compile, modify, and test ROS 2 packages in an isolated environment. Multiple workspaces can be stacked using overlays, where a new workspace builds on top of an existing ROS 2 installation.

By default, each package is installed separately in the install directory to keep the workspace modular.

ROS 2 – Creating a workspace

Install colcon

`colcon` is a build system used for ROS 2 that allows managing and compiling multiple packages efficiently. To install `colcon`, run:

```
sudo apt install python3-colcon-common-extensions
```

This package provides additional tools and extensions for building, testing, and packaging ROS 2 workspaces.

Create a workspace

To create a new workspace:

```
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws
```

The tilde (~) symbol in a Linux terminal represents the home directory of the current user. In this case the last command is:
`cd /home/username/ros2_ws`

ROS 2 – Creating a workspace

Sourcing an underlay

Before building the workspace, you must source the ROS 2 installation. This step ensures that the workspace has access to the necessary build dependencies for ROS 2 packages.
To source the underlay, run:

```
source /opt/ros/humble/setup.bash
```

An "underlay" is an existing ROS 2 installation. Your new workspace (`~/ros2_ws`) will act as an "overlay," meaning it extends the functionality of the existing ROS 2 environment.
Use an overlay if you want to modify a few packages without rebuilding everything from scratch.

ROS 2 – Building the workspace

Clone the BeamNG-ROS 2 repository

Move into the `src` directory of your ROS 2 workspace, where the source code of ROS 2 packages will be placed:

```
cd ~/ros2_ws/src
```

Download the **BeamNG-ROS2 integration package** by cloning its GitHub repository:

```
git clone https://github.com/BeamNG/beamng-ros2-integration.git
```

Move back to the root of the workspace (`ros2_ws`):

```
cd ~/ros2_ws
```

You can also use the command **`cd ..`** to move up to the parent directory of your current location.

ROS 2 – Building the workspace

Installing dependencies with rosdep

`rosdep` is a dependency management tool for ROS that helps install required packages and external libraries before building a workspace.
Use `rosdep` to install the required dependencies automatically (comes bundled with the distribution):

```
rosdep install -i --from-path src --rosdistro humble -y
```

If you encounter any errors while running the command above, follow these steps to ensure `rosdep` is properly installed and initialized:

```
sudo apt-get install python3-rosdep
rosdep update
sudo rosdep init
rosdep install --from-paths src -y --ignore-src
```

These steps will install **rosdep**, initialize it, update its package index, and then attempt to install the dependencies again. After completing these steps, try running the original **rosdep** install command again to ensure all dependencies are correctly installed.

ROS 2 – Building the workspace

Optional dependencies

BeamNG-ROS2 will use the **python-rapidjson** library to support JSON files that do not strictly follow the standard. You can install it by using the command:

```
pip install python-rapidjson
```

ROS 2 – Building the workspace

Build the workspace

Compile the packages inside the workspace using:

```
colcon build
```

Alternatively, you can use:

```
colcon build --symlink-install
```

it saves you from having to rebuild every time you tweak python scripts.

Now, the **BeamNG-ROS2 integration** should be successfully added to your ROS 2 workspace and ready to use!

Terminal Setup & Workflow for BeamNG-ROS2 Integration

This is an example of how to manage a BeamNG scenario using the bridge to enable communication between the simulator and ROS 2 and the recommended workflow is to manage multiple terminals separately, as shown.

Step 1: Launch BeamNG.tech

The first terminal to open is **Windows PowerShell** (or **Command prompt**), where you need to run the following command:

```
<destination_to_BeamNG.tech>\BeamNG.tech.v0.34.2.0\Bin64\BeamNG.tech.x64.exe -console -nosteam -tcom-listen-ip 0.0.0.0 -lua "extensions.tech_techCore.openServer(25252)"
```

`-tcom-listen-ip` **`0.0.0.0`** → Tells the simulator to listen for external connections on all network interfaces. Working with the simulator on Windows and ROS on WSL, you can either set `0.0.0.0` or the WSL IP address.

`-lua "extensions.tech_techCore.openServer(`**`25252`**`)"` → Runs a Lua script to open a communication server on port 25252, which allows external applications (such as ROS 2) to connect and control the simulation.

Terminal Setup & Workflow for BeamNG-ROS2 Integration

Step 2: Set Up ROS 2 Environment (Ubuntu Terminal 1)

Open the first **Ubuntu terminal** and enter the following commands:

```
source /opt/ros/humble/setup.bash
cd ~/ros2_ws
source install/local_setup.bash
```

These commands must be executed in every new Ubuntu terminal:

- `source /opt/ros/humble/setup.bash` → Loads the ROS 2 environment variables so that ROS commands can be used
- `cd ~/ros2_ws` → Moves into the ROS 2 workspace, where the integration package is located
- `source install/local_setup.bash` → Sources the colcon-built workspace, ensuring that the installed ROS 2 packages (including beamng_ros2) are recognized

Terminal Setup & Workflow for BeamNG-ROS2 Integration

Step 3: Start the ROS 2 Bridge (Ubuntu Terminal 2)

Open a second **Ubuntu terminal** and enter the following:

```
source /opt/ros/humble/setup.bash
cd ~/ros2_ws
source install/local_setup.bash
```

Then, start the ROS 2 bridge with:

```
ros2 run beamng_ros2 beamng_bridge
```

This launches the **ROS 2 node** that acts as a bridge between the simulator and ROS 2.

A **ROS 2 node** is a process that performs computations in a ROS 2 system. It can communicate with other nodes via **topics**, **services**, or **actions**.
Once this node starts, it continuously listens for messages and communicates with BeamNG.tech, so you cannot enter new commands in this terminal while it is running.
If you need to stop it, press **Ctrl + C**.

Terminal Setup & Workflow for BeamNG-ROS2 Integration

In case of issues with the **NumPy** version required by the bridge to function properly (the warning displayed is: "`NumPy version >=1.17.3 and <1.25.0 is required for this version of SciPy`"), here are the commands to uninstall and reinstall a compatible version of NumPy. During the project, an incompatible version of NumPy (1.26.4) was used, but the bridge still worked.

```
pip uninstall numpy
pip install numpy==<compatible_version>
```

Terminal Setup & Workflow for BeamNG-ROS2 Integration

Step 4: Set the Windows IP Address (Back to Ubuntu Terminal 1)

Since BeamNG.tech is running on Windows, and ROS 2 is running on Ubuntu, the ROS 2 bridge needs to know where to send its messages.

Return to the **Ubuntu terminal 1** and set the IP address of the Windows machine running BeamNG.tech:

```
ros2 param set /beamng_bridge host <your_windows_IP>
```

Open Command Prompt and insert the command **`ipconfig`**, and copy the address corresponding to "IPv4"

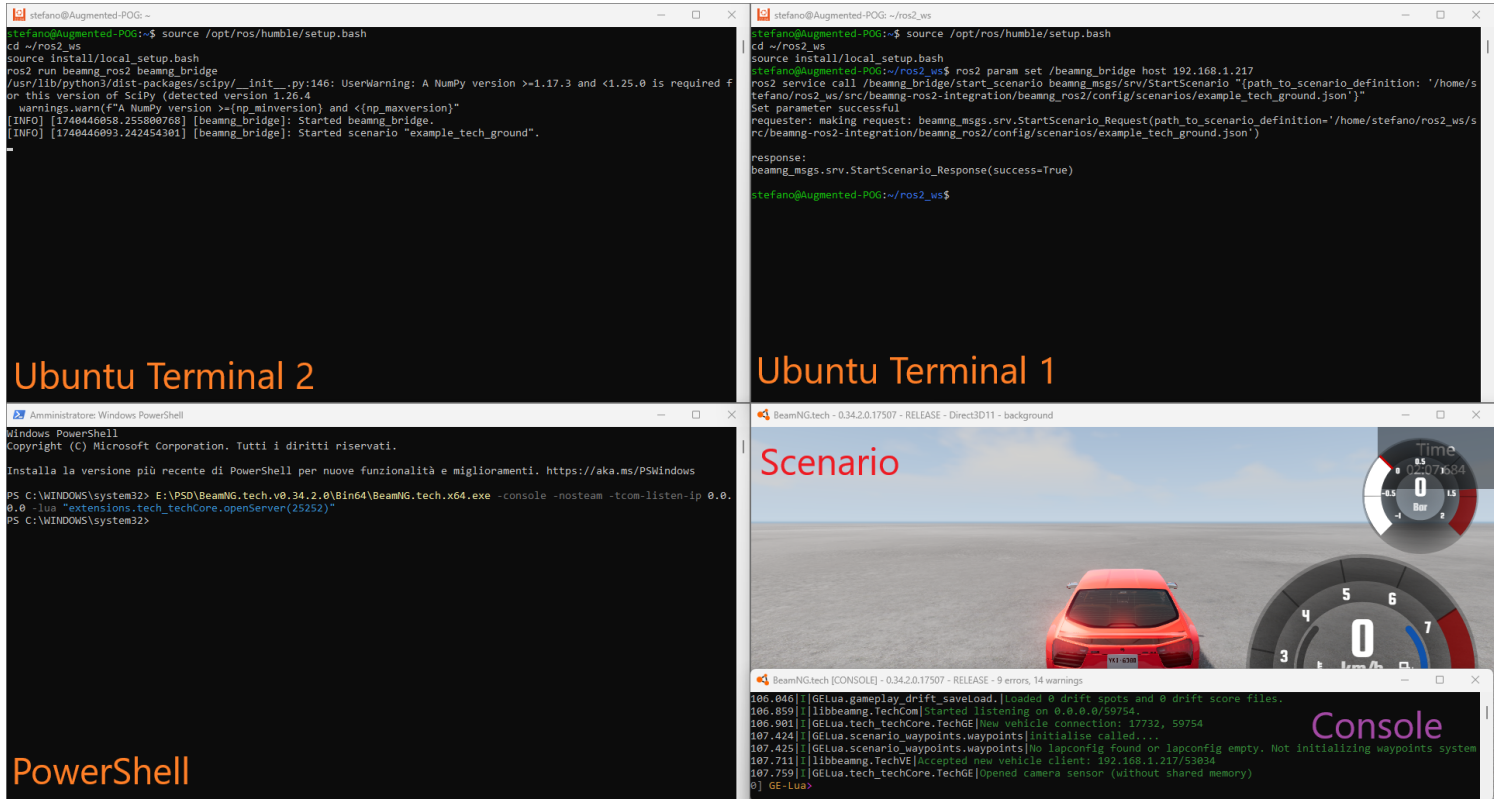
Step 5: Start a Sample Scenario (Ubuntu Terminal 1)

In the same terminal, you can now request BeamNG.tech to start a predefined scenario. For example:

```
ros2 service call /beamng_bridge/start_scenario beamng_msgs/srv/StartScenario
"{path_to_scenario_definition: '/home/<ubuntu_username>/ros2_ws/src/beamng-ros2-
integration/beamng_ros2/config/scenarios/example_tech_ground.json'}"
```

Terminal Setup & Workflow for BeamNG-ROS2 Integration

When the scenario loading is complete, it will be possible to enter commands in terminal 1 again.



In this guide, we have covered:

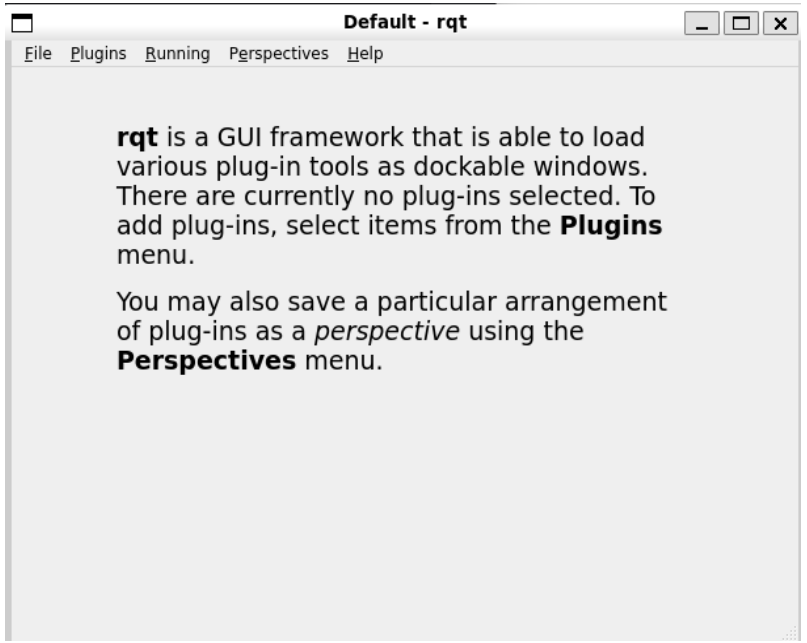
- How to install WSL
- How to install ROS2
- How to create a workspace and install the necessary dependencies
- The terminal workflow used to test scenarios

In the next one, we will go into the details of the scenario.

rqt

rqt is a ROS 2 graphical tool that provides a visual interface for monitoring and debugging topics, nodes, services, and parameters in a ROS-based system.
Some of his useful plugins are:

- **Node Graph**: Visualizes active ROS nodes and their connections, helping track data flow and diagnose system interactions.
- **Topic Monitor**: Monitors and displays real-time information about ROS topics, publishers, and subscribers, enabling analysis of message flow.
- **Image View**: Displays image messages from ROS topics, allowing real-time inspection of visual data from sensors like cameras.



rqt

rqt is already included with ROS 2; to add the plugins mentioned earlier, you need to run the following command:

```
sudo apt install ros-humble-rqt-graph ros-humble-rqt-plot ros-humble-rqt-image-view
ros-humble-rqt-reconfigure
```

After the ROS sourcing, you can open rqt using the following command:

```
rqt
```

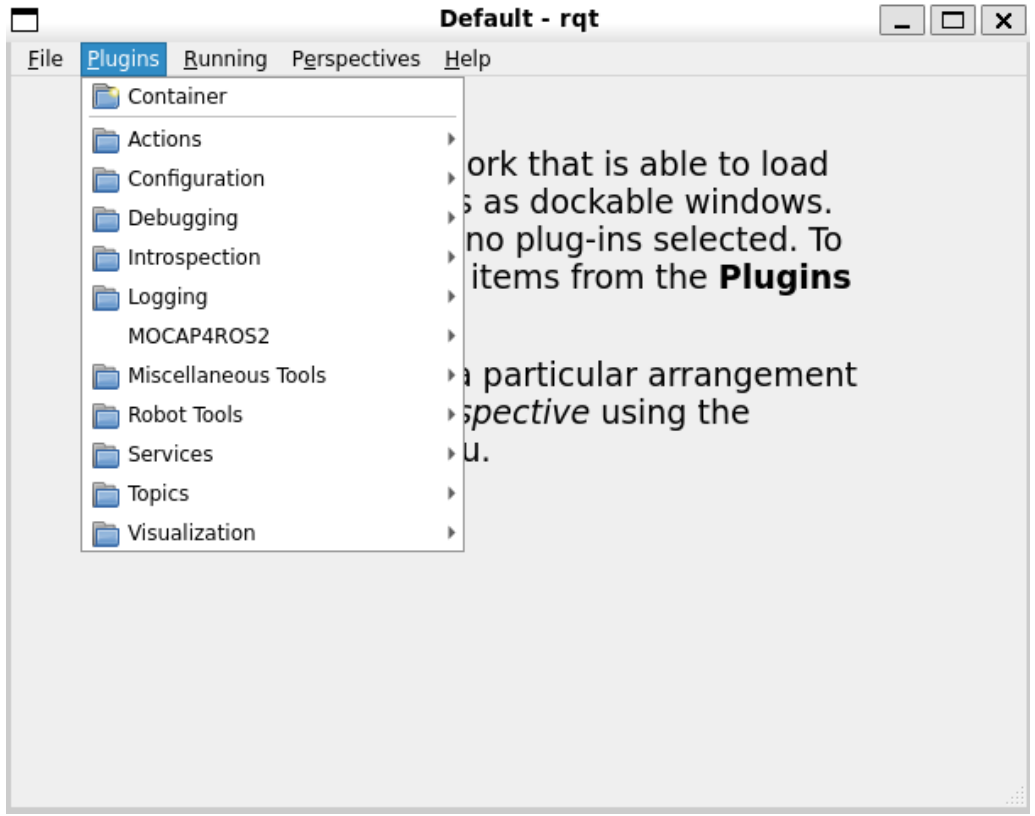
Once you open rqt, you won't be able to use the terminal from which it was launched until you close rqt. The terminal will remain occupied by the running instance of rqt.

If rqt experiences a visual glitch, you can return to the default visualization by closing rqt by pressing **Ctrl + C** in the terminal where rqt was opened, and then run the following

```
rm -rf ~/.config/ros.org/rqt*
```

rqt

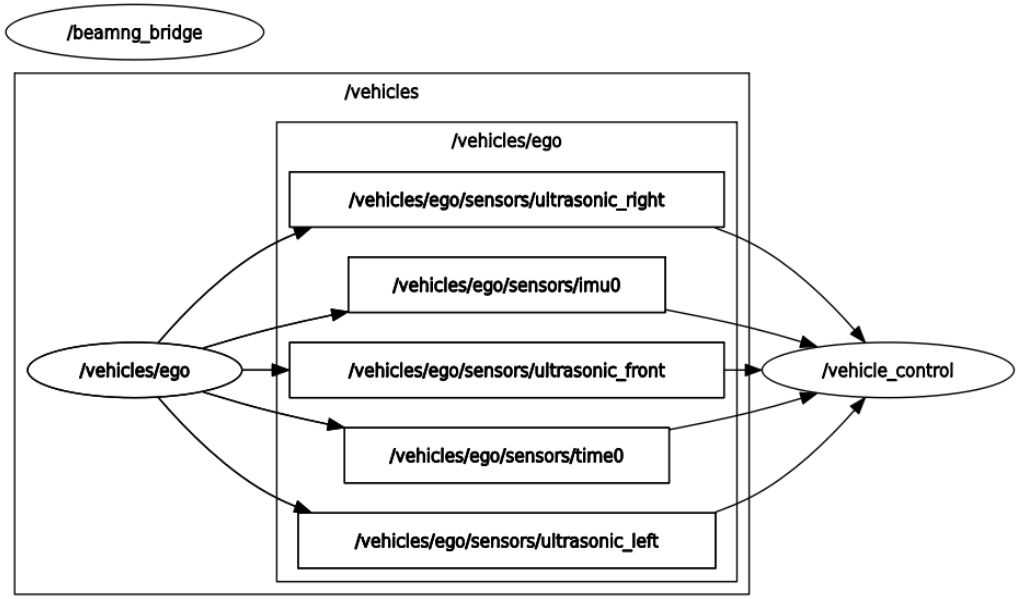
After opening rqt, you can view all the available plugins by clicking on the "**Plugins**" button.



You can also set the view you prefer for better visualization of the plugins.

Nodes and Topics visualization using rqt

For example, if you are running the **algorithm example** as shown later**,** and open the **introspection menu,** by selecting the "**Node Graph**" plugin you will see something like this:



Where the ROS topic and node structure of the bridge and our algorithm are described.

Track Building, Scenario and Sensors

Creating a track, a scenario and sensors overview

Chapter index

What we need?

At this point of the documentation, you have set up the BeamNG.tech environment and the vehicle. However, three more elements are required to start the simulation of the modded vehicle on a modded map:

- **Level**: The map where the scenario takes place.
- **Track**: The circuit used to test the vehicle's driving algorithm
- **Scenario**: Defines the vehicle's spawning position and sensor setup.

Before explaining how to create and configure these elements, it is essential to understand the two primary tools used for track building and level creation:

- **Track Builder**: A user-friendly circuit design tool specifically designed for creating racing tracks.

- **World Editor**: A comprehensive level-editing tool that allows users to modify existing maps or create entirely new environments.

How to install levels

We can use these two tools to create the track, but before doing so, we must first select a base level.

Levels serve as the maps where the vehicle is spawned. While there are several vanilla levels available (vanilla refers to those already included in the game), we need to modify the level's folder. Therefore, we will be using a modded level.

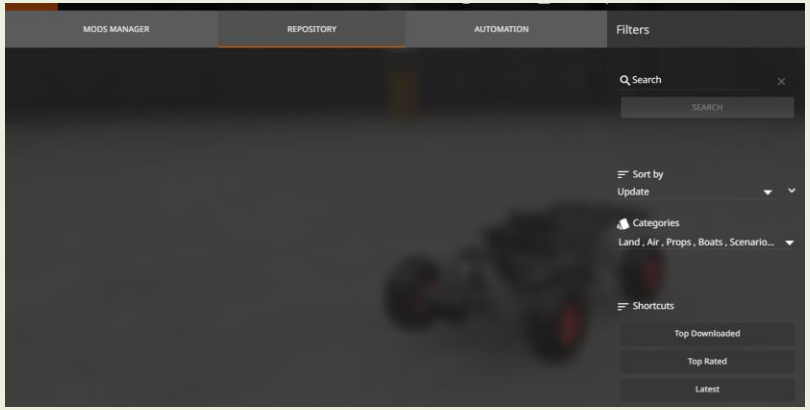
You are free to download and use any modded map you prefer, but we recommend downloading the official mod map from the BeamNG repository:

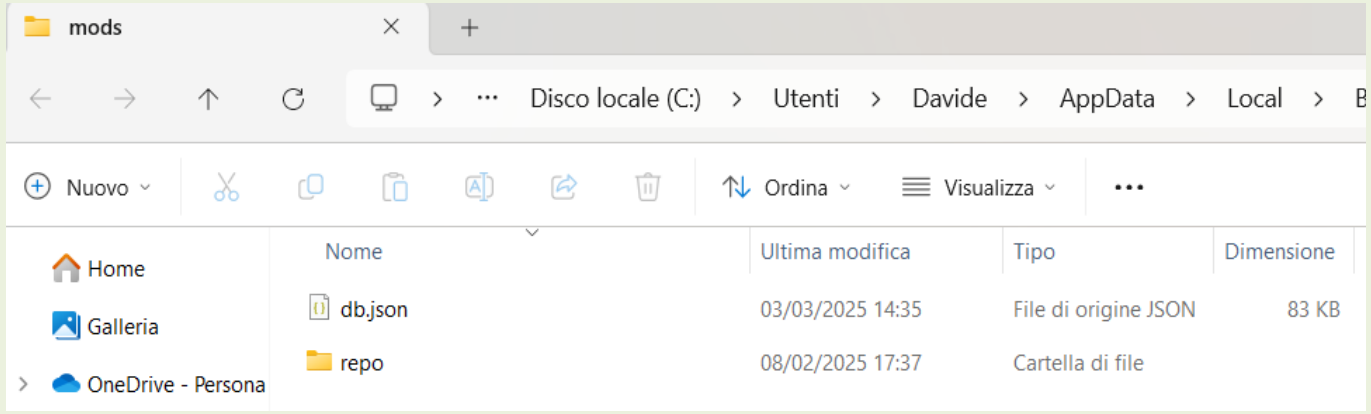
Gridmap_v1_download

How to install levels

There are two ways to install a level in BeamNG: **In-Game Installation** or **Manual Installation.**

1. You can check in the repository in game and search the level you want to use.



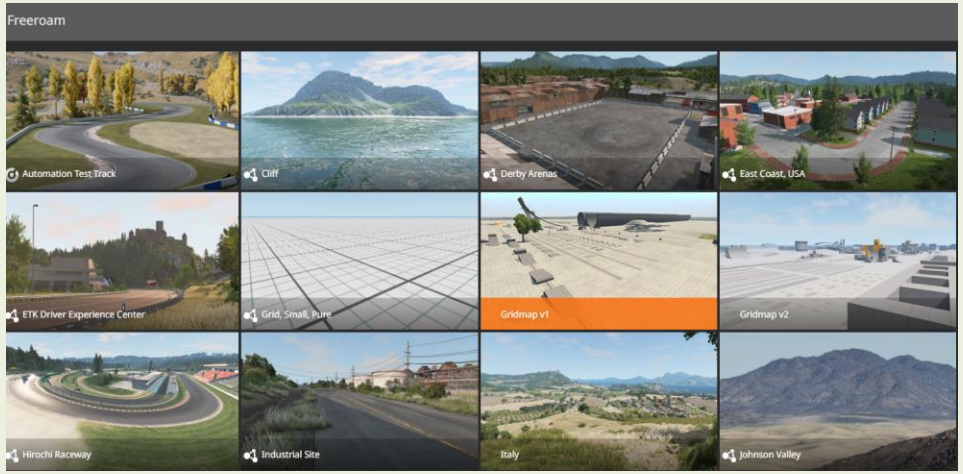


2. You can download the zip file from the following link: Gridmap_v1_download. Once downloaded, place the zip file in your **mods folder/repo**.

This folder is different from the one we previously created for the vehicle. It is automatically generated by the game and can be accessed through the **"Mods" menu** by clicking the **"Open Mod Folder"** button.

How to install levels

After the installation you can see in game the level downloaded in the "Freeroam" menu.

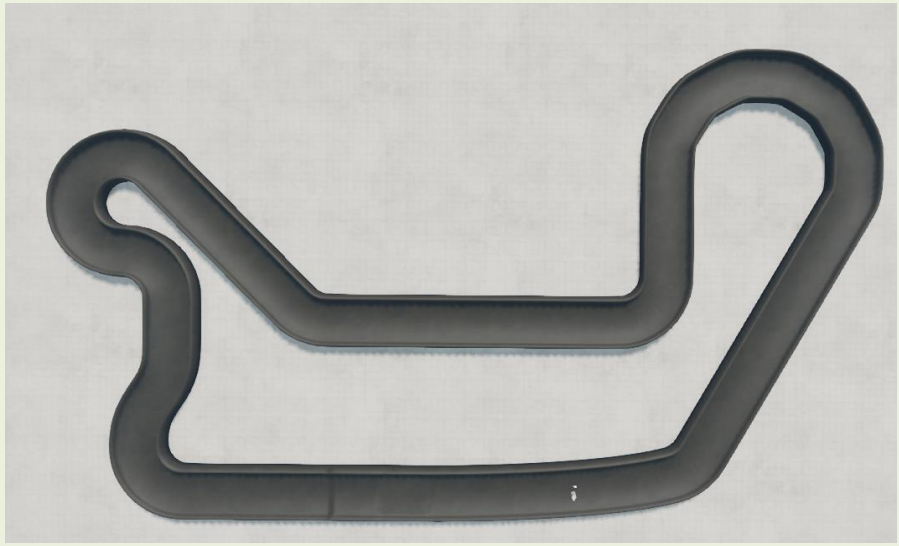


We suggest this map for his lightweight, there is a lot of space for working and also you can modify everything without worry about to modifying vanilla game's files.

How to setup a given track

With the World Editor and Track Builder, there are virtually no limits to what players can create, from custom racetracks to fully interactive environments.

For this example, we have created the track shown on the left. You can download the circuit **.dae** file from the **PSD Drive**: **placefolder.dae**.



The material's name of the .dae is "**MaterialCircuit**", This name is very important because we only need to setup the material for use the circuit.

The first thing to do is extract the mod's file of the level we install before and go inside the folder to find the level's file components (you can see the art folder for example)

Local › BeamNG.drive › 0.34 › mods › repo › gridmap_legacy.zip › levels › GridMap_legacy ›

How to setup a given track

Place the **.dae** file you downloaded into this folder. Once done, rezip the main folder of the mod.
Be careful with the folder structure—ensure that it remains unchanged.

Now, launch the game and start the level from the "**Freeroam**" menu. The selected vehicle does not matter at this stage.

Next, open the **World Editor** by pressing **F11**. If you press **F11 + Ctrl**, the **Safe Mode** of the World Editor will open. To switch to the correct version, simply load the level you are working on using the "**Open Level**" button.

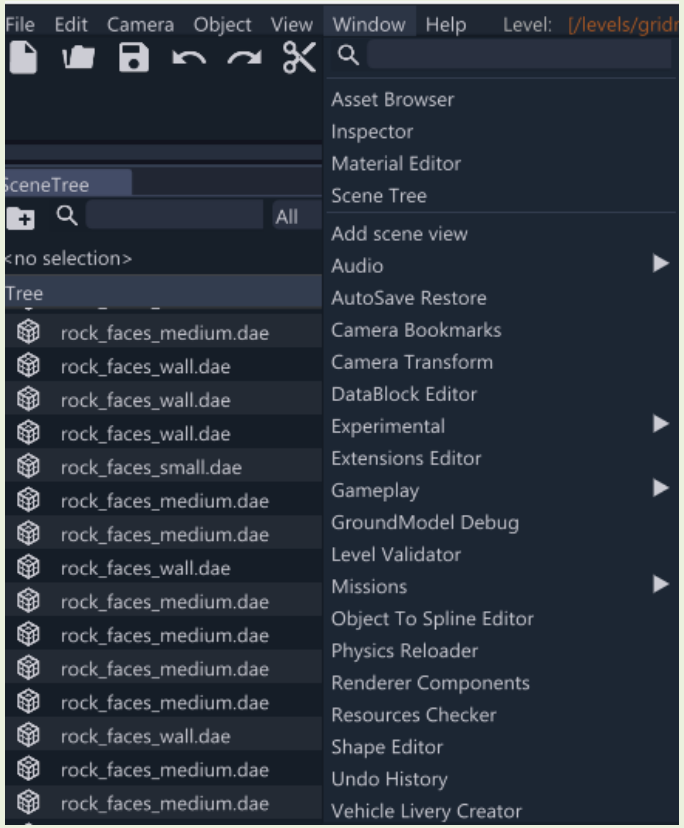
In the **World Editor**, we need to complete two tasks:

- **Place the circuit into the level.**
- **Assign materials to the circuit.**

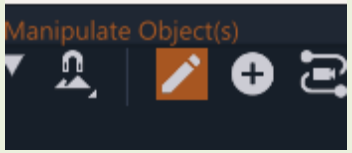
How to setup a given track

In the **World Editor**, you will find several useful tools in the **"Window"** menu:

- **Asset Browser**: Used to search for the circuit.
- **Inspector**: Displays information about the selected element.
- **Material Editor**: Used to configure materials for objects.



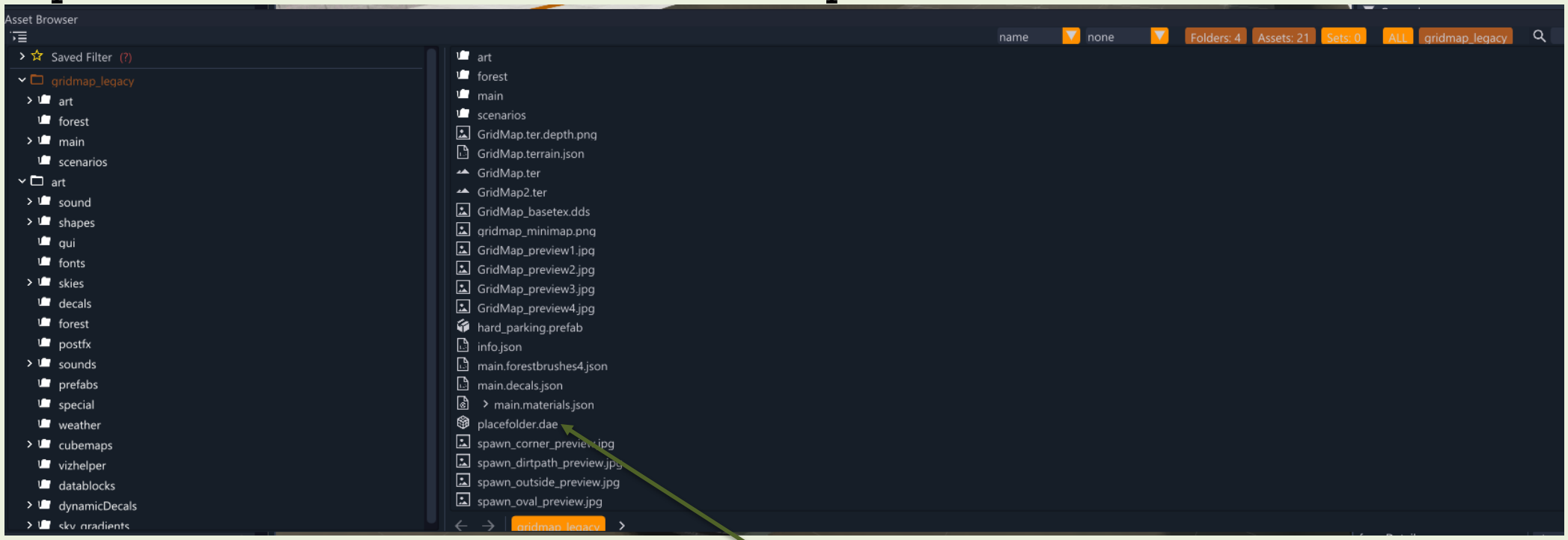
We will primarily use the **Material Editor** for setting up the circuit's materials.
You can select an object within the level by clicking on this icon:



To explore the map freely, press **Shift + C**. This will allow you to exit the car view and navigate the environment freely.

How to setup a given track

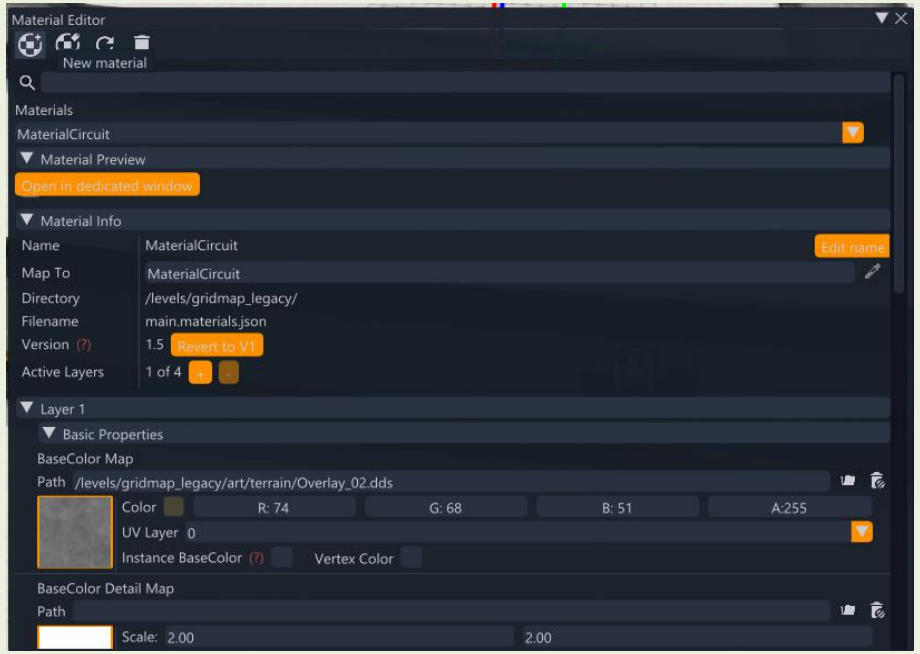
Open the Asset Browser tool and search the placefolder.dae file:



Click and hold the **.dae** file, then drag it into the level. The exact position is not crucial, but make sure to place it in an open space!

How to setup a given track

You will see the circuit appear inside the level, but it may have a strange orange texture—this is normal, as no material has been applied yet.
First, select the circuit while the **Inspector** tool is active and locate the **collision options**.
Make sure the collision settings are correctly configured to allow the vehicle to interact with the circuit.



Next, we need to create a material for the circuit.

- Select the circuit again.
- Open the **Material Editor**.
- Click on "**New Material**".
- Create a new material with the **same name** as the
- circuit's material (for this circuit, it should be "**MaterialCircuit**").

How to setup a given track

- **Assigning a Material (Alternative Method)** and if the "**New Material**" assignment does not work, you can follow this alternative method:

- Open the **zip file** of the mod and locate the "**main.materials.json**" file. This file contains the material definitions for the level.

- Open it in a JSON editor and add a new material by inserting the necessary elements alongside the existing materials.

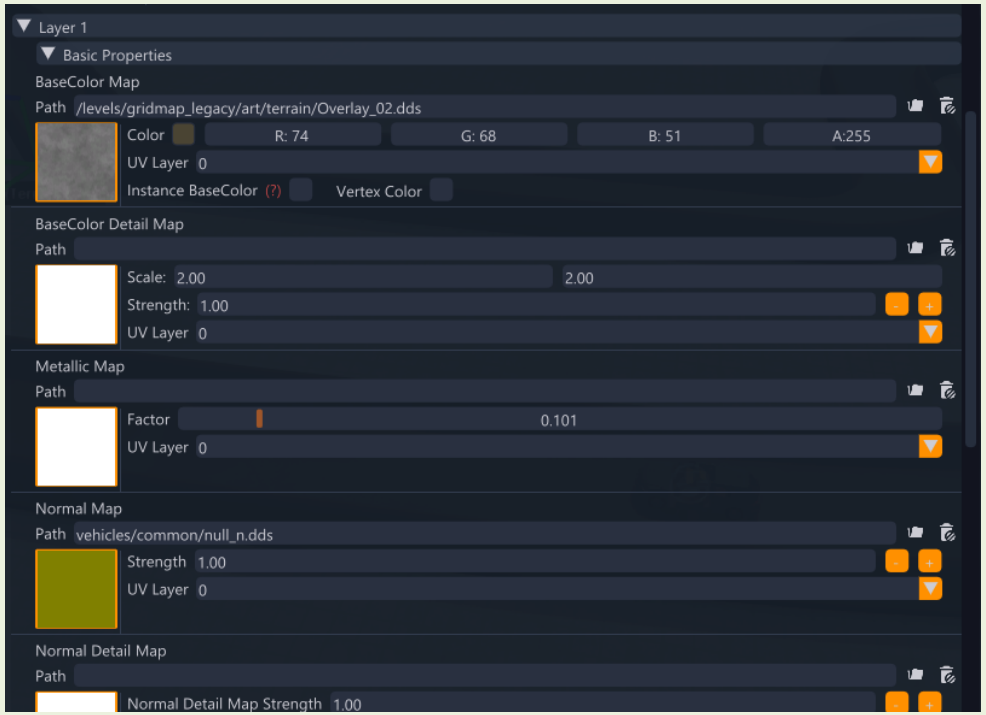
- You can use the material provided on the next page but remember to change the name if you doesn't use the name "MaterialCircuit" for your collada (.dae)

How to setup a given track


```
"MaterialCircuit": {
    "name": "MaterialCircuit",
    "mapTo": "MaterialCircuit",
    "class": "Material",
    "persistentId": "77d93c213ew128f-adwa1c6e-488d-a98d-
b454b99635a2",
    "Stages": [
      {
        "baseColorFactor": [
          0.759438992,
          0.596023977,
          0.0895709991,
          1
        ],
        "baseColorMap": "vehicles/common/null_n.dds",
        "metallicFactor": 0.101000004,
        "normalMap": "vehicles/common/null_n.dds",
        "pixelSpecular": true,
        "roughnessFactor": 0.683000028
      },
      {
        "baseColorFactor": null,
        "metallicFactor": null,
        "pixelSpecular": null,
        "roughnessFactor": null
      },
      {
        "baseColorFactor": null,
        "metallicFactor": null,
        "pixelSpecular": null,
        "roughnessFactor": null
      },
      {
        "baseColorFactor": null,
        "metallicFactor": null,
        "pixelSpecular": null,
        "roughnessFactor": null
      },
      {
        "baseColorFactor": null,
        "metallicFactor": null,
        "pixelSpecular": null,
        "roughnessFactor": null
      }
    ],
    "alphaRef": 0,
    "dynamicCubemap": true,
    "materialTag0": "beamng",
    "materialTag1": "vehicle",
    "translucent": true,
    "translucentBlendOp": "None",
    "version": 1.5
}
```

How to setup a given track

Remember to save the material by pressing the save botton in the Material Editor and to save the level in the upper menu. If everything has been done correctly, you should now see the circuit with the material you created.
In the **Material Editor**, you can also modify the material itself by adjusting various **maps** or **properties**, allowing for further customization of its appearance and behavior.



You can click on any **map image** to change the selected map. Additionally, check the **common folder** for other available maps.

With these steps, you can add everything you need to your level not just the circuit but also other objects like cones or obstacles, for example.
This allows you to customize the environment to fit your simulation needs.

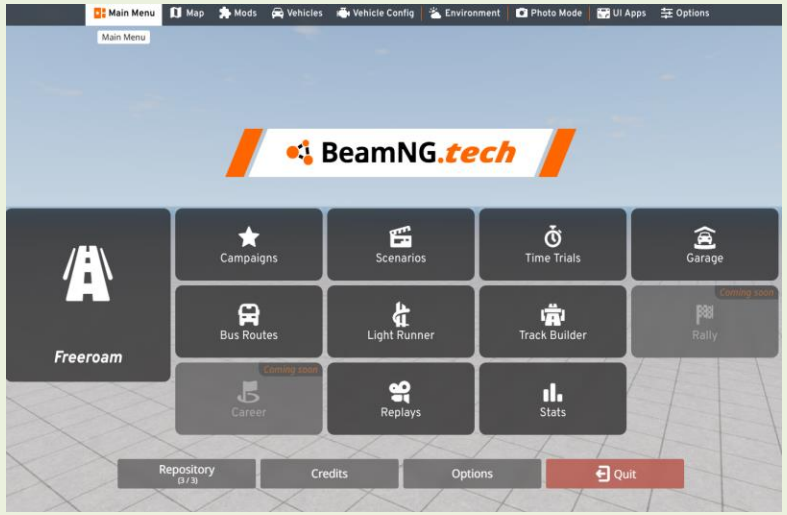
How to create your track

These steps explain how to use a given track, but what if you need to create your own?
To build a custom track, you can use the **Track Builder** tool in BeamNG or **Blender**, a 3D modeling environment. Blender is a powerful, free software that allows you to export custom track models.
You can download Blender for free here: [blender_download]

Before using **Blender**, we first need to design the track using the **Track Builder** tool in BeamNG:

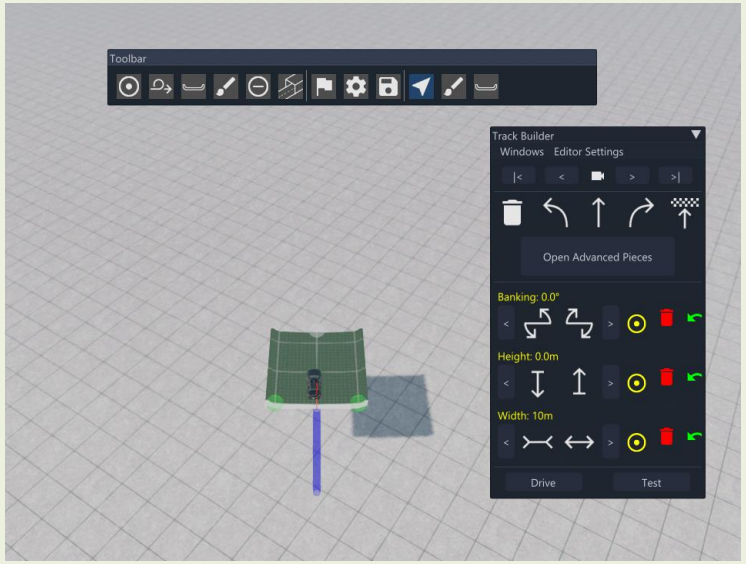
- Open the game and go to the **Freeroam** menu.
- Select your level.
- In the upper menu, go to "**Main Menu**".
- Open the "**Track Builder**" tool.

This tool will allow you to create the initial design of your track before refining it further in Blender.



How to create your track

The game will ask to you where you want to open the truck builder, start the track builder in the current level.



This will open the **Track Builder** menu, where you can design your track and define its key properties, such as **banking, cornering, and width**.
At this stage, don't focus too much on the textures, as you will define them later using materials, as explained earlier.
If you need it, you can create even a road street or an open track.

In the track builder menu you can find a lot of helpful tools for your track creation:

Track building tools

1. **Track builder :** Allows you to place and edit individual track segments. Segments can be curved, straight, or custom-shaped.

2. **Track Shape :** Enables to modify the track structure

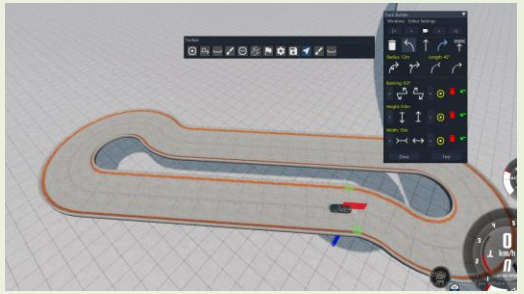
3. **Advanced modifiers :** Allows for precise control over track curvature, width and banking

4. **CheckPoints :** Places checkpoints along the track.

5. **Obstacles :** Lets you place props, barriers, and decorations around the track.

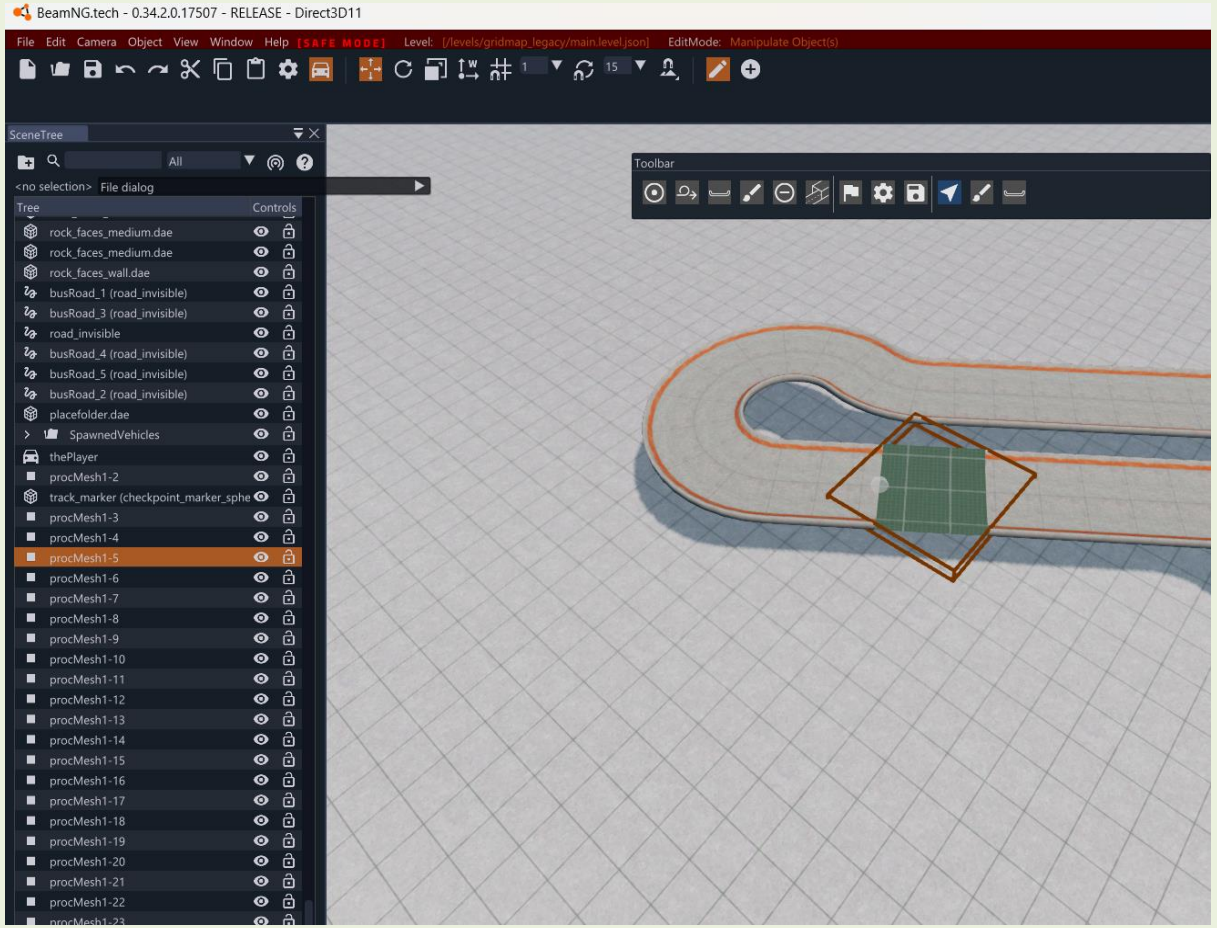
7. **Walls and Ceiling :** Allows to define some other track characteristics.

After creating the circuit, we can proceed to Blender. However, before doing so, we need to convert the circuit into a .dae file.



How to create your track

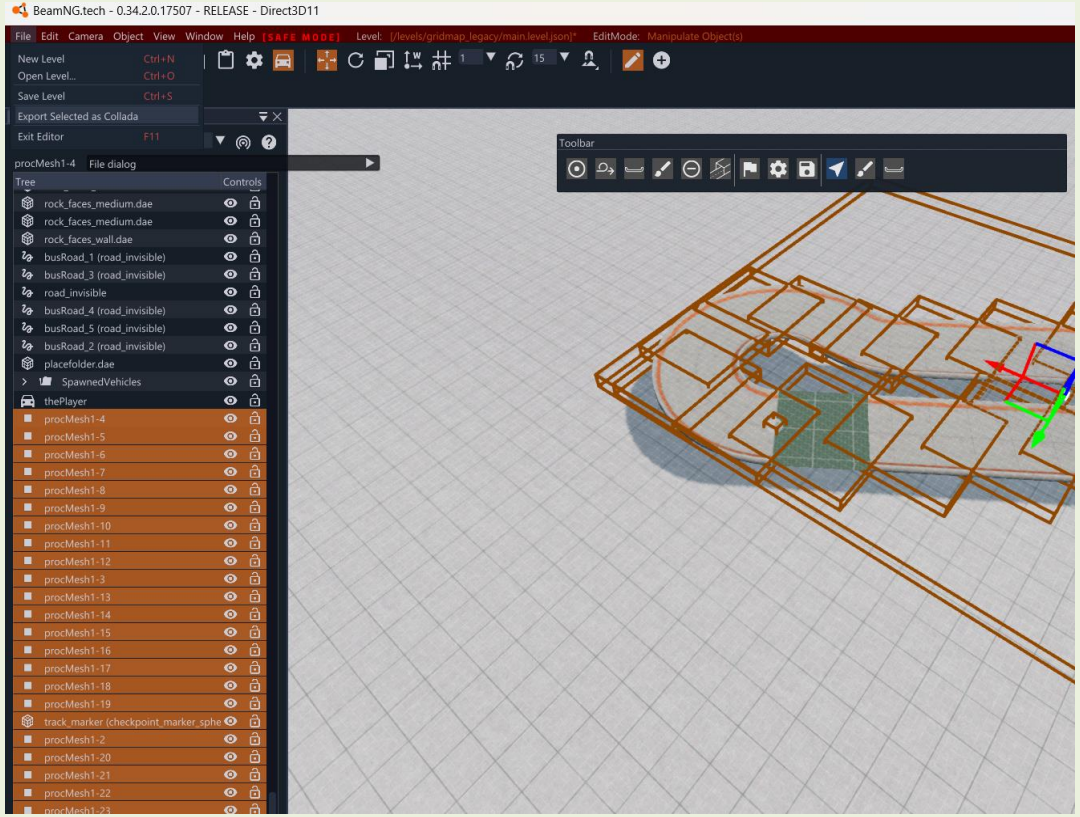
For doing it open, without closing the track editor, the World Editor:

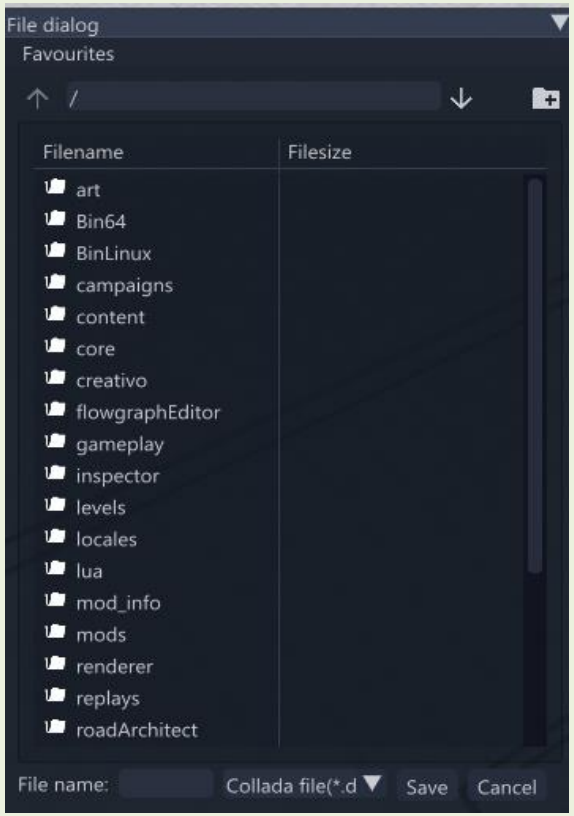


On the left you will see some elements named "procMesh...", that's what we need for the creation of the .dae!

How to create your track

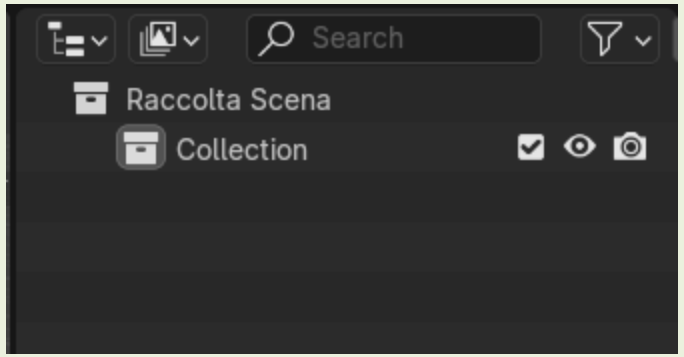
Then, select all the elements, go to the **File** menu, choose **"Export Selected as Collada"**, and save the **.dae** file in your preferred location.

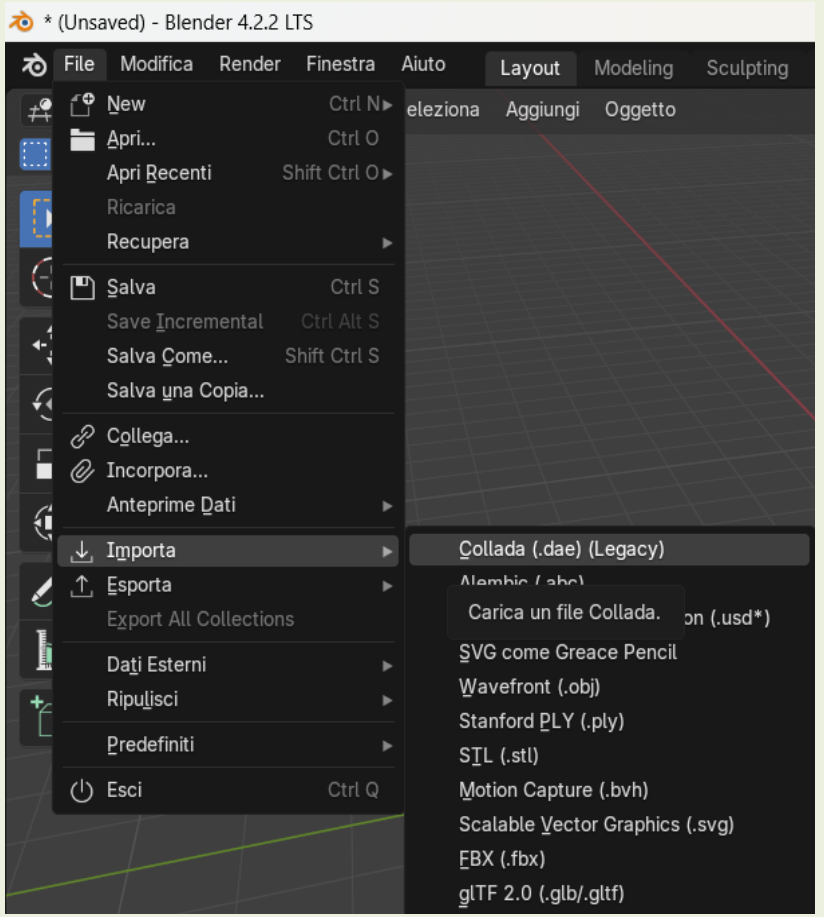




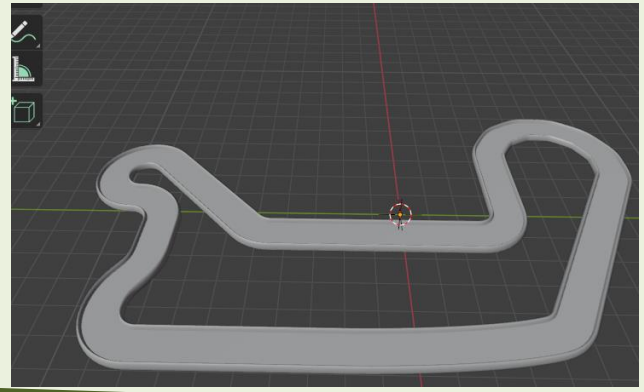
How to create your track

We can now move to Blender. Open Blender, create a new blank file, and delete everything in the scene.





Next, import your **Collada (.dae)** file created from BeamNG.
You can do this by navigating to **File > Import > Collada (.dae).**
Select your Collada and in your scene you will see your track:

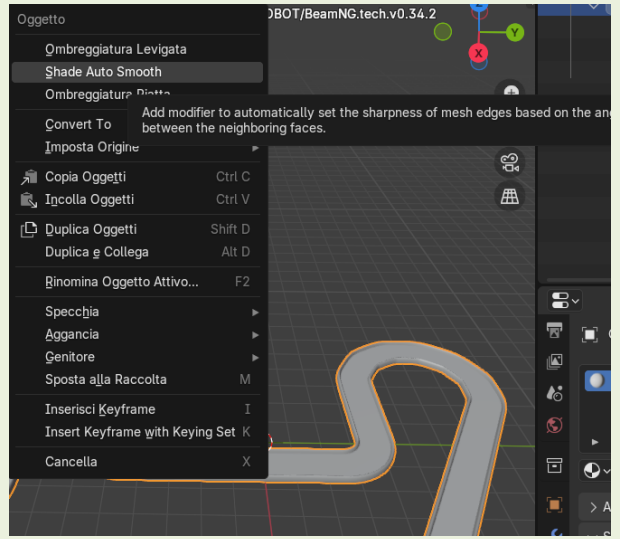


How to create your track

Now, in Blender, we need to complete four more steps:

1. Smoothing the Meshes

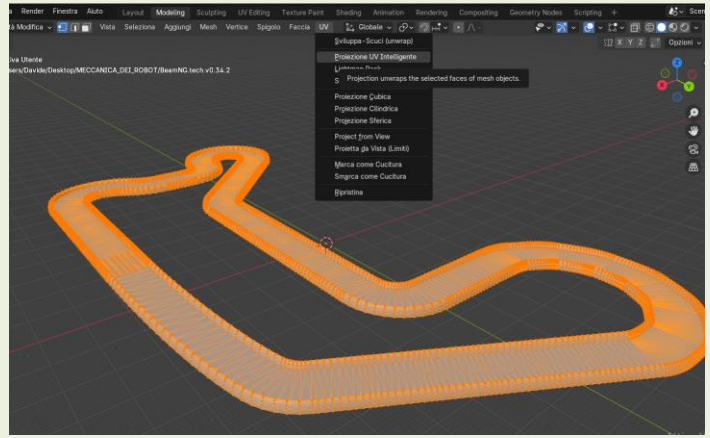
- Select the track.
- Right-click to open the context menu.
- Click "**Shade Auto Smooth**" to smooth the mesh of the **.dae** file.



2. Creating UV Mapping

UV mapping is the process of projecting a 2D texture onto a 3D model, allowing textures to be applied correctly without distortion. To create a UV map:

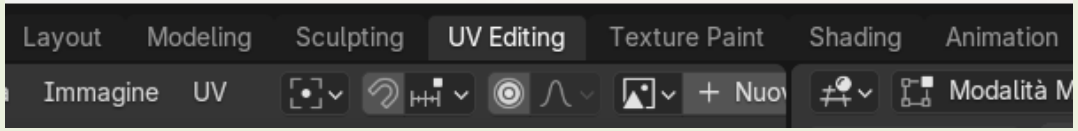
- Select the track and press "**TAB**" on your keyboard to enter **Edit Mode**.
- Press "**A**" to select the entire track.
- Go to the **UV** menu and select "**Smart UV Project**" to automatically generate the UV mapping.



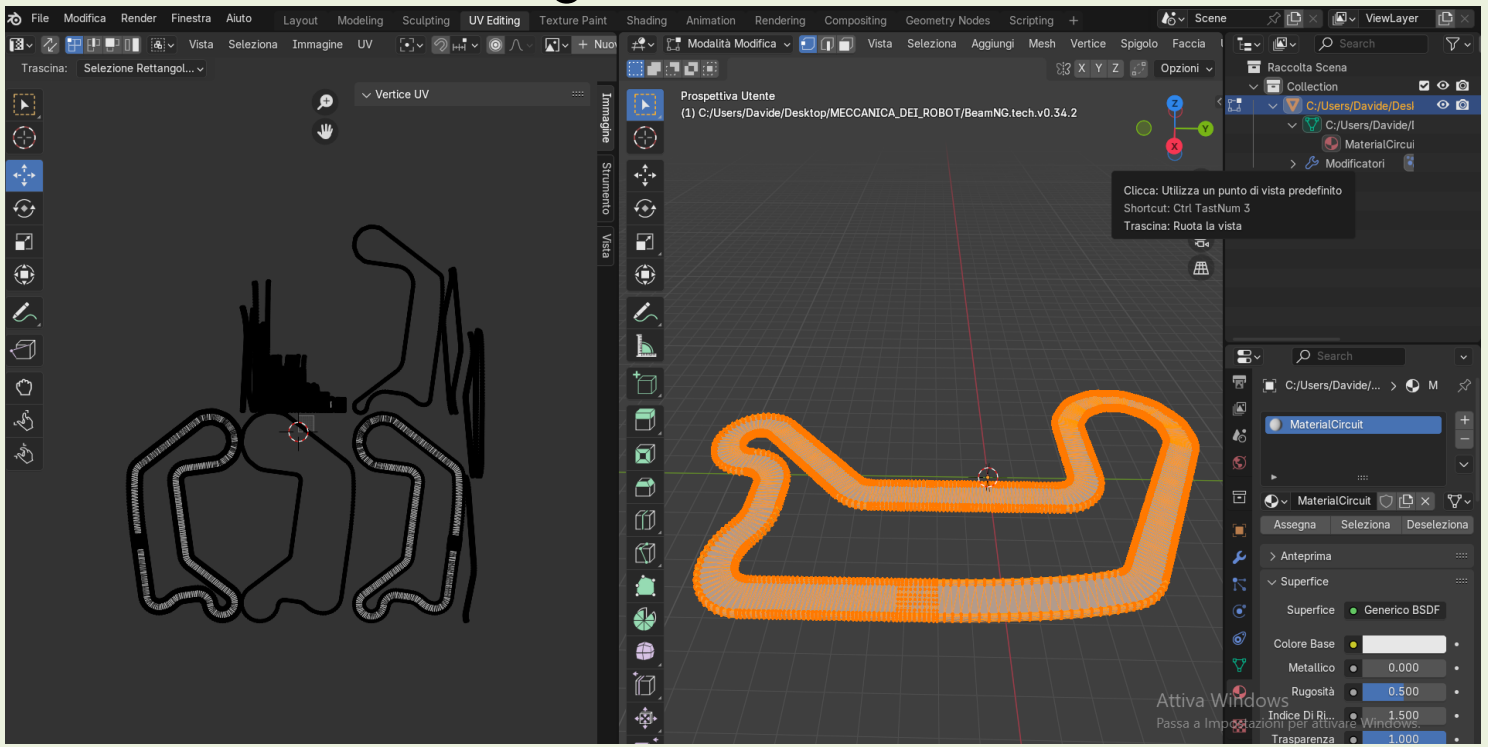
How to create your track

2. Creating UV Mapping

Now change from "modelling" to "UV mapping" mode



You will see something like this:



How to create your track

3. Adjusting the UV Scale

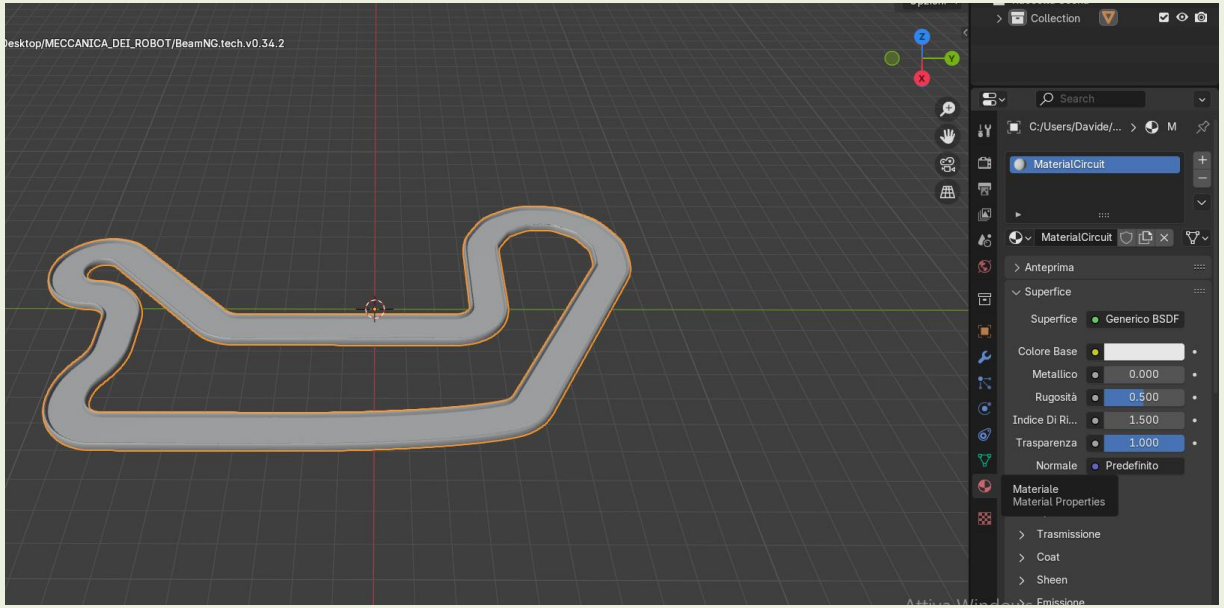
- Keep the entire circuit selected on the right side of the screen (as in the previous step).
- On the left side, in the **UV Editor**, press "**A**" to select the entire UV map.
- Press "**S**" to scale the UV map and make it larger.

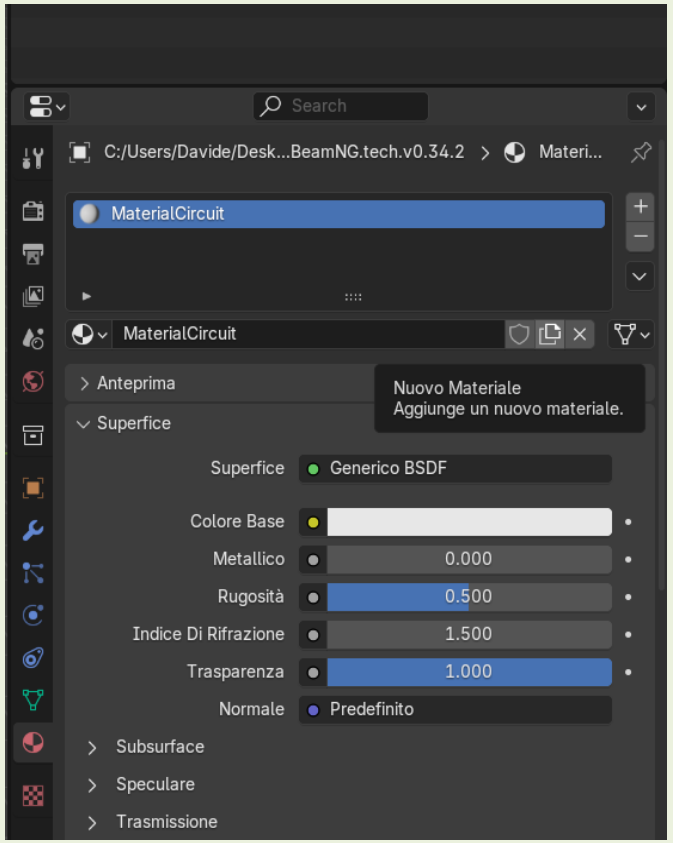
Scale every scale to 20 or 25

How to create your track

4. Assigning a Material

- Switch back to **Object Mode** by pressing "**TAB**" again.





- Open the **Material Properties** panel (found in the right-side menu).
- Click "**New**" to create a new material.
- Assign the material to the track by ensuring it is selected.

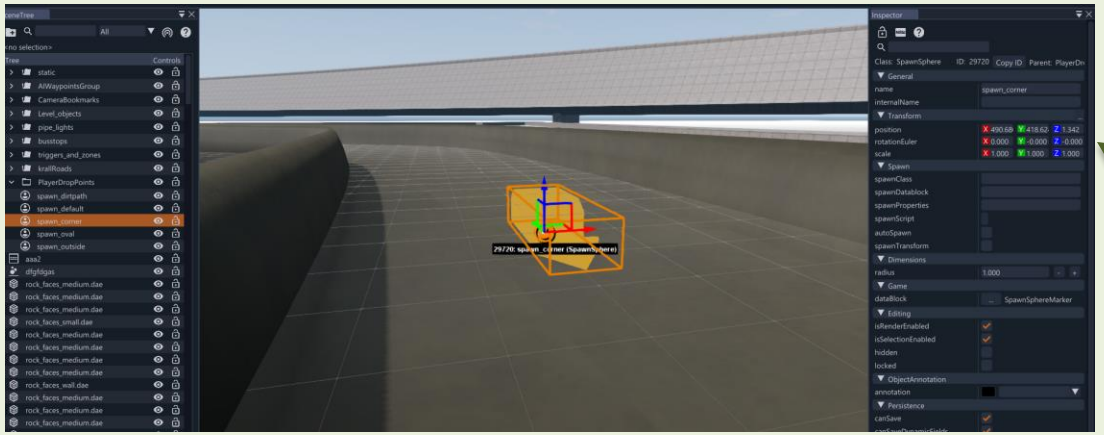
Once done, you can save the Blender file to make future modifications easier.

How to create your track and setup the scenario

Now, we are ready to export the **.dae** file from Blender using the same method we used for importing it. Navigate to the **File** menu, select **"Export"**, and save the new **.dae** file.
You can now follow the same steps as you did for the provided track.

These steps can be repeated multiple times, allowing your algorithms to be tested in various scenarios with a high degree of flexibility and creativity.

Be careful about the name of the material and it's important to take the coordinates for the spawn of your vehicle, open the World Editor again and check where you what to spawn your vehicle:



Save this position for the vehicle spawn position in the scenario !!

What is a scenario?

A **scenario** is a pre-set challenge or mission that places the player in a specific situation with defined objectives. Scenarios can involve tasks such as racing against AI opponents, performing precision or completing obstacle courses.
We can use the scenario as an initializer for the simulation environment. It defines the vehicle's initial position and orientation within the circuit while also configuring various sensors in the scenario script (json file).

A **scenario** is a json file, you have to define:

- The level (the map)
- The position of the vehicle's spawn inside the map (xyz)
- The orientation of the vehicle inside the map (quaternion)
- The vehicle you will use
- The sensors of the vehicle

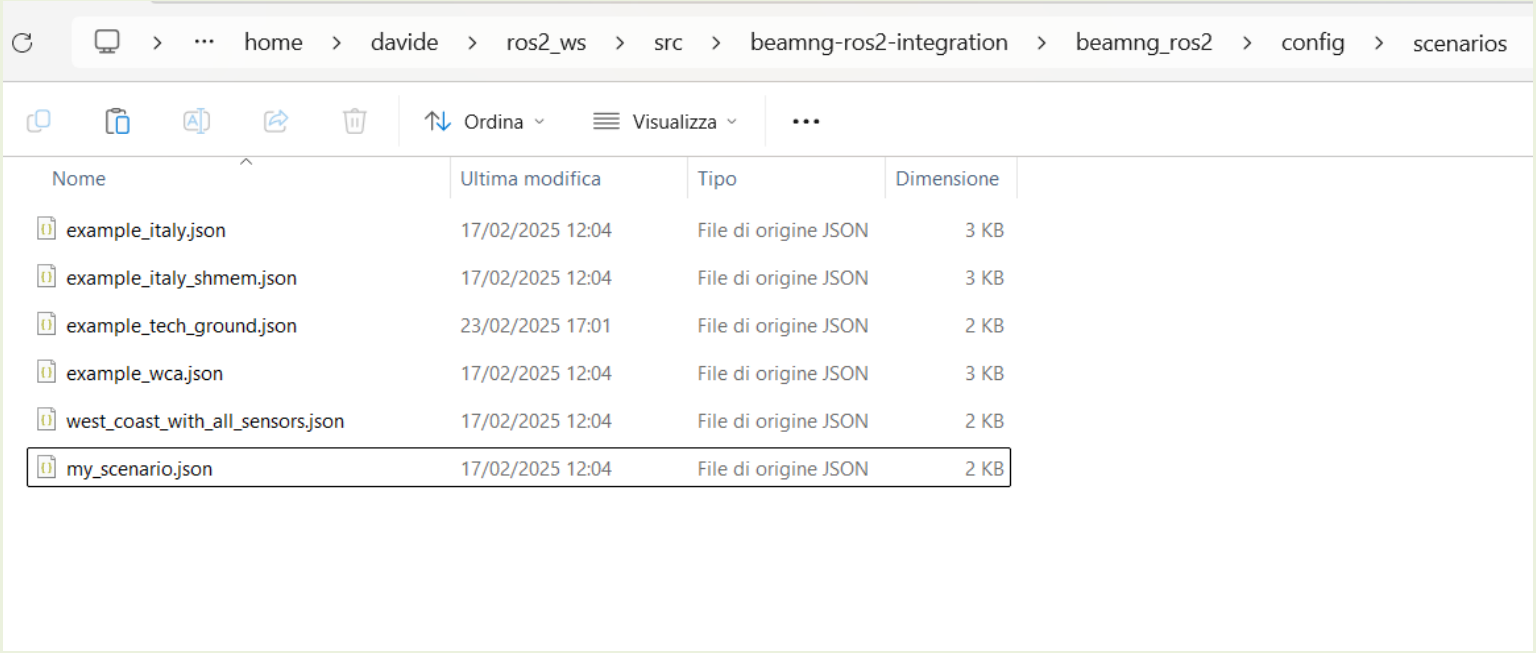
You can also check the example scenario on the folder:

ros2_ws > src > beamng-ros2-integration > beamng_ros2 > config > scenarios

How to write a scenario

After you create your scenario you can put it inside the scenario's folder and press on your Linux terminal:

After t `colcon build` for your project



Example of scenario

example_tech_ground.json

Ubuntu-22.04 > home > davide > ros2_ws > src > beamn

```
{
  "version": 1.0,
  "level": "gridmap_legacy",
  "name": "example_tech_ground",
  "mode": "None",
  "network_visualization": "off",
  "vehicles": [
    {
      "name": "ego",
      "model": "Desert_Buggy",
      "pos": [
        490.680,
        418.68,
        1.100
      ],
      "rot": [
        0.0000,
        0.0000,
        0.9000,
        0.0100
      ],
```

Scenario's level

Scenario's name

Scenario's type: you can create time attack, races of task event

Vehicle selected (you can find the model in the .pc in the vehicle folder

Position (from the world editor)

Orientation

Example of scenario

example_tech_ground.json

Ubuntu-22.04 > home > davide > ros2_ws > src > beamng-ros2-inte

```
7     "vehicles": [
8       {
22        "sensors": [
23          {
26          },
27          {
28            "name": "imu0",
29            "type": "IMU",
30            "pos": [0.0, 0.0, 0.2],
31            "is_send_immediately": true
32          },
33          {
34            "name": "ultrasonic_front",
35            "type":"Ultrasonic.default",
36            "pos": [0.0, 0.089, 0.3],
37            "dir": [0.0, 1.0, 0.0],
38            "is_visualised": false,
39            "range_min_cutoff": 0.15,
40            "range_direct_max_cutoff": 15.0,
41            "requested_update_time": 0.02
42          },
43          {
44          "name": "ultrasonic_right",
45          "type":"Ultrasonic.default",
46          "pos": [0.0371, 0.089, 0.4],
47          "dir": [1.0, 0.0, 0.0],
48          "is_visualised": false,
49          "range_min_cutoff": 0.15,
50          "range_direct_max_cutoff": 15.0,
51          "requested_update_time": 0.02
52          },
53          {
54          "name": "ultrasonic_left",
55          "type":"Ultrasonic.default",
56          "pos": [0.0371, 0.089, 0.4],
57          "dir": [-1.0, 0.0, 0.0],
58          "is_visualised": false,
```

IMU sensor (acceleration , angular speed)

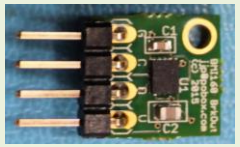
Front, right and left ultrasonic sensors (distance with the wall)

You can add as many sensors as you want to the file by simply inserting them in the code alongside the other sensors.
You can also modify the sensor properties, such as the **Field of View (FOV)** for cameras, the **direction**, or the **visualization** settings.

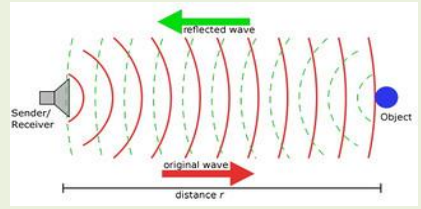
Sensors

BeamNG.tech provides a wide range of sensors, powerful tool for vehicle simulation, autonomous driving research, and robotics applications. The available sensors include:

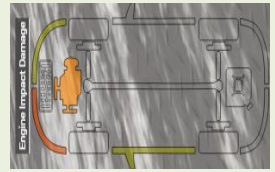
- **Advanced IMU (Inertial Measurement Unit)** : Combines accelerometer and gyroscope.



- **Ultrasonic Sensor** : Uses sound waves to measure short-range distances, useful for obstacle detection.



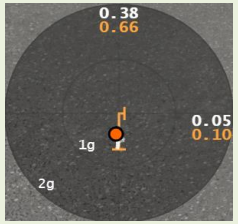
- **Damage Sensor** : Monitors and reports structural damage sustained by the vehicle during simulations.



- **Electrics Sensor** : Provides data on various vehicle electrical parameters, such as speed, fuel level, temperature, and lighting status.

Sensors

- **GForces Sensor** : Measures the G-forces acting on the vehicle, helping in vehicle dynamics analysis.



- **Timer Sensor** – Tracks elapsed time during a simulation.
- **GPS Sensor** – Provides precise geographical positioning data, essential for navigation and SLAM algorithms.

- **Lidar Sensor** – Generates 3D point cloud data of the surrounding environment, for mapping and obstacle detection.

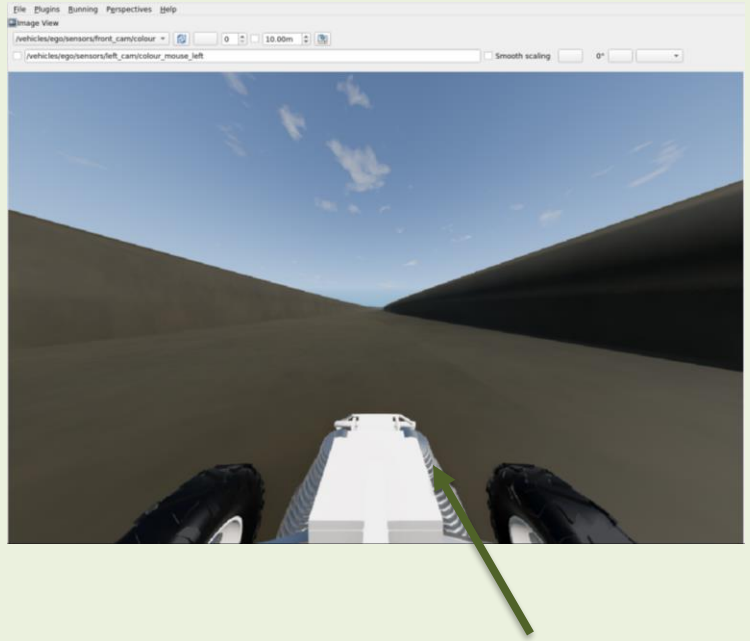
- **Radar Sensor** – Detects objects and measures their relative speed, useful for driver assistance systems.

Sensors

- **Camera Sensor** – Simulates the vehicle's cameras, capturing images or videos for computer vision applications. The camera can also determine the distance based on pixel data.

- **Powertrain Sensor** – Monitors the performance of the vehicle's powertrain system.

- **Roads Sensor** – Provides information about road characteristics, such as lane geometry and surface conditions.



You can see it also on the rqt plugin "camera view"

Sensors

In the **Drive** folder, you will find a JSON file named "**example_of_sensors**", where all sensors are defined along with their **modifiable characteristics**. You can use this file as a starting point for implementing your own sensors.

Other information are present on the BeamNG documentation:

BeamNG_Sensors_Documentation

Or in the Git Hub documentation page:

ROS_Sensor_documentation

You can also check the default sensors file presents in the folder:

Implementation of Vehicle Control through BeamNG-ROS2 Integration

Defining and running custom vehicle control scripts in the simulation.

BeamNG

Chapter index

Create the Script

For the vehicle control script, it is sufficient to create a Python script (in our case `PID_control_buggy`) and place it in the following folder: `~/ros2_ws/src/beamng-ros2-integration/beamng_ros2/beamng_ros2`.

But before going into the details of the script used for our tests it is necessary to define an **entry point**.

Adding an Entry Point

In ROS 2, the **entry point** is the executable defined in the setup.py file (for Python packages) or in the CMakeLists.txt file (for C++ packages). It tells ROS2 which script or binary should be executed when running the command:

```
ros2 run <package_name> <executable_name>
```

To add the entry point for the script we will use, on Windows, open the folder `~/ros2_ws/src/beamng-ros2-integration/beamng_ros2`, then open the `setup.py` file and add the following line in the `entry_points` section as shown:

```
entry_points={
    "console_scripts": [
        "beamng_bridge = beamng_ros2.beamng:main",
        "example_client = beamng_ros2.examples.example_client:main",
        "PID_control_buggy = beamng_ros2.PID_control_buggy:main",
    ],
},
```

Then save the file.

Test Script Overview

This script is designed to control a vehicle using a PID controller for speed and steering adjustments. The vehicle uses three **ultrasonic sensors** for distance measurement and an **IMU** for detecting acceleration, to navigate along a track while maintaining a constant speed.

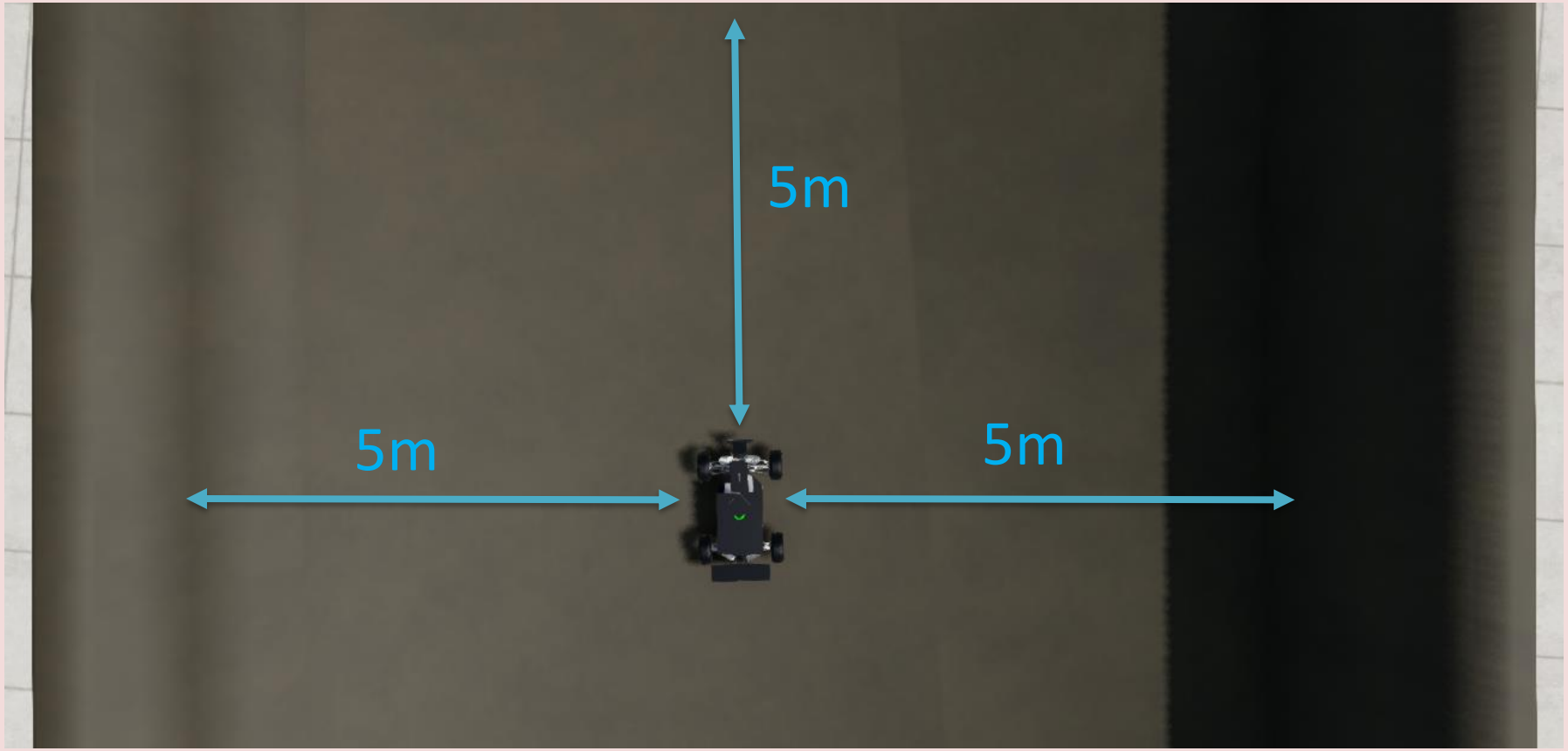
The goal is to keep the vehicle at a desired speed and ensure it stays a fixed distance from the track's walls by making continuous adjustments to the **steering** and **throttle**.

The script listens to sensor data in real-time, calculates the necessary control signals using the PID controller, and sends those signals to the vehicle, ensuring smooth and accurate navigation throughout the simulation.

In the following section the various parts of the test script used will be explored in greater detail.

Test Script Overview

Inte the picture, the ultrasonic sensors' range and desired trajectory of the vehicle are shown.



Test Script Overview

1 Importing Libraries and Setting Constants

```
import sys
import rclpy
import math
import os
from rclpy.node import Node
from geometry_msgs.msg import Twist
from sensor_msgs.msg import Range, Imu
from beamng_msgs.msg import TimeSensor, VehicleControl
import beamngpy as bngpy

NODE_NAME = "vehicle_control"
EXP_LIMIT = 500
```

This section imports the necessary Python libraries and modules that are used throughout the script. It includes ROS2-specific libraries (`rclpy` for ROS2 node handling), geometry and sensor messages (like `Range` for sensor data), and the BeamNGpy library to interface with BeamNG.tech.

Test Script Overview

2 PID Controller Class

```
class PIDController:
    def __init__(self, kp, ki, kd, integral_limit=10.0):
        self.kp = kp
        self.ki = ki
        self.kd = kd
        self.prev_error = 0.0
        self.integral = 0.0
        self.integral_limit = integral_limit



    def compute(self, error, dt):
        self.integral += error * dt
        self.integral = max(-self.integral_limit, min(self.integral, self.integral_limit))
        derivative = (error - self.prev_error) / dt if dt > 0 else 0.0
        output = self.kp * error + self.ki * self.integral + self.kd * derivative
        self.prev_error = error
        return output
```

Test Script Overview

In this section, the script defines the class that implements a PID controller and it is crucial for controlling the vehicle's speed and steering, ensuring that it stays within the desired parameters.

The constructor `__init__` initializes the PID with `kp`, `ki`, and `kd` (proportional, integral and derivative gain). An `integral_limit` is also set, which defines the maximum allowed value for the integral component of the controller.
The method `compute`, computes the output of the PID based on the `error` value (the difference between the target and the current state) and the time step `dt`.
The `integral` accumulates the `error` over time, while the `derivative` calculates the rate of change of the `error`.
The output is a combination of the proportional, integral, and derivative components.
The computed `output` is returned, which is used to adjust either the vehicle's speed or steering.

Test Script Overview

3.1 `VehicleControlNode` Class Initialization

```
class VehicleControlNode(Node):
    def __init__(self):
        super().__init__(NODE_NAME)


        host = self.declare_parameter("host", "192.168.1.217").value
        port = self.declare_parameter("port", 25252).value
        vehicle_id = self.declare_parameter("vehicle_id", "ego").value
```

Remember to replace the IP address with that of the machine the simulator is running on to make the script work

This section initializes the `VehicleControlNode` class, which is responsible for managing the connection between ROS 2 and BeamNG for vehicle control.

The class inherits from ROS2's `Node` class to function as a ROS node.

- `host`: declares the IP address for connecting to BeamNG, it needs **your Windows IPv4 address**
- `port`: declares the port number (default is `25252`)
- `vehicle_id`: declares the ID of the vehicle to control in BeamNG (default is `ego`).

Test Script Overview

```
        if not vehicle_id:
            self.get_logger().fatal("No Vehicle ID given, shutting down node.")
            sys.exit(1)
        self.game_client = bngpy.BeamNGpy(host, port)
        try:
            self.game_client.open(listen_ip="*", launch=False, deploy=False)
            self.get_logger().info("Successfully connected to BeamNG.tech.")
        except TimeoutError:
            self.get_logger().error("Could not establish game connection.")
            sys.exit(1)
        current_vehicles = self.game_client.get_current_vehicles()
        if vehicle_id not in current_vehicles:
            self.get_logger().fatal(f"No vehicle with id {vehicle_id} exists.")
            sys.exit(1)
        self.vehicle_client = current_vehicles[vehicle_id]
        try:
            self.vehicle_client.connect(self.game_client)
            self.get_logger().info(f"Connected to vehicle {self.vehicle_client.vid}")
        except TimeoutError:
            self.get_logger().fatal("Could not establish vehicle connection.")
            sys.exit(1)
```

Test Script Overview

The parameters declared for the `host`, `port`, and `vehicle_id`, are needed to connect to the BeamNG simulation using **BeamNGpy**.
In this part the script checks if a valid vehicle ID is provided and attempts to establish a connection with the BeamNG.tech simulation.
If the connection is successful, it retrieves the current vehicles and connects to the specified vehicle, allowing control over it.

Test Script Overview

3.2 Setting Up ROS2 Subscriptions

```
self.create_subscription(VehicleControl, "/control", self.send_control_signal, 10)
self.create_subscription(Imu, 'vehicles/ego/sensors/imu0', self.imu_listener_callback, 10)
self.create_subscription(TimeSensor, 'vehicles/ego/sensors/time0', self.time_listener_callback, 10)
self.create_subscription(Range, 'vehicles/ego/sensors/ultrasonic_left', self.left_listener_callback, 10)
self.create_subscription(Range, 'vehicles/ego/sensors/ultrasonic_right', self.right_listener_callback, 10)
self.create_subscription(Range, 'vehicles/ego/sensors/ultrasonic_front', self.front_listener_callback, 10)
```

The `create_subscription` method is used to subscribe to various **topics**.
In ROS, a **subscription** allows a node to receive messages from specific topics that are being published by other nodes. When a message is published to a topic, all nodes that are subscribed to that topic will receive the message and process it.
In this case, the script subscribes to the vehicle's sensors (topics) and control signals. Each subscription links a specific topic to a **callback function**.
Whenever new data is published on a topic, the corresponding callback function is triggered to process that data.
This architecture enables asynchronous communication, where the vehicle node can process incoming messages as they are received without needing to block or wait for new data.

Test Script Overview

3.3 PID Controllers for Speed and Steering

```
self.speed_pid = PIDController(0.00001, 0.001, 0.0005)
self.steering_pid = PIDController(0.05, 0.001, 0.0005)
self.target_distance = 5
self.target_speed = 2.0
self.current_speed = 0.0
self.last_time = None
self.ultrasonic_left = None
self.ultrasonic_right = None
self.ultrasonic_front = None
```

In this section, the script initializes two PID controllers—one for controlling the vehicle's **speed** and one for **steering**.
It also defines target values for distance and speed, as well as variables to store sensor readings and current speed.

Test Script Overview

3.4 Control Signal "Publisher" and Control Commands Logic

```
def send_control_signal(self, signal):
    self.vehicle_client.control(
        steering=signal.steering,
        throttle=signal.throttle,
        brake=signal.brake,
        parkingbrake=signal.parkingbrake,
        clutch=signal.clutch,
        gear=signal.gear,
    )
```

This method publishes control signals to the vehicle, including steering, throttle, brake, and gear settings based on the values received from the control system.

However, in this regard an important clarification must be made on how the commands could be given to the vehicle.

This method for controlling the vehicle differs from using a **publisher** in how the control commands are applied to the vehicle and the flow of data between nodes.

Test Script Overview

In the subscriber-publisher model of ROS, commands are typically sent from one node to another by **publishing** messages on a **topic**. The node responsible for controlling the vehicle would publish control commands (such as **steering** or **throttle** values) to a specific topic, and another node (acting as the vehicle interface) would subscribe to that topic, receive the commands, and apply them to the vehicle.

In contrast, the `send_control_signal` method directly applies the control commands to the vehicle **client** using the `control()` method from `beamngpy`. This method is called whenever a new message is received on the `/control` topic thanks to the subscription.

In this case the node is both listening for incoming messages (subscriber) and acting immediately on those messages by sending commands directly to BeamNG's simulation interface via beamngpy and commands are executed as soon as they are received without needing an intermediary node to republish the commands.

Test Script Overview

3.5 Sensor Data Callbacks

```
def imu_listener_callback(self, msg: Imu):
    a_x = -msg.linear_acceleration.x
    self.current_speed += a_x * 0.1

def time_listener_callback(self, msg: TimeSensor):
    current_time = msg.beamng_simulation_time.sec + msg.beamng_simulation_time.nanosec * 1e-9

    if self.last_time is not None:
        dt = current_time - self.last_time
        self.control_vehicle(dt)

    if self.last_time is not None and current_time < self.last_time:
        self.get_logger().info("Level reset detected. Resetting variables...")
        self.last_time = current_time
        self.current_speed = 0.0
        self.ultrasonic_left = None
        self.ultrasonic_right = None
        self.ultrasonic_front = None
        self.last_time = current_time
```

Test Script Overview

3.5 Sensor Data Callbacks

```
    def left_listener_callback(self, msg: Range):
        self.ultrasonic_left = msg.range
    def right_listener_callback(self, msg: Range):
        self.ultrasonic_right = msg.range
    def front_listener_callback(self, msg: Range):
        self.ultrasonic_front = msg.range
```

These methods are the callbacks for all sensor data.

The time callback helps to calculate the time difference between updates and manage control loops.
The first `if` condition makes sure the control system updates based on the time elapsed between two messages and the second detects when the simulation is reset, preventing inconsistencies that could arise from outdated sensor data or incorrect time steps.

Test Script Overview

3.6 Vehicle Control Logic

```
def control_vehicle(self, dt):
    if self.ultrasonic_left is None:
        self.ultrasonic_left = self.target_distance
    if self.ultrasonic_right is None:
        self.ultrasonic_right = self.target_distance
    if self.ultrasonic_front is None:
        self.ultrasonic_front = self.target_distance

    steering_error = self.ultrasonic_right - self.ultrasonic_left

    if abs(steering_error) < 0.02:
        steering_error = 0.0

    scaling_factor = max(1.1, (self.target_distance - self.ultrasonic_front))
    steering = self.steering_pid.compute(steering_error * scaling_factor, dt)
    steering = max(-0.5, min(steering, 0.5))
```

Test Script Overview

```
speed_error = self.target_speed - self.current_speed
throttle = self.speed_pid.compute(speed_error, dt) if speed_error > 0 else 0.5
throttle = max(0.05, min(throttle, 0.1))
brake = 0.0
gear = 2

self.get_logger().info(f"steering: {steering:.3f} throttle: {throttle:.3f}")

self.send_control_signal(VehicleControl(
        throttle=throttle,
        steering=steering,
        brake=brake,
        parkingbrake=0.0,
        clutch=1.0,
        gear=gear
        ))
```

This section contains the main control loop. It calculates **errors** for **steering** and **speed**, adjusts them using the PID controllers, and then **sends the control commands** to the vehicle.

Test Script Overview

4 Main Function to Run the Node

```
def main(args=None):
    rclpy.init(args=args)
    rclpy.spin(VehicleControlNode())
    rclpy.shutdown()
```

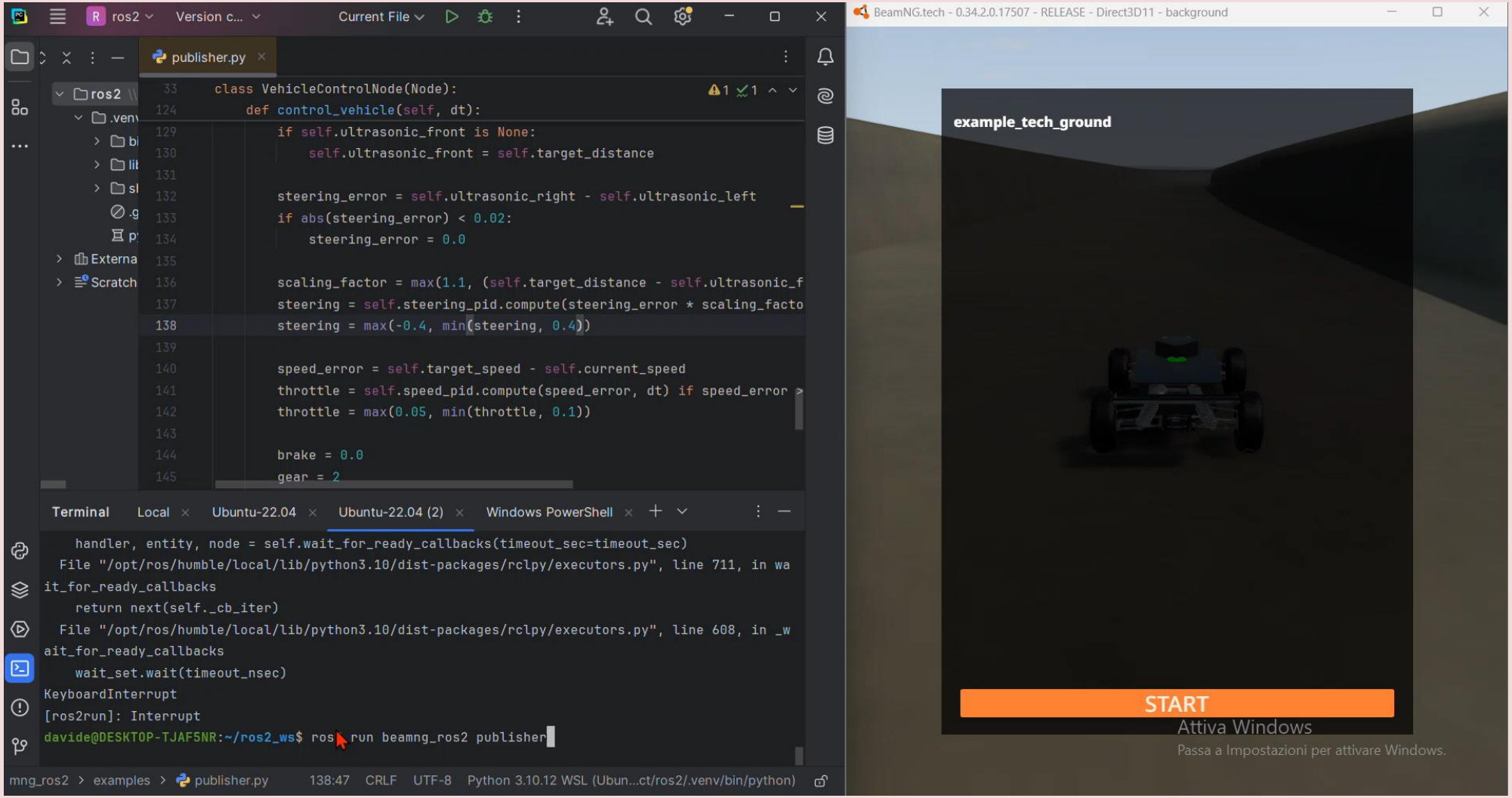
This is the entry point of the script. It initializes ROS 2, runs the vehicle control node, and shuts down ROS 2 after the node finishes running.

Vehicle Control in Action

The video in the next slide shows the integration between ROS 2 and BeamNG.tech through the described control script. On the left, the terminal executes the command to launch the executable.
On the right, the simulation visualizes the vehicle moving along the track, maintaining a constant speed while keeping a safe distance from the walls using ultrasonic sensors and a PID controller.

The vehicle is able to complete multiple laps without collisions or interruptions, demonstrating the effectiveness of the implemented control logic.

Vehicle Control in Action



BeamNG

Considerations on the Environment

Regarding the BeamNG-ROS2 environment, we can conclude:

- The environment is stable, and the scenario can be easily restarted by pressing the **R** key.
- Integration can be done without the need for a virtual machine, eliminating concerns about changing the operating system.
- The maximum writing frequency for the sensor topics we observed was around **30 Hz**, which is sufficient but not the fastest, particularly for sensors like the **IMU**.
- The algorithms can be written in Python, leaving the freedom to use all the Python's libraries and an easier coding language.
- There is a lot of freedom also for the track or map and scenario realization, you can create a race, a time attack or a task event.
- The simulated vehicle can be used, with his 1:1 scale, for testing of the AI driving algorithm meant for the real vehicle.

Understanding BeamNG: Simulator Mechanics & Vehicle Modeling

Understanding BeamNG.tech and Building a Custom Vehicle

BeamNG

Introduction

The following section of the guide focuses on **BeamNG.tech** and has been divided into two parts for clarity:

- **Part 1: BeamNG.tech**

The first part covers the **simulator's usage**, **physics engine** and **jbeam files syntax**

- **Part 2: Vehicle Creation**

The second part provides a **step-by-step walkthrough** of how we created a 1:1 scale model of the "Losi - DBXL-E 2.0"

The concepts explained in this guide focus on aspects that were useful for **our implementation**, rather than covering everything that can be done with the simulator. Additionally, all the tools that may be useful for model creation, debugging and simulation will be presented.
This guide is meant to support learning, and it is strongly recommended to have the simulator, programs, or files shown in the guide readily available for a more effective understanding.

Part 1: BeamNG.tech

1. Essential Basics

- Game Modes
- User Interface and UI Apps
- Camera Modes, Slow Motion and Controls
- Nodegrabber, Teleporting Vehicles, Reset
- How to Install and Manage Mods
- Creating Vehicle Configurations

3. Tutorials

- Basic Car Tutorial – Autobello Kit Car

2. Modding

- Introduction to JBeams
- JBeam Syntax
- The Part/Slot System
- Debug Tools
- Common JBeam Issues
- Vehicle Modeling Guidelines
- JBeam File Sections:

JBeam File Sections:

- Nodes
- Beams
- Triangles
- Hydros
- Information
- Slots
- Flexbodies
- Camera
- Props
- Pressure Wheels
- Variables
- Refnodes
- Controller
- Energy Storage
- Powertrain

Part 1: BeamNG.tech

This section of the guide is entirely based on the concepts provided and explored in greater detail in the official BeamNG documentation, which can be fully accessed at the following link.

BeamNG Documentation

The goal of this guide is to introduce users to the capabilities of the simulator and provide them with the necessary skills to understand the scripts and processes shown in **Part 2** for vehicle creation.

1. Essential Basics

This subsection provides beginners with essential information on game modes, menu navigation, controls, troubleshooting and more, starting from the **Main Menu**, immediately after launching the simulator.

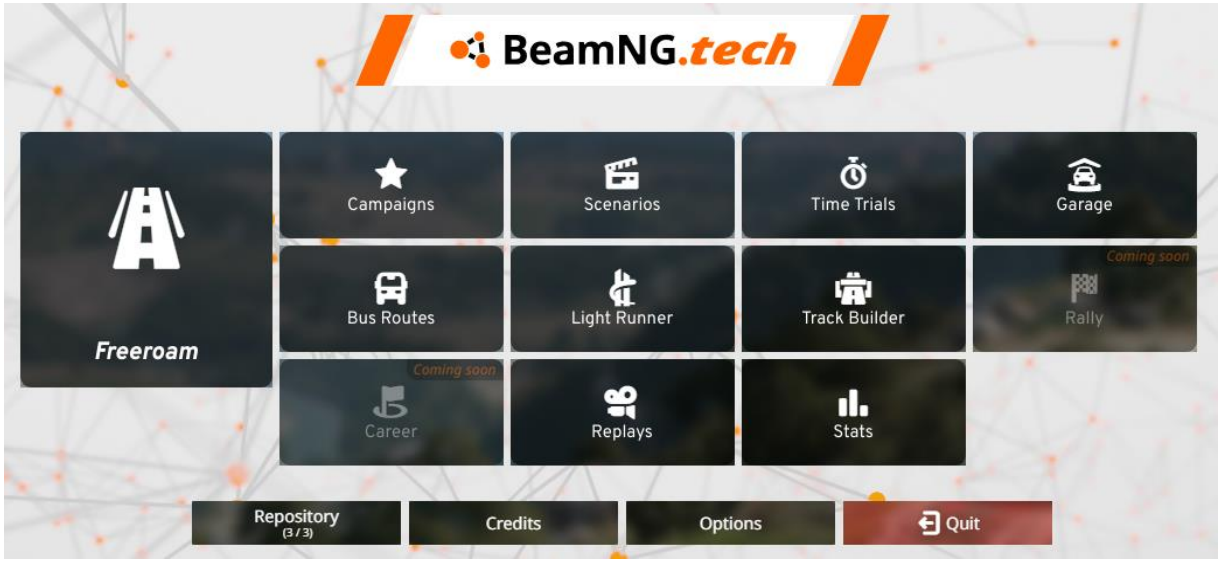
Game Modes

Campaigns, Scenarios and **Time Trials** offer a great way to progressively familiarize yourself with the game, starting with simple challenges and gradually increasing in difficulty.

Garage mode allows users to inspect vehicles, modify parts, adjust setups, and change paint colors.

Freeroam mode allows you to freely explore a chosen map, with the option to select the spawn point for your vehicle. It is the most intuitive mode for **quickly testing a vehicle** and experimenting with different scenarios.

Track Builder mode allows you to create a custom track and test it. A detailed guide on this mode can be found in the [specific section of our manual].



1. Essential Basics

User Interface

Pressing the **ESC key** brings up two very useful menus. The first is called **Side Menu**, on top of the screen and contains the following items (from left to right):

- *Main Menu*: displays the main menu
- *Map*: overview of the current level, displaying all the available missions, allowing to teleport around, and so on..
- *Mods*: in-game mod repository and mod manager
- *Vehicles*: vehicle selector from where you can change or spawn new vehicles, trailers and props
- *Vehicle Config*: vehicle parts selector also containing the tuning menu and configuration save/load menu
- *Environment*: settings for the time of day, weather, gravity and so on..
- *Photomode*: mode specifically designed to take artistic screenshots of the game
- *UI Apps*: edit, modify UI apps, as well remove/add them from the current layout

The second is the **Radial Menu**, and some of the useful items are (it depends on the vehicle chosen):

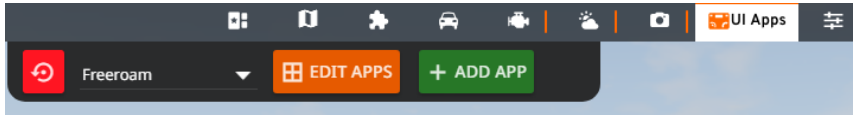
- *Manage*: controls for managing vehicle and spawns
- *Powertrain*: controls for all powertrain options (gearbox mode, 4WD, rangebox) and engine ignition
- *Load/Save*: load previously saved vehicle or saves vehicle in its current form

1. Essential Basics

User Interface

The **default UI** shown (the game mode is *Freeroam* in *Gridmap v2*) is activated by default in all game modes and contains:

- *Messages*: space for alerts
- *Vehicle Damage app*: displays current state of different vehicle systems
- *Race Time app*: displays a timer if the game mode or scenario includes one
- *Race Countdown*: for start of a race/task
- *Force Induction*: displays pressure gauge (only for applicable vehicles)
- *Tacho 2 (Tachometer)*: displays RPM and speed of the vehicle
- *Simple Powertrain Control*: allows selecting powertrain modes (only for applicable vehicles)



The UI is fully customizable, allowing you to add new apps beyond the default ones and adjust their position and size.
This can be done by pressing **ESC** > **UI Apps**, then clicking on **Edit Apps** and **Add Apps**.

1. Essential Basics

Camera Modes and Slow Motion

The simulator offers seven camera modes, you can switch between them using **keys 1-7**.
Two very useful modes for debugging are Free Camera and Slow Motion.

You can activate **Free Camera** by pressing **Shift + C**: it allows you to move the view freely across any point of the environment and move around the vehicle model to better visualize the node and beam structure, if necessary.
To activate **Slow Motion**, press **Alt +** ↑ or **Alt +** ↓: it allows you to analyze frame by frame what happens to the model before it becomes unstable, which is particularly useful when testing unfinished or unstable designs.
You can increase the motion speed with **Alt +** →, and decrease it with **Alt +** ←. The game will slow down by x2/x4/x8/x16/x100 times.

Controls

Driving

- ↑ - Throttle
- ↓ - Brake
- ← - Steering (left)
- → - Steering (right)
- Space - Parking BrakeSpace
- P - Parking Brake (permanent)
- LShift - Clutch
- X - Shift Up
- Z - Shift Down
- Q - Toggle Gearbox Mode
- V - Engine Starter Motor
- H - Horn
- N - Headlights
- Alt + N - Foglights
- , - Left Signal
- . - Right Signal
- / - Hazard Lights
- M - Lightbar Lights

- Ctrl + B - Visualizes beams
- Ctrl + N - Visualizes nodes
- Ctrl + M - Visualizes nodes' names

Gameplay

- Shift + R - Reset all vehicle physics
- R / I - Reset physics
- Insert - Recover Vehicle
- Ctrl + Tab - Switch to previous vehicle
- Tab - Switch to next vehicle
- Ctrl + Home - Set home position
- Home - Load home position
- F8 - Drop Camera at Player
- F7 - Drop Player at Camera
- E - Enter/Exit vehicle

General

- J - Pause/Resume physics
- Alt + F4 - Quit the game
- Alt + Enter - Toggle Fullscreen
- Alt + U - Hide UI

1. Essential Basics

Node Grabber , Teleporting Vehicles, Reset

The **Node Grabbing** tool allows you to manipulate a vehicle's nodes using the cursor, making it useful for various testing and debugging tasks. You can grab, drag, pin, and attach nodes.

To grab a node, hold **CTRL**, hover over the vehicle to highlight nodes, then click and hold **Left Mouse Button**.
To adjust **grab strength**, use the **Mouse Wheel** while holding a node.
To **pin a node**, press the **Middle Mouse Button** while grabbing it.

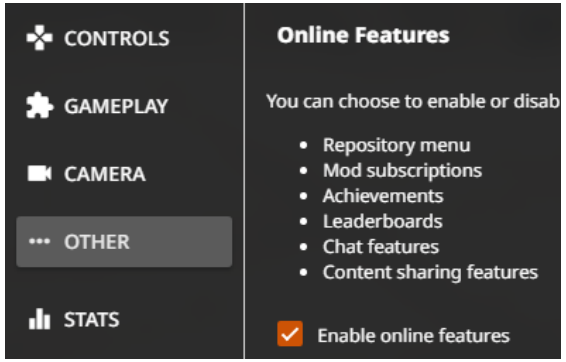
This tool is particularly useful for: physically moving a stuck vehicle, checking the **vehicle's overall weight**, testing the **tension limits of beams**, stressing the structure during tuning of beam **stiffness and damping**, verifying that the node-beam structure is properly **constrained**.

Another way to quickly move a vehicle is by **teleporting** it to the Free Camera position. **Press Shift + C** to activate Free Camera, move the camera to the desired location and press **F7** to teleport the active vehicle to the camera position.

But probably the most important command while using the simulator and testing models is the **R** key, for **resetting** the vehicle and **Ctrl + R** to **force reloading** after tweaking the vehicle's scripts.

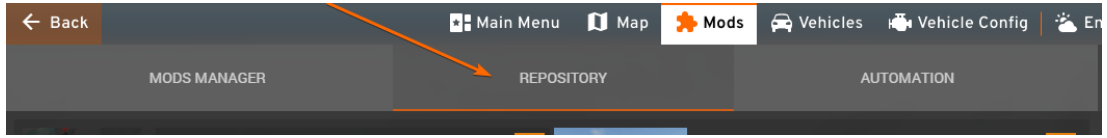
1. Essential Basics

How to Install and Manage Mods



You can find community mods for the game in the Official Repository and on the Official Forum.
Mods can be installed **automatically** through the in-game repository or via the BeamNG.drive repository website.
To enable the online repository, you may need to go to the settings by pressing **ESC** > **Options** (in the side menu) > **Other** (in the left-hand menu) > Scroll down and toggle "**Enable Online Features**".

Through In-game repository:
1. Open the Repository from the **Main Menu** or **Side Menu**.
2. Search for a mod and use filters to refine your search. Check ratings and dates to avoid outdated mods.
3. Click **Subscribe** to download and activate the mod automatically. Subscribed mods receive automatic updates.



To **manually** install them keep in mind that mods are in **ZIP format** and extracting them is **not required** for them to work.
1. Download the ZIP file
2. Open your **Userfolder**(do not extract it)
3. Create a **"mods" folder** (if it doesn't exist)
4. Move the ZIP file into this folder

1. Essential Basics

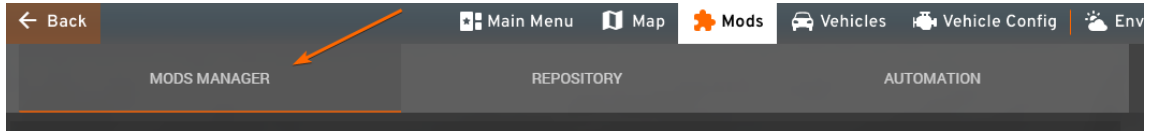
How to Install and Manage Mods

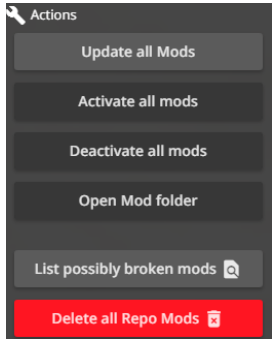
There are two "**mods**" folders available:

1. One inside the **BeamNG installation folder** (**Game Folder**):
   `<your_BeamNG_location>\BeamNG.tech.v0.34.2.0\<your_BeamNG_user_folder>\0.34\mods`
2. The other inside the **user directory** on your PC (**User Folder**):
   `C:\Users\<PC_user_name>\AppData\Local\BeamNG.drive\0.34\mods\repo`

This separation allows you to modify the game without affecting the original installation, making it easy to revert to the default state if needed.
The **User Folder** structure mirrors the **Game Folder** structure. If a file exists in the same relative location in both folders, the game will prioritize the one in the **User Folder** over the original.

Installed mods can be managed through the in-game **Mods Manager**. From here you can update, enable/disable mods, as well as manage individual mods from the list.





During vehicle **debugging** and **script customization**, to ensure that file modifications take effect immediately without restarting the simulator, go to: **Mods** > **Mods Manager** > **Deactivate all mods** and then **Activate all mods**.

1. Essential Basics

Creating Vehicle Configurations

Vehicle Configurations allow you to customize a vehicle. You can access the **Vehicle Config** menu from the **Side Menu.**

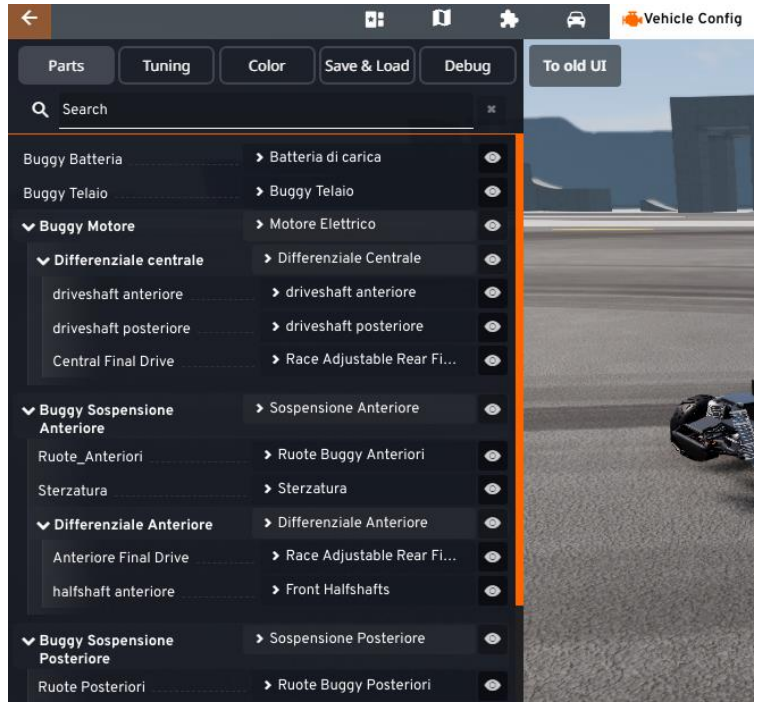
Parts: Allows selecting individual vehicle components, structured hierarchically as defined in the JBeam files. You can **show/hide**, **replace**, or **remove** parts. Some parts depend on others, meaning certain options may only be available based on previous selections. Always press **Apply** to confirm changes.

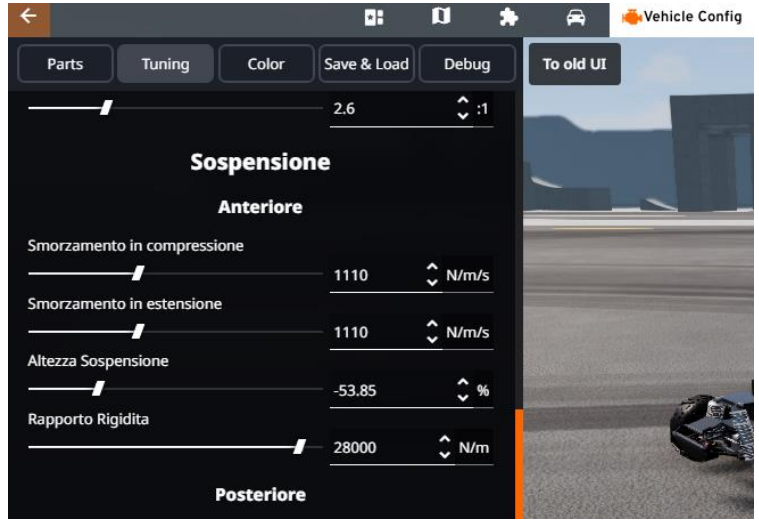
Tuning: Lets you adjust specific parameters of parts that have been pre-configured for modification via the `variables` function in the scripts. Adjustments are made using sliders or text input fields, with predefined minimum and maximum values.

Color: Modifies the color of vehicle parts that support customization.

Save/Load: Enables saving and loading custom configurations.

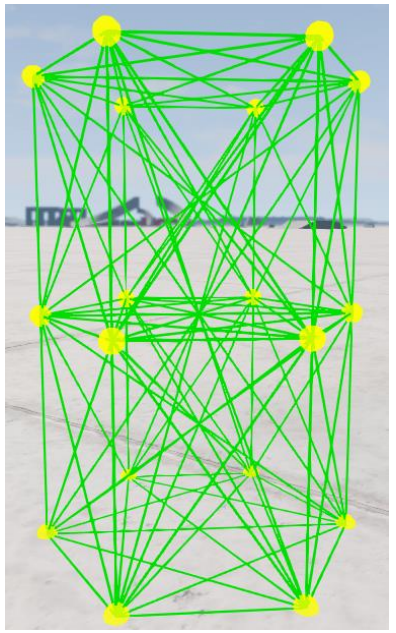
Debug: Provides tools for testing new models or scripts. This section is **highly recommended for debugging** and fine-tuning vehicle behavior.





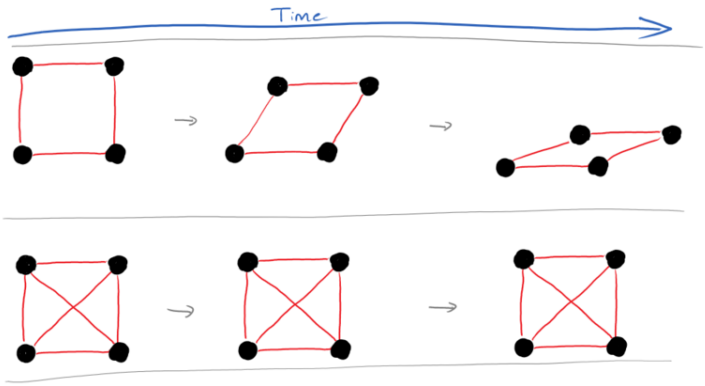
2. Modding – Introduction to JBeams

Unlike most games that use **RigidBody** physics, BeamNG utilizes a **SoftBody** physics system. This means that objects, such as vehicles, are fully **deformable**. This is achieved through a **Node and Beam structure**, which acts like a skeletal framework.



In this system:
- **Nodes** are points with mass that can move freely in space.
- **Beams** connect nodes (they always have a node at each end), maintaining a fixed distance between them. They function as springs, **have no mass**, and do not transmit twisting forces. Beams cannot bend but allow rotation around the nodes they connect, functioning similarly to **spherical joints**.

When creating a node-beam structure, if the structure is not properly **constrained**, it will tend to be unstable or collapse on itself.
With enough nodes and beams, it is possible to build complex structures like cars, with node-beam structures simulating the chassis, suspension, wheels, and many other components.

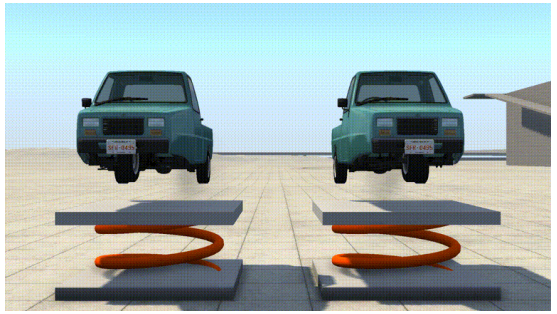


2. Modding – Introduction to JBeams

Basic properties of node-beam structures

beamSpring

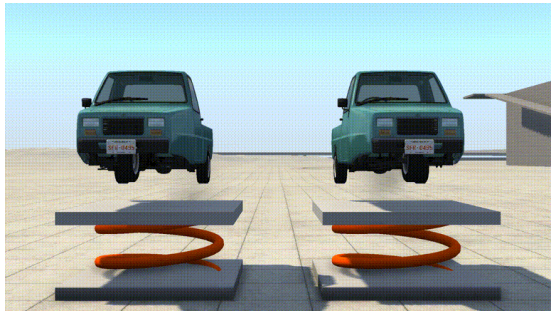
The `beamSpring` value determines a **spring's stiffness**, indicating the force needed to compress it by a set amount.
Higher values suit rigid structures like the chassis, while lower values are used for flexible materials like springs or rubber.
Stiffness shouldn't be confused with deformation: a low spring value allows compression but remains elastic, returning to its original length, while a low deformation value means the beam compresses permanently.



Soft Spring | Stiff Spring

beamDamp

Damping is the resistance to movement.
In a frictionless vacuum, a spring with no damping will tend to oscillate indefinitely. To avoid this, we need to have some damping, which will resist movement with a force inversely proportional to speed.
The main effect of damping will be to prevent oscillations, although damping will also have an effect in absorbing some energy from deformation, reducing the strength of the rebound.



No Damping | Some Damping

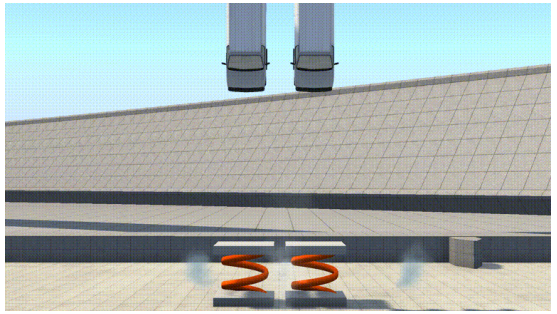
```
{"beamSpring":40000,"beamDamp":0}, //Example for suspension springs
{"beamSpring":0,"beamDamp":4500}, //Example for suspension dampers
{"beamSpring":8000000,"beamDamp":125}, //Structural vehicle components, such as suspension arms
{"beamSpring":14001000,"beamDamp":250}, //Steering rack, it needs to be super stiff to keep wheels pointing in the right direction
```

2. Modding – Introduction to JBeams

Basic properties of node-beam structures

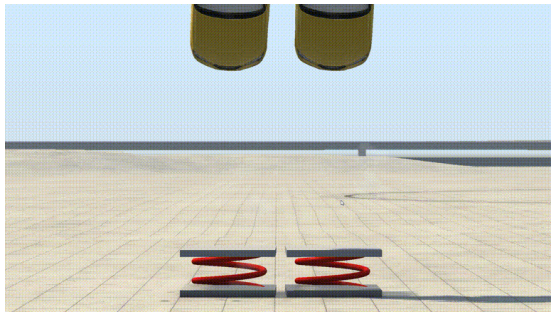
beamDeform

The `beamDeform` value sets the amount of force required before a beam **permanently deforms**. Once deformed, the beam will no longer return to its original shape. This is central to creating vehicles that deform accurately.
(In the image: Low deformation value (5000) | Practically infinite deformation value)



beamStrenght

The `beamStrenght` value sets the amount of force required to **break a beam**. A broken beam acts as if it has been snapped in half, this means it no longer connects two nodes together. This is useful for allowing components to fall off a vehicle. For example, a bumper can be made to fall off a car, by making the beams connecting it to the chassis break easily.
As shown in the following image, breaking beams will also result in the visible mesh being destroyed too.



Low Strenght | High Strenght

nodeWeight

The `nodeWeight` value can be used to adjust how **heavy** each individual point of vehicle is.

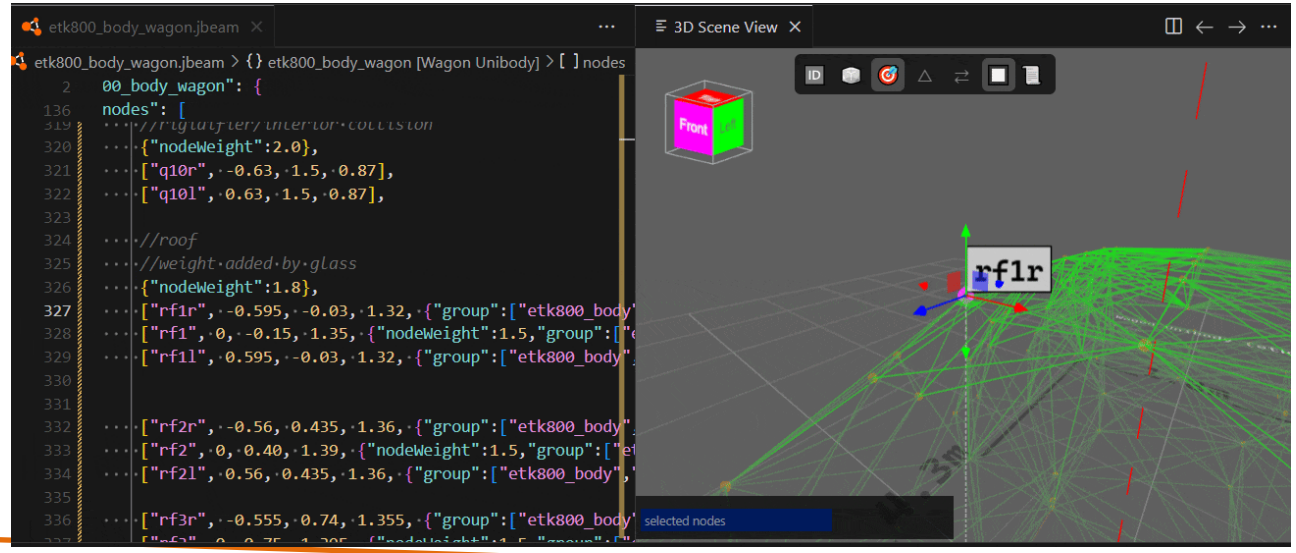
However, if the overall stiffness of all connected beams are too high, it will begin to vibrate and may even explode. To prevent this vibration, you either need to make the node heavier, or the beams less stiff. To make things even more complicated: beams will also start to vibrate and explode if your `beamDamp` is too high.

2. Modding – Introduction to JBeams

Before continuing, the provided **plugin** for viewing JBeam files enhances your editing experience by highlighting syntax errors and improving readability, which helps avoid common mistakes during development, tailored specifically for JBeam and JSON files. This makes it much easier to navigate and modify complex vehicle configurations.

It allows even a 3D visualization of the beam and node structure and the meshes.
It makes easier to move nodes and directly update their position inside the script.

You can download it through the extensions research directly using Visual Studio Code or you can download it in the following link.



2. Modding – JBeam Syntax

General Concepts

JBeam is the file format that defines the physics skeleton in the BeamNG engine. It is called JBeam as it is based on **JSON** (with some exceptions) in order to define node/beam constructs.
Just about everything is case sensitive, and it is not at all friendly to syntax errors, so be careful!

Comments: C-style, multi-line, and single-line comments are supported: `//...` and `/* ... */`
Commas: all commas are optional, but it is advised to only omit the commas at the end of lines.

Tables function like spreadsheets with a header row defining column names and rows containing data. This format saves space by avoiding repeated keys.

```
"vehicle": {
  "nodes": [ /* ... */ ],
  "refNodes":[
    ["ref:", "back:", "left:", "up:"],
    ["f3r", "f5r", "f4l", "f8r"],
    ["f2r", "f3r", "f1l", "f1r"]
  ]
}
```

In the example, `f3r` is the first value in its row in the column `ref:` of the table `refNodes` that is contained in `vehicle`.
Tables are used when you have a lot of the same data, so you save space specifying the key over and over again.

2. Modding – JBeam Syntax

General Concepts

Shown in the colon (:) in the header row of a table specifies a link (**section links**) to another section of the vehicle. The format is as follows: `valueName:targetSectionName`. If `targetSectionName` is omitted, it will be `nodes`.
In the example, with `"ref:nodes": "f3r"` the engine would reference to `id = f3r` in the nodes table.

```
"vehicle": {
  "nodes": [
    ["id", "posX", "posY", "posZ"],
    ["f3r", -0.35, -1.56, 0.25]
  ],
  "refNodes":[
    ["ref:", "back:", "left:", "up:"],
    ["f3r", "f5r", "f4l", "f8r"]
  ]
}
```

Dictionaries are `key-value` data storages. They are used when the data is quite unique and does not repeat itself so much. There is no post-processing required with dictionaries, but they are rarely used.

```
"vehicle": {
  "cameraExternal": {
    "distance": 6.7,
    "distanceMin": 9,
    "offset": {"x": 0.43, "y": 0.11, "z": 0.55},
    "fov": 77,
  },
}
```

A **scope modifier** applies a property to all subsequent rows at the same level. The modifier `{"nodeWeight": 3}` affects all following rows within the same scope. Unlike normal table rows (which use `[]`), scope modifiers use `{}` to define properties globally.

```
"vehicle": {
  "nodes": [
    ["id", "posX", "posY", "posZ"],
    ["f3r", -0.35, -1.56, 0.25],
    {"nodeWeight": 3},
    ["f2r", 0.37, -0.98, 0.22],
    ["f1r", -0.65, -0.64, 0.22]
  ]
}
```

2. Modding – JBeam Syntax

General Concepts

Negating the effects of scope modifiers is also possible by setting them to empty values. In the example the group modifier would only apply to node `f4l` and `f8r`.

```
[...],
["f3r", -0.35, -1.56, 0.25],
["f5r",  0.00, -1.58, 0.24],
{"group":"body"},
["f4l", -0.37, -0.98, 0.44],
["f8r", -0.37, -0.98, 0.22],
{"group":""},
["f2r",  0.37, -0.98, 0.22],
[...],
```

The example contains a **table row modifier** with a dictionary behind the columns. These values are applied for the row only and do not leak anywhere else. The node `f2r` would have the `nodeWeight` property, the node `f3r` would not.

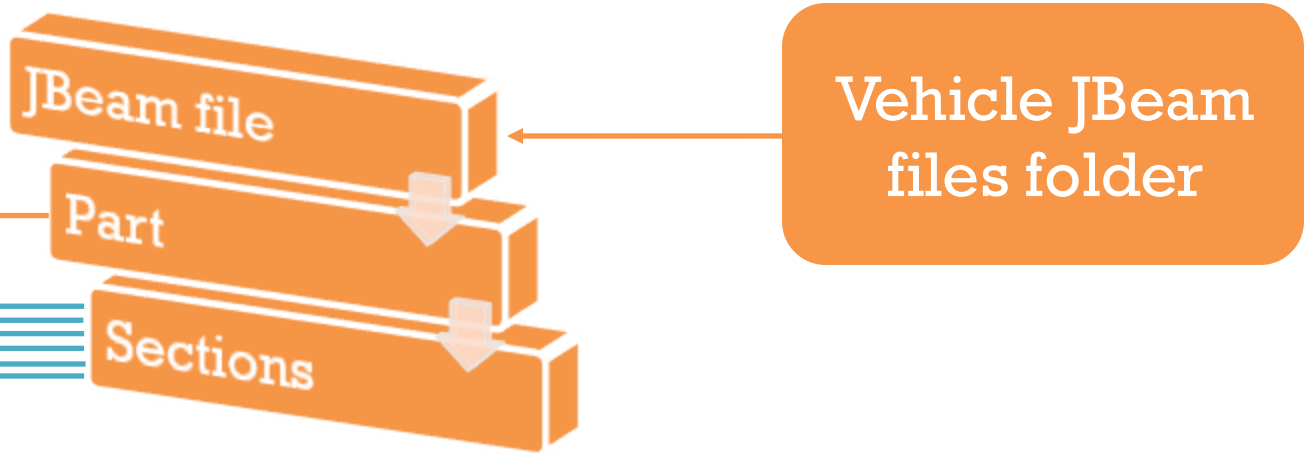
```
[...],
["f2r",  0.37, -0.98, 0.22, {"nodeWeight": 3}],
[...],
```

2. Modding – JBeam Syntax

Anatomy of a Vehicle JBeam

Vehicles in BeamNG are build together with **parts**, those parts contain then the so-called **sections** that contain the actual data.

```
{
    "Buggy":{

        "information":{ ···
        }
        "slotType" : "main", ···
        "slots": [ ···
        ],

        "controller": [ ···
        ],

        "flexbodies": [ ···
        ],

        "refNodes":[
            ["ref:", "back:", "left:", "up:"]
            ["fr4", "fr5", "fr4l", "fr4t"]

        ],

        "cameraChase":{
           "distance":2.0,
           "distanceMin":0.5,
           "defaultRotation":{"x":0,"y":-10,"z":0},
           "offset":{"x":0, "y":0.00, "z":0.35},
           "fov":65,
        },
```



Every jbeam file is a flat key-value dictionary of available parts this file contains. The **key** is the name of the **part**, its **value** are its **sections**. During loading, the engine constructs a hierarchical tree of parts and must identify the base or **root part**. This is determined by "slotType": "main", which designates the root part and allows it to be spawned directly.

2. Modding – JBeam Syntax

Advanced Concepts – Scaling process modifiers

There are some advanced out-of-bounds modifiers that can exist in the JBeam syntax. only used in very specific circumstances, like **scaling process modifiers**. They are declared next to the **Tables** and **Dictionaries** in the **main part** of Jbeam and they are used to scale the numeric values of other modifiers.

```
{
  "vehicle": {
    "slots": [
        ["type", "default", "description"],
        ["vehicle_body","vehicle_body", "Body"],
        ["paint_design","", "Paint Design"],
    ],
    "scaledragCoef":2.15,
    "controller": [
        ["fileName"],
        ["vehicleController", {}],
    ],
  }
}
```

The modifier works by searching for the string `scale` in keys inside the Jbeam structure next to the tables and dictionaries, and then multiplying all modifiers with the name specified in the rest of the string with the provided value.

For example, the one shown will multiply all `dragCoef` modifiers by `2.15`. It works globally on the whole vehicle, until a part that negates it is loaded.

It is most often used to fine-tune the overall **drag coefficient** of the vehicle, but it's possible to scale all other modifiers with it too. It should only be used in specific cases.

2. Modding – JBeam Syntax

Advanced Concepts – Disable modifiers

The **disable modifier** is a special modifier that will force rows to be skipped by **lua** and not read. For example the following lines will not be read, but the lines below will:

```
[
    {"disable": true},

    ["5","7","6"],
    ["5","6","4"],

    ["0","5","4"],

    ["2","6","7"],
    ["2","7","3"],
    {"disable": ""},
]
```

It is useful when combined with **Slot Variables** (more on that later) which pass a Boolean value. That allows you to merge multiple variants of a Jbeam part with small changes, which have separate `slotTypes`, into one part, with the sections that contain changes being surrounded with Disable modifiers which use Boolean slot variables.

```
[
  {"disable": "$variant1"},

  ["5","7","6"],
  ["2","7","3"],

  {"disable": "$variant2"},

  ["5b","7","6"],
  ["2b","7","3"],

  {"disable": ""},
]
```

The **Include modifiers** allows storing JBeam table data in an external CSV file, referenced with a path relative to the game folder.

2. Modding – JBeam Syntax

Advanced Concepts

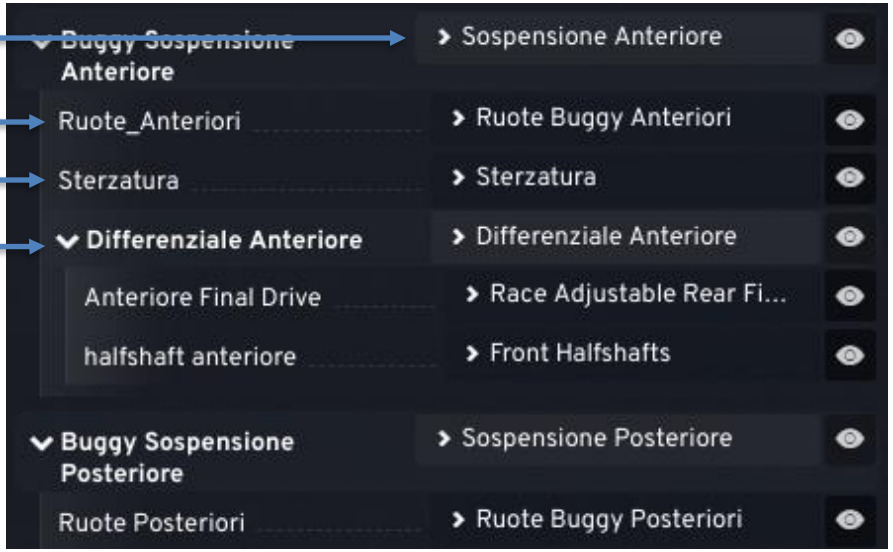
JBeam **functions** allow dynamic calculations and string concatenation by evaluating mathematical and logical expressions. They support standard **Lua** operators and built-in functions for rounding, clamping, and table operations. Numeric values in JBeam can be replaced with functions using `$=` inside double quotes. The math library is a modified version of Lua's standard library but doesn't require the `math.` prefix.

Since functions haven't been necessary for our work, we won't cover them in detail.

2. Modding – The part/slot system

BeamNG's car modification system relies on components and slots, organized in hierarchical levels, where available options depend on previously selected components.
Everything related to what you can see in the parts selector (**Side Menu** > **Vehicle Config** > **Parts**) is defined in the jbeam for each component.

```
{
  "Buggy_Sospensione_A": {                  //Internal name of the component
    "information": {
      "authors": "Stefano_Davide",
      "name": "Sospensione Anteriore"       //Name in the part selector
    },
    "slotType": "Buggy_Sospensione_A",      //Slot Type
    "slots": [                              //Slots
      ["type", "default", "description"],
      ["Buggy_Ruote_A", "Buggy_Ruote_A", "Ruote_Anteriori"],
      ["Buggy_Differenziale_A", "Buggy_Differenziale_A", "Differenziale Anteriore"],
      ["Buggy_Sterzatura", "Buggy_Sterzatura", "Sterzatura"]
    ],
 [...]
```



Each part has an **internal name**, defined by the very first line of a part's jbeam and will be used by the game to refer to that part, it is however not seen by the user. It is important for this name to be unique, to avoid issues like parts overriding each other.

2. Modding – The part/slot system

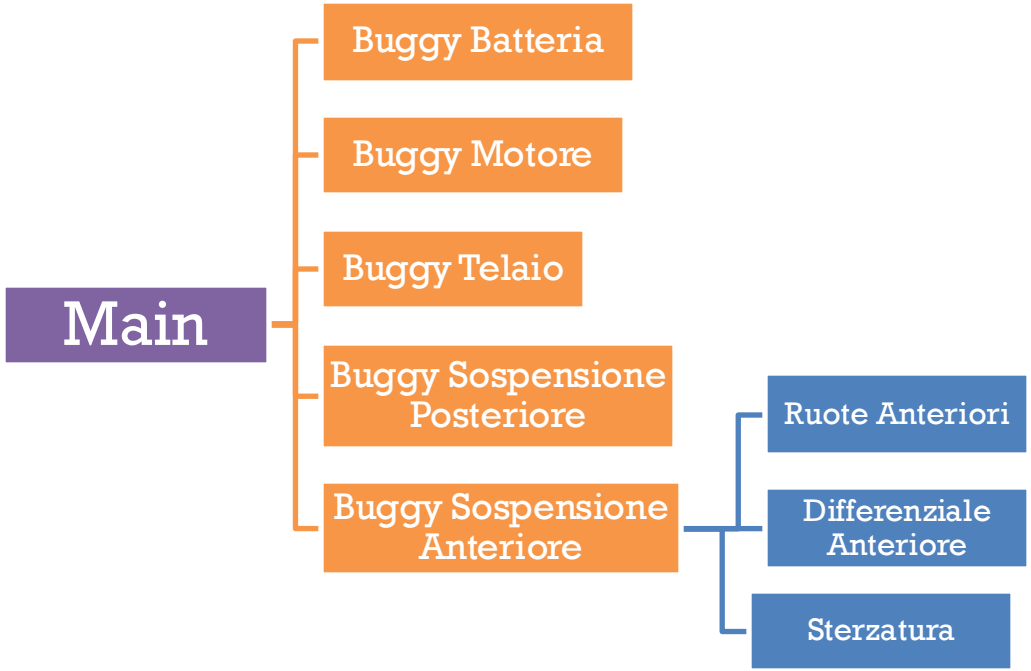
The information section shows some basic information on that component, including the name that will be shown in the parts selector.
Each part also has a `slotType`, which identifies where in the vehicle it fits (in the second part, an extensive example of that will be given).
The slot type allows you to have multiple parts with this specific slot type that would act as **alternatives**, with every part with an identical "slot type" appearing as different options of the same part in the part selector.

One unique slot type is `main`, which refers to the base component of the vehicle. This identifies the component as being the **root** part of the vehicle and will be the first component loaded by the game. **There can only be one part per car with that slot type.**

```
{"Buggy":
  {"information":{
        "authors":"Stefano_Davide",
        "name":"Buggy",
    }
    "slotType" : "main",
    "slots": [
        ["type", "default", "description"]
        ["Buggy_Telaio","Buggy_Telaio","Buggy Telaio"],
        ["Buggy_Motore","Buggy_Motore","Buggy Motore"],
        ["Buggy_Batteria","Buggy_Batteria","Buggy Batteria"],
        ["Buggy_Sospensione_A","Buggy_Sospensione_A","Buggy Sospensione Anteriore"],
        ["Buggy_Sospensione_P","Buggy_Sospensione_P","Buggy Sospensione Posteriore"],
    ],
  [...]
```

2. Modding – The part/slot system

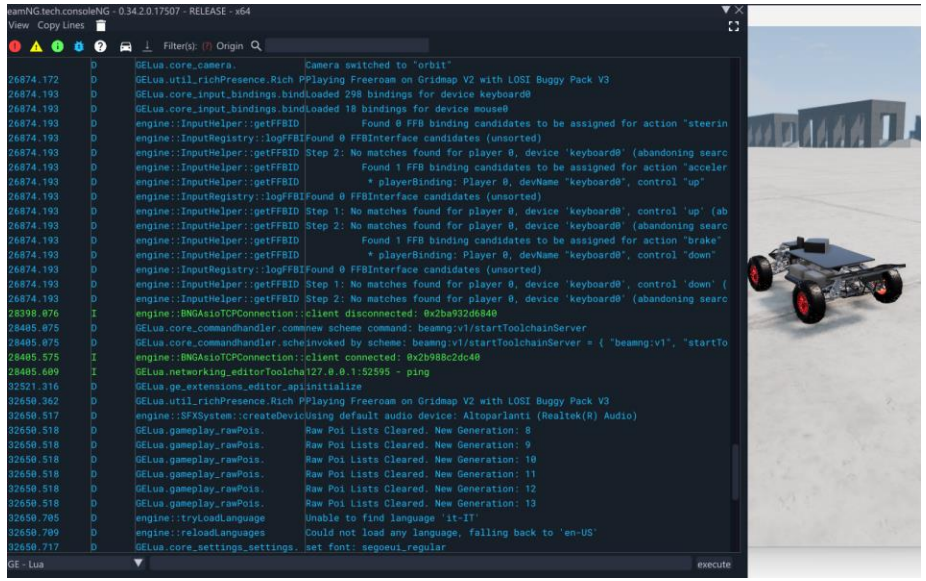
The slots section defines which components are the "children" of that component based on their `slotType`. If we follow with the example of our **front suspension**, you would find **front wheels**, **steering** and **front differential.**
The result is a parts selection tree, with the "main" part at the top, and multiple layers of "children" components.



2. Modding – Debug Tools

To help with Jbeam design, the game has multiple debug views that allow you to get more information about your Jbeam.

The **console** can be accessed using the ` (or ~) key, but for those without a US keyboard layout, it is recommended to change it in **Options** > **Controls** > **General Debug** > **Toggle System Console** and assign a more convenient key. It shows information and errors that happen while loading a car, and can be helpful to investigate why a vehicle isn't loading, has missing textures, etc.
Alternatively, the console can be accessed as previously mentioned (through **PowerShell** when launching the simulator).

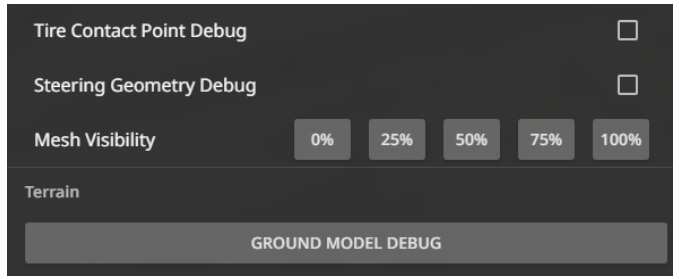


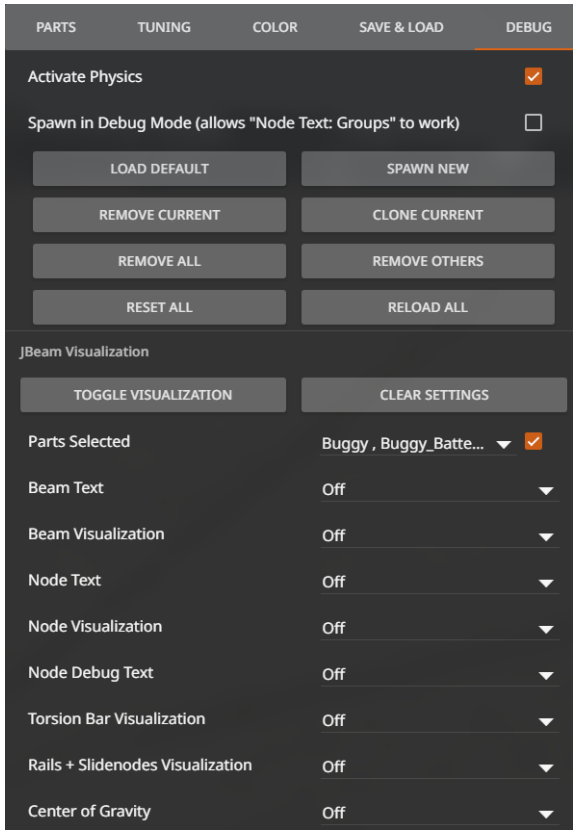
The information is **color coded**, with errors in red, warnings in yellow, info in green and debug information in blue. You can hide the various types of messages by clicking on the buttons in the upper left corner.
The text box at the bottom is used to enter commands.

2. Modding – Debug Tools

The **Debug UI** contains many ways to debug the vehicles' JBeam physics properties. This UI includes the debug modes listed and also includes vehicle spawning functions.

The **mesh visibility** option is useful for displaying nodes and beams while still keeping track of their position within the vehicle.





The Debug UI features a checklist-style part selector for debugging tools. Highlighted parts are affected, while others are not. A checkmark icon allows quick selection/deselection of all parts. Changes in the vehicle's part selector or visibility settings sync automatically. A search bar is also included.

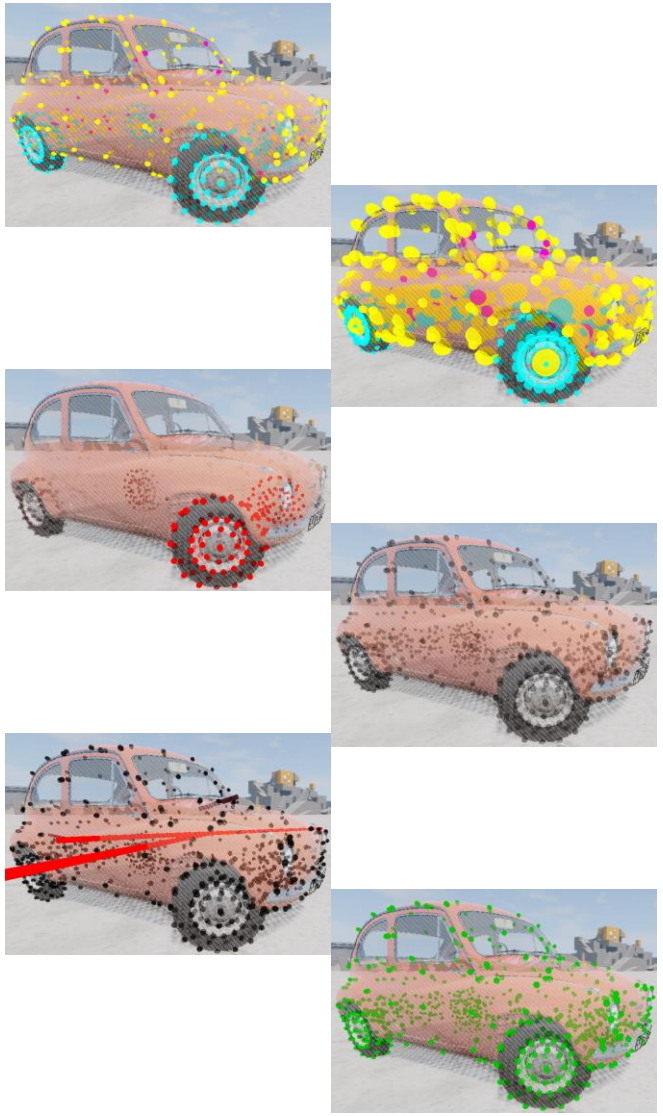
Node Visualization

The **node visualization** is a **fundamental tool**. The debug views can be accessed by default with **Ctrl** + **M**. In the debug UI, you can edit the width and transparency of the highlight.

2. Modding – Debug Tools

The following views are available:

1. **Simple** - Shows the nodes with a color code based on **collisions**. Yellow nodes have both **internal** and **external collision**, light blue nodes have **external collision** but **no internal collision**, purple nodes have **no collision** at all
2. **Weights** - Shows the nodes as dots whose size varies based on the weight of the node, color coded the same way as in the Simple mode.
3. **Displacement** - Highlights the nodes that have displaced from their initial position on spawn via elastic deformation. The bigger displacement, the more opacity the highlight color has.
4. **Velocities** - Shows node velocity compared to the reference nodes.
5. **Forces** - Shows a vector of the total forces applied on a node. Can be very useful to find instabilities or nodes which are being pushed by unwanted triangles collisions.
6. **Density** - Shows if the node is currently in the air (green) or in water (red).



2. Modding – Debug Tools

The **node text** views can be accessed by default with **Ctrl** + **N**. In the debug UI, you can toggle whether these text are also shown for wheel nodes (turned off by default). We can find:

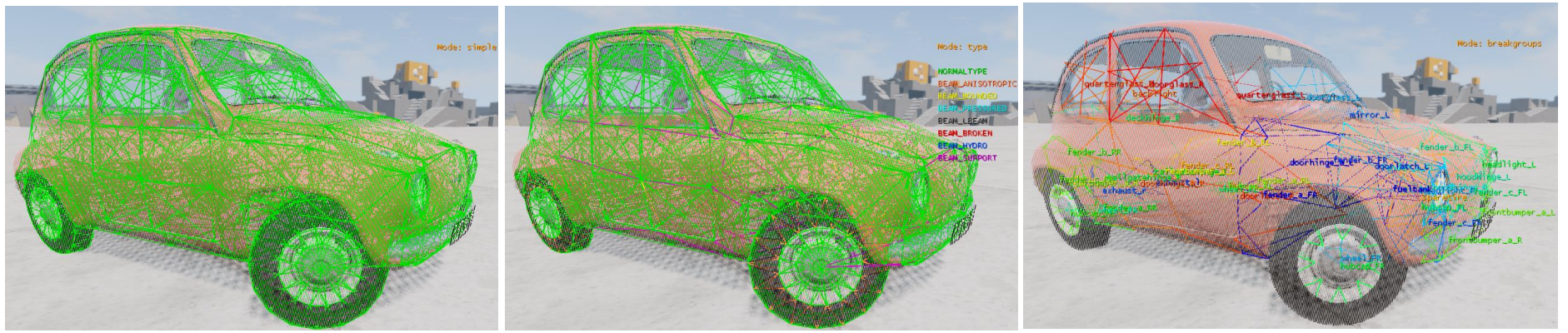
1. **Name** - Shows the name of each node as defined in the jbeam. If it has no name, which is true for wheel nodes, it shows the id number instead.
2. **Numbers** - Shows each node's id number. Mostly used by lua and the physics engine.
3. **Name+Numbers** - Combination of the two previous modes.
4. **Weights** - Shows each node's name and weight in kg. Also shows the total weight of all nodes around the top right corner of the screen.
5. **Materials** - Shows each node's name and assigned material.
6. **Groups** - For each node, shows its name and the group it belongs to. Requires turning on the "Spawn in Debug Mode" checkbox in the debug UI to work.
7. **Forces** - Shows each node's name and the value of the vector of total forces applied on it. Also shows average force around the top right corner of the screen. Couplers are shown with red font instead of black, and are not counted in the average calculation.
8. **Relative positions** - Shows each node's name and its position relative to the vehicle's local coordinates.
9. **World positions** - Shows each node's name and its position relative to the world.

2. Modding – Debug Tools

Node debug text shows extra debug text which contains info about some additional properties defined for some nodes. Can be accessed by default with **Ctrl** + **K**.

Beam Visualization

The **beams debug views** offer multiple views to help show the various states of beams and represent another **fundamental tool**. They are useful to help building your jbeam, and investigate issues caused by instability or leaking properties. The beam debug views can be accessed by default with **Ctrl** + **B**. Some beam debug views have a value range parameter to limit the values that are visualized. This can be set through **Vehicle Config** > **Debug** > "**Range Min**" and "**Range Max**" sliders and the "**Show Infinity Values (FLT_MAX)**" checkbox.



2. Modding – Debug Tools

This section contains beam visualization modes based on the **current status** of the beam in game. Available modes are:

1. **Simple** - Shows the beams in green (shown in the image).
2. **Type** - Draws the beams different colors depending on their type. **Regular** beams are green, **support** beams are purple, **bounded** beams are yellow, **hydro** beams are dark blue, **pressured** beams are cyan, **l-beams** are grey and **anisotropic** beams are orange. This color legend can be found in the game as well (shown in the image).
3. **With broken** - Same as the "Type" view, but shows broken beams in red.
4. **Broken only** - Only draws broken beams. Useful to investigate issues like beams breaking on spawn.
5. **Stress** - Shows how much stress is on the beams, coloring them red in compression, and blue in tension. Useful to see **instability issues** and **vibration.** There are two stress modes, one with and one without constraining range values in the debug UI, the latter named "Stress (Old)".

2. Modding – Debug Tools

6. **Displacement** - Shows how much the beams are displaced via elastic deformation from their initial lengths. Blue means the beam has been stretched and red means that it has been shortened. Debug UI lets you constraint range values.
7. **Deformation** - Shows how much the beams are permanently deformed. Blue means the beam has been stretched past its initial length, red that it has been shortened. Debug UI lets you constraint range values.
8. **Break Groups** - Highlights the beams by `breakGroups` with different colors. Useful to investigate issues of `breakGroups` that aren't working as they should, or leaking (shown in the image).
9. **Deform Groups** - Highlights the beams by `deformGroups` with different colors.

2. Modding – Debug Tools

10. **Bounded Beam Bounds** - Shows only **bounded** beams.
    - If the beam has not expanded or compressed from the spawn length (taking precompression into account), it is shown fully in green with no overshoot length from the nodes.
    - If it has been shortened from the spawn length, then the current length is shown in green, and it is extended to the spawn length on both sides by dark blue color. If the beam's `shortBound` has been triggered, it is shown in cyan color on top of the blue, and additionally light blue visualizes the transition zone.
    - If the beam has been extended from the spawn length, then the green color shows the original length, while the extension is filled in by red color. If the beam has expanded beyond the `longBound` limit, the remaining extension is filled in with yellow color, and additionally orange visualizes the transition zone.
11. **Support Beam Bounds** - Does the same as above, but for **support** beams. Transition zones are not shown due to being zero length. These beams frequently utilize precompression lower than `1` combined with high `longBound` values, so they will often have long reaching yellow `longBound` limits.
12. **Frequency** - Highlights beams which have the set frequency with the set max amplitude.

2. Modding – Debug Tools

There also beam visualization modes which visualize beams based on their **properties set in Jbeam**. These debug modes visualize lower end range values in white, upper end range values in red, and infinity (FLT_MAX) values in purple. The modes for these are:

13. **Beam Damp**
14. **Beam Damp Fast**
15. **Beam Damp Rebound**
16. **Beam Damp Rebound Fast**
17. **Beam Damp Velocity Split**
18. **Beam Deform**
19. **Beam Limit Damp**
20. **Beam Limit Damp Rebound**
21. **Beam Long Bound**
22. **Beam Precompression**
23. **Beam Precompression Range**
24. **Beam Precompression Time**
25. **Beam Short Bound**
26. **Beam Spring**
27. **Beam Strength**
28. **Bound Zone**
29. **Damp Cutoff Hz**
30. **Damp Expansion**
31. **Deform Limit**
32. **Deform Limit Expansion**
33. **Deformation Trigger Ratio**
34. **Long Bound Range**
35. **Precompression Range**
36. **Short Bound Range**
37. **Spring Expansion**

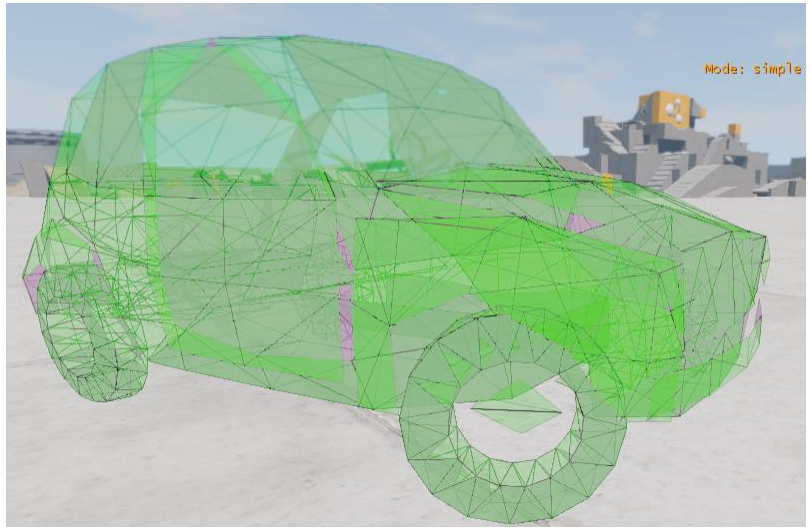
2. Modding – Debug Tools

Neither **slidenodes** nor **torsion bars** were used, however, these tools can be helpful:

- **Torsion Bar Debugging**: Available in the debug UI, it visualizes torsion bars, useful for debugging rigid structures like subframes and suspensions. Different modes highlight intact, broken, stressed, or deformed bars using colors and opacity.
- **Rails & Slidenodes Debugging**: Also in the debug UI, it helps identify misaligned slidenodes causing beam breakage or deformation on spawn. Modes allow distinguishing between intact, broken, and detached elements.

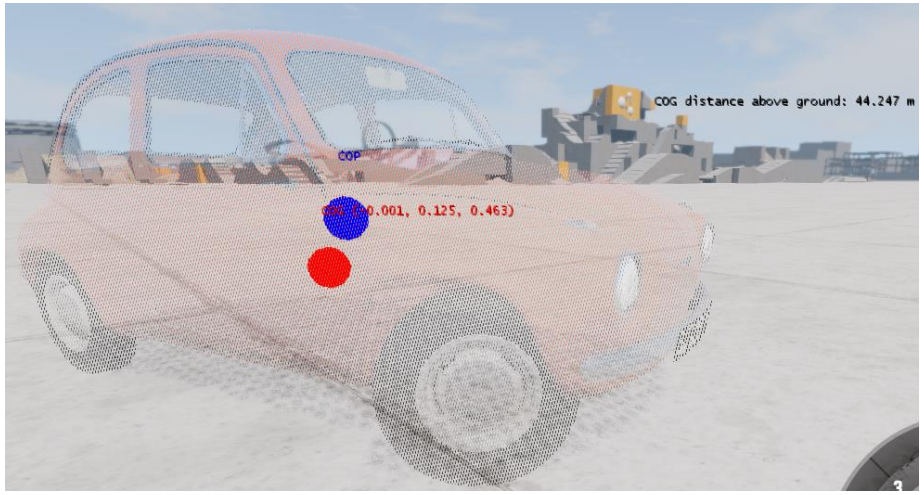
Collision Triangle Debugging (**Ctrl** + **T**) helps analyze collision and aerodynamics. The Simple mode highlights the front side of triangles in green and the rear in purple.

Other modes differentiate non-collidable, pressured, or broken triangles with distinct colors.

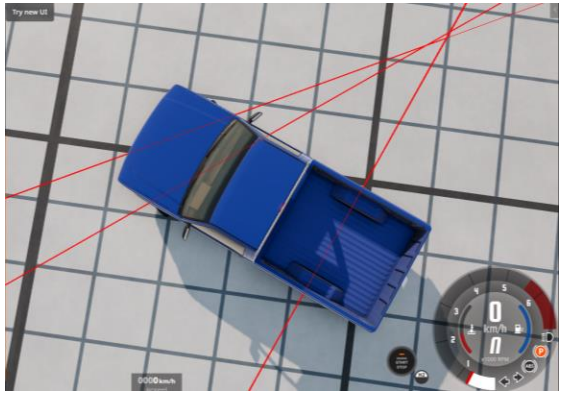


2. Modding – Debug Tools

There are specific debug functions for **aerodynamics**, accessible via the debug menu. The **Aerodynamics Debug** visualizes key forces: **drag** (red vector), **lift/downforce** (blue vector), and the **angle of attack**. The **Center of Gravity** (**Ctrl** + **G**) **tool** shows the CoG (red dot) and the Center of Pressure (CoP) (blue dot).



The **tire contact point debug** shows ground contact point for each tire of the vehicle.



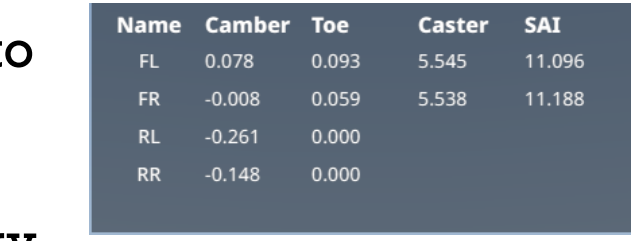
The **steering geometry debug** casts rays from wheel centers in both directions. The intersection points between them help visualize and debug the steering geometry. The rays have adjustable length.

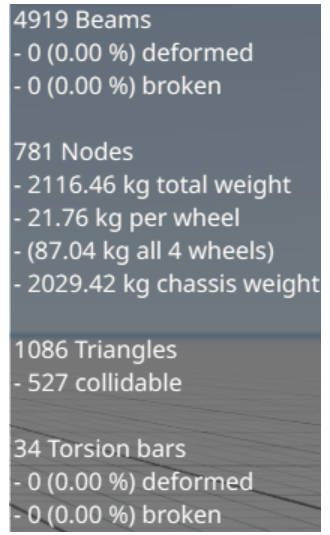
While not a debug tool by itself, the **mesh visibility** scale is very useful to help you see the various debug views better by making the mesh partly or full transparent. The default keys are **Ctrl** + **NumpadPlus** and **Ctrl** + **NumpadMinus** to raise and lower mesh visibility, respectively.

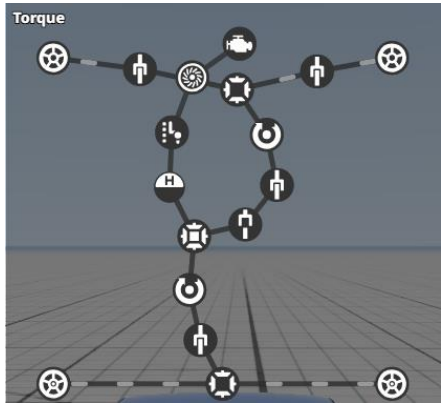
2. Modding – Debug Tools

BeamNG includes a number of useful **debug UI apps** that can be added to the interface using the UI Apps tab in the top side menu.
Here's some of them:

- **Advanced Wheels Debug** shows the **toe** and **camber angles** for every wheel on the vehicle, and additionally caster and SAI angles for steer axis wheels. Can be used to help set up and debug suspension geometry and also makes `refNodes` alignment issues easy to notice.
- **Node/Beam Info** reliably displays Jbeam related statistics of the vehicle, such as a number and percentage of deformed or broken beams and torsionbars, **precise total weight**, number of triangles, etc.
- **Powertrain Visualization** visualizes the **powertrain tree** of the vehicle with all of its components and the torque going through it. This can be changed to power by clicking on the 'torque' text. Highlighting a component with the mouse will display its name and RPM. Some components such as locking wheel axles, disconnectable shafts, locking differentials and rangeboxes can be interacted with by clicking on them in the app.

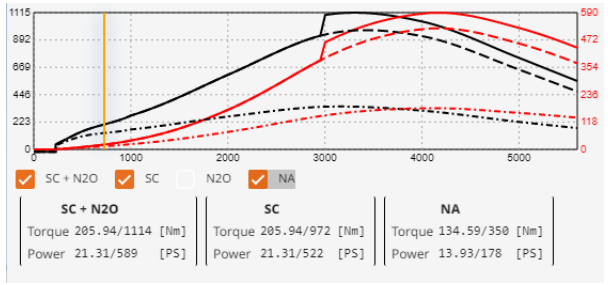






2. Modding – Debug Tools

- **Torque Curve** is practically a must have for **tuning engines**, this app shows the torque to RPM curve in Nm and power to RPM curve in Ps, displaying the current and peak power and torque values together with the current point on the curve.
- **Weight Distribution** shows the load on each wheel of the vehicle in different ways at once: as a background visual highlighting wheels with over 25% load, as weight load in kg, force load in N, and percentage of total weight on each wheel.
- **World Editor** can be accessed with the **F11** key by default, contains a few useful tools for Jbeam debugging. They can be added via the Window tab.
- **Vehicle Editor** is a group of useful Jbeam tools which can be accessed via **Shift** + **F11** or from the World Editor tabs: Window > Experimental> Vehicle Editor. It has two tabs: **Static Editor**, focused on editing Jbeam files, and **Live Editor**, focused on debugging Jbeam behavior in real time. Both have customizable layouts that can be saved,



2. Modding – Common JBeam Issues

Your first reflex with any jbeam issue is to check **console** for error messages.

Among the most common errors are **syntax-related issues**, such as missing quotation marks, missing brackets, and duplicate component names.

Flexbody issues can cause a part to load with its JBeam structure but without the flexbody appearing:

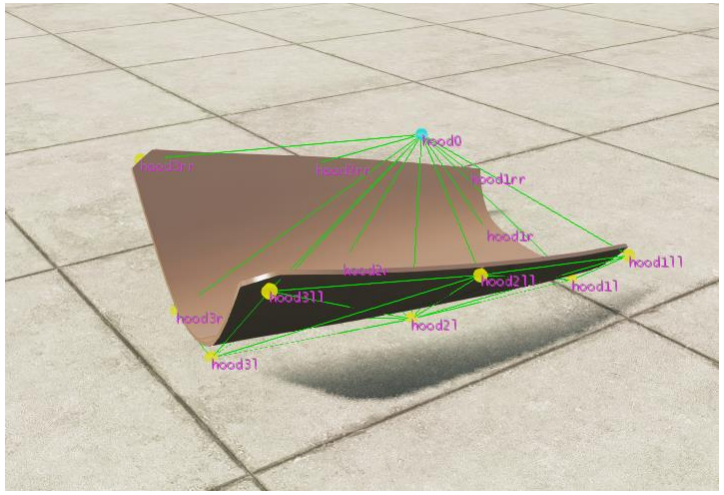
- **Offset Flexbody**: If the model appears in the wrong position despite being correct in Blender, the part's origin is likely misaligned.
- **Missing Flexbody**: If the flexbody doesn't appear, the mesh may be missing from the mod's folder or incorrectly named in the flexbody section.
- **Missing Nodegroup**: If there are no console errors but the flexbody is invisible, the referenced nodegroup may be missing or misnamed.
- **VY Node Error**: The physics engine needs at least three nodes to map the flexbody properly. Issues arise if there are too few nodes, if they are misaligned, or if they don't adequately cover the flexbody.
- **No Material Assigned**: If no material is assigned in Blender, the model won't render in-game.

2. Modding – Common JBeam Issues

Most of these issues can be resolved by checking model origins, ensuring correct naming, and assigning sufficient nodes.

Structural issues don't usually generate console errors but can cause visible problems like vibration or excessive floppiness:

- **Weak or Deforming Parts**: Check beam parameters, as incorrect deform values, stiffness, or damping settings can lead to instability.
- **Collapse on Spawn**: Often caused by a lack of bracing. Flat components may need a **rigidifier node**, which should be positioned correctly to provide stability.
- **Instability**: Excessive `beamSpring` or `beamDamp` values relative to node weight can cause extreme vibrations or even make the vehicle disappear on spawn. This can be diagnosed using the Stress debug view and resolved by adjusting beam stiffness/damping or increasing node weight. Nodes being pushed on spawn can also contribute to instability.



If you already have a rigidifier node, you might need to move it further away from the surface.

2. Modding – Common JBeam Issues

Major deformation on spawn have a few possible causes, to confirm these you can look at the stress debug just as the car spawn, using 100x slow motion. You should see vibration happening just as the car spawn.
It might also be caused by excessive or excessively quick beam precompression. Make sure that you didn't make an error and are precompressing the beam too much.

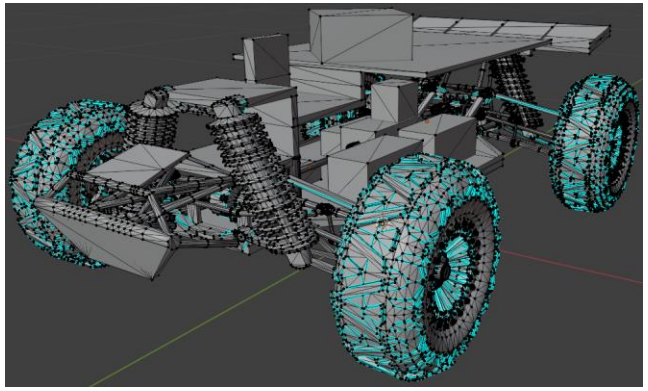
Other **common console issues**:

- **Duplicate beams** can happen with typos and errors while copy pasting. Fixing those simply requires removing one of the superfluous references to said beam.
- Another error you might see which won't necessarily affect the functionality of your mod is a **missing node error**. This can either be the result of a typo, or a beam/triangle that you forgot to remove after removing a node.
- A **missing flexbody error** might happen if you based your jbeam on pre-existing jbeam or did some changes in your flexbodies. If there are no visibly missing meshes on your vehicle, simply remove the reference to that flexbody in your jbeam.

2. Modding – Vehicle Modeling Guidelines

BeamNG utilizes the **Collada** (**.dae**) file format for its models. While there are some differences, such as how parts are separated and the inclusion of additional details—especially for mechanical components—creating models for BeamNG is quite similar to modeling vehicles for other video games.
To help you avoid common issues, here are some **essential guidelines** to follow. If you're new to 3D modeling, you may find Blender Guru's tutorial series on YouTube a great starting point.
Blender Beginner Tutorial or Beginner Blender Anvil Tutorial

It's important to find a balance between **polygon density**, **visual quality**, and **performance**. BeamNG continuously recalculates vertex positions based on chassis deformation, making high-poly models more computationally demanding. To optimize performance, it's best to **minimize unnecessary polygons** and rely on **normal maps** (special textures that simulate surface details) for details like panel gap bevels, underbody cutouts, structural components, and bolts. Using **vanilla models** as a reference can be extremely helpful in maintaining an efficient polycount.



2. Modding – Vehicle Modeling Guidelines

Each component must have an assigned **material**. We recommend using existing materials or custom ones with a unique prefix, such as your mod's name, to avoid conflicts with BeamNG's common materials.

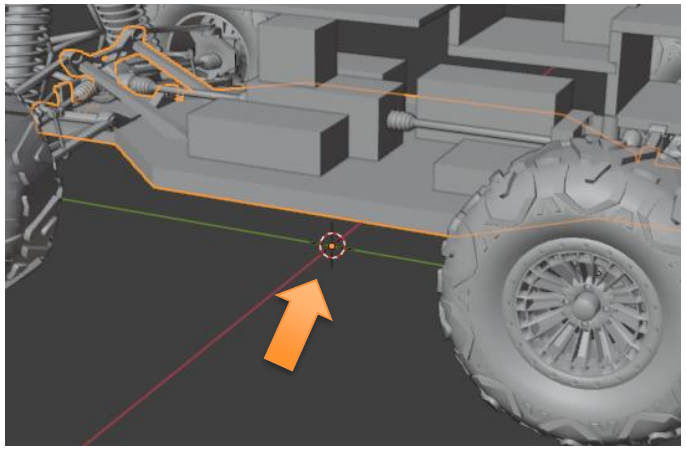
For modeling, parts that can detach or move independently should be **modeled separately** and have their own mesh, but there's no hard rule. Components using `deformGroups`, like lights and windows, should also be split into multiple parts, each with its own material.

Similarly, **part names** should include the mod's name or a unique prefix to prevent conflicts with existing game files.

The game will be able to load meshes from **multiple dae files** as long as they are located inside the vehicle's folder, or the common folder.

2. Modding – Vehicle Modeling Guidelines

Part Origins

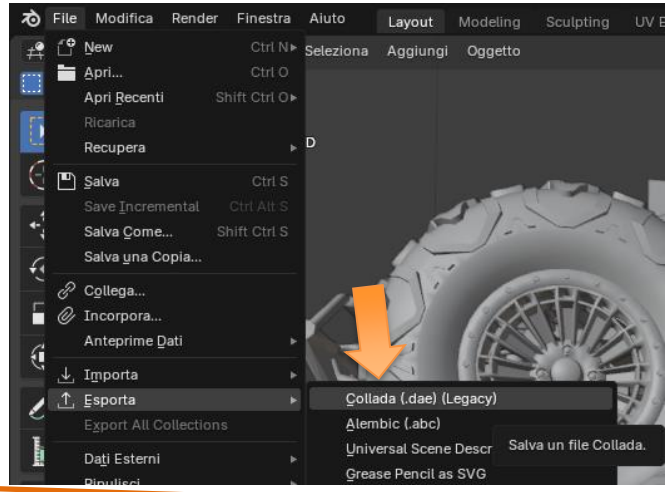


Part origins are a common issue. You can see a part's origin in Blender by clicking on the component and looking for the orange dot (as shown).
For **flexbodies** , which are components whose position and shape will be affected by the jbeam (chassis, suspension components, etc), the position of the part's origin must be at **0,0,0**. If the origin of one of your component's origin is offset, the part will show up offset in BeamNG.

If you have this issue, within Blender press **Ctrl** + **A** and choose "**All transforms**".

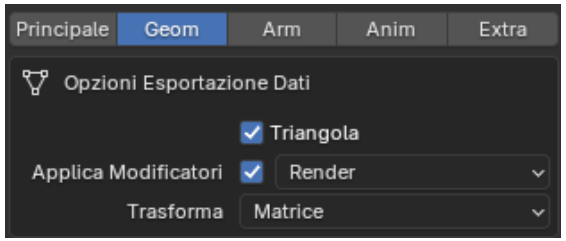
Exporting Models

To export your model, start by making sure you are in "**Object mode**" and aren't editing the geometry of any object. Then go into **File** > **Export** > **Collada**



2. Modding – Vehicle Modeling Guidelines

Exporting Models



In the export window, make sure to go under "**Geom**" and tick the "**Apply Modifiers**" box and set it to "**Render**". This will make sure that any modifiers like mirrors and edge split gets applied to the exported model. Then press "**Export Collada**" to export the model into your mod's folder.

If you have BeamNG open with your car loaded in another window, this will cause the vehicle's model to get updated immediately, which can be useful to see how it looks in game after small changes.

Importing models from the game

A first step in a lot of modding projects is to import parts from **vanilla models**. Go to "`<your_BeamNG_location>\BeamNG.tech.v0.34.2.0\content\vehicles`". You should see the zip files of all the vehicles and props in the game. It includes a folder for each car/prop, along with a "**common**" folder which contains all assets that are shared between vehicles (this includes wheels, tires, components shared between the D/H-Series, etc).

2. Modding – Vehicle Modeling Guidelines

Importing models from the game

Take note however that all those names are the "internal" names of vehicles. Some are fairly obvious, like "etk800" is the ETK 800 series. However, especially older cars, the internal name is referring to the car's body style.
Open the folder until you see **.dae** file you need. For cars it should be in the same folder as the jbeam files.

The first step is to copy it somewhere outside of the zip file, like on your desktop, so Blender can access it. In Blender then select **File** > **Import** > **Collada** and select the file you just chose.
The model should then appear in Blender.

Do note that the model usually includes **every single option component available** for the car; this means that there can be multiple versions of bumpers and other components included. The model files for vehicles won't include wheels, tires, and shared components like the engine and suspension on the pickup/van. If you need to import those components, their model files are located in the "**common**" folder.

2. Modding – JBeam File Sections

Jbeam files are divided into multiple **sections** which each represent one type of element in a vehicle's structure and we will further explore the ones that were useful in the creation of our vehicle model.

Basic structural elements

These are the **main elements** of any jbeam and handle the **shape** and **collisions** of a vehicle.

- **Nodes**: The first element in a node/beam structure. They are point masses and handle most of a jbeam's collisions along with the weight of the structure.
- **Beams**: Connectors between nodes. Beams allow for flexibility and deformation in a jbeam structure and define how the different elements are connected to each other.
- **Triangles**: Surfaces defined between nodes. Triangles are used for aero calculations, and for collisions between vehicles.

2. Modding – JBeam File Sections

Advanced structural elements

These elements enable advanced structures and are useful for specific applications:

- **Hydros**: Beams whose length can be changed on command.
- **Rails** and **slidenodes**: Allow the definition of nodes that slide along a rail. Used for things like steering racks, rigidifying some structures, etc
- **Thrusters**: Allows to propel structures directly. Used for testing purposes, jato and others.
- **Torsion bars**: Creates a torsion resistance between two lever arms. Used for sway bars and to rigidify some structures.
- **Torsion hydros**: Similar to hydros but instead changes the angle between two lever arms. Used for some steering systems, and other types of actuators.

Part/slot system

These elements are mostly used within the part/slot system, either to define some additional information related to the part, or the slots within a part:

- **Information**: Defines a few basic parameters associated to the component, like it's display name in the part selector.
- **Slots**: Defines the list of slots within a component, along with the default selection.

2. Modding – JBeam File Sections

Graphical and sound elements

- **Flexbodies**: Defines the models that will be used to graphically represent your vehicle, and how they are tied to the nodes for deformation.
- **Glowmaps**: Allows the definition of materials that can switch on demand. Mostly used to switch the material of lights to an emmissive material when the lights are turned on.
- **Soundscape**: Defines sound effects to be played when certain conditions are met, along with the node used as an emitter.
- **Cameras**: Defines the position and angle of the various cameras in the vehicle, including exterior and interior cameras.
- **Props**: Animated 3D elements that do not deform with the jbeam structure. Used for things like gauges, pedals, steering wheels, etc.

2. Modding – JBeam File Sections

Powertrain

The last section is the elements related to powertrain components, like powertrain definitions and wheels:

- **Powertrain**: Used to define the different powertrain elements within a vehicle's powertrain.
- **Pressure Wheels**: Allows to define wheels with a pressured tire model.
- **Rotators**: Alternative to pressure wheels in specialized applications, where power needs to be sent to something other than a wheel like a propeller.

Others

- **Variables**: Allows the definition of adjustable parameters for a vehicle, usually adjustable suspension and drivetrain components.
- **Triggers**: Defines set triggers that the player can activate with the mouse to control different functions of a vehicle.
- **Refnodes**: Set of nodes which define the central coordinate system of a vehicle.
- **Controller**: Used to add special functions to a vehicle, like police lightbars, electronics assists, etc.

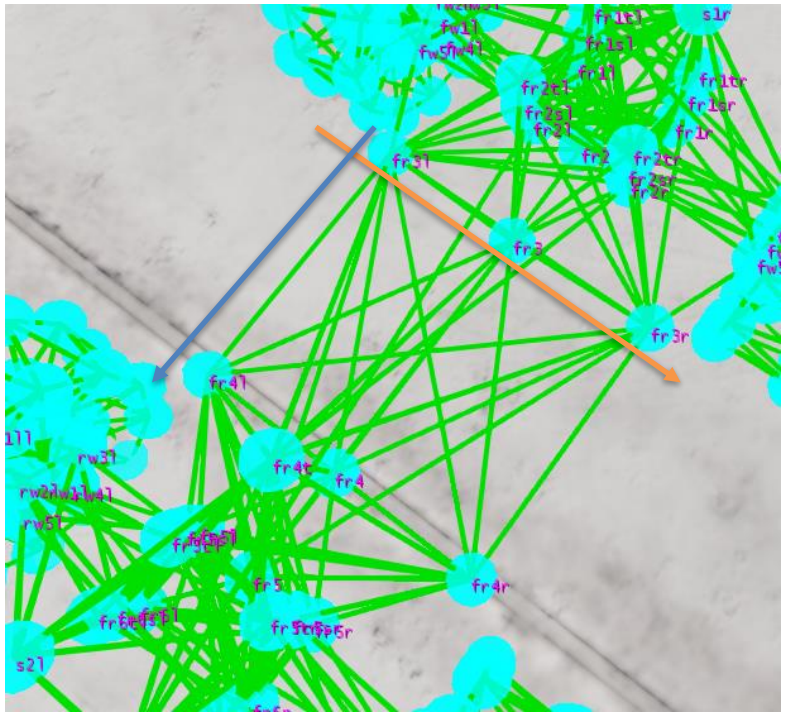
2. Modding – JBeam File Sections – Nodes

back

Nodes are mass points and the core of the BeamNG physics, they are **dimensionless** (infinitely small) **mass point**, defined by a **unique name** and a **position** in 3D space.

Nodes can have any string for their name. A good naming scheme is a sequence of relevant letter, number and letter(s) again, signifying which side of the vehicle it's on. A node sitting in the middle (posX = 0) would have no letter suffix.

Nodes also have many properties which can be set by placing dictionary lines above the nodes which are desired to be affected. Node weights are in **kilograms**, and coordinates are in **meters**. The "group" modifier is for assigning flexbodies. Node material only affects sound and particle properties, not physics.
The game engine uses a z-up coordinate system and SI units for most parameters.



This is the node and beam structure of our vehicle (body). The progression of node names is shown in orange from left to right and in blue from front to rear.

2. Modding – JBeam File Sections – Nodes

back

Collision system

In BeamNG, collision is handled by nodes alone, beams can not collide with anything. There are **three types of node collision**:

1. **Heightmap** - with the ground
2. **Static** - with collision mesh faces of map objects
3. **Dynamic** - with vehicle triangles, additionally this can be divided into:
   - **Self-collision** (triangles of the same vehicle node belongs to)
   - **External collision** (triangles of other vehicles)

Heightmap collision differs from static collision by having several layers which helps vehicle tires rotate smoother. Collision types have priority, with **dynamic being prioritized over static and heightmap** collisions. Practically this means that with high enough force a node can be slightly pushed through a solid wall or the ground. It will pop back out once the (surface) triangle stops pushing it.

2. Modding – JBeam File Sections – Nodes

back

Collision system

In Jbeam, these types of collision are controlled by three properties:

- The `collision` argument affects all types of collision and has the **highest priority** enabling any other collision type will not work if this one is disabled.
- The `selfCollision` argument only affects the self-collision type of dynamic collision.
- The `staticCollision` argument affects static and heightmap collision.

In BeamNG, when a node approaches a surface triangle, collision forces attempt to stop it, while anti-clip mechanics determine on which side of the triangle the node will end up. Collision forces start applying slightly before contact to prevent clipping, and various properties allow fine-tuning of this behavior. Additionally, node groups influence collision handling by preventing nodes from colliding with triangles formed by nodes in the same group. This system helps with optimization and structural design.

2. Modding – JBeam File Sections – Nodes

back

Required arguments

id name	**dictionary** type	Defines the node name. **Need to be unique** for the whole vehicle
posX name	**number** type	The X (left/right) position in 3D space (m)
Left is positive, right is negative		
posY name	**number** type	The Y (forward/back) position in 3D space (m)
Backward is positive, forward is negative		
posZ name	**number** type	The Z (up/down) position in 3D space (m)
Up is positive, down is negative		

Optional arguments

Will be divided them into categories and sorted from most to least frequently used.

2. Modding – JBeam File Sections – Nodes

back

Optional arguments - General

nodeWeight name	**number** type	option.nodeWeight default	The weight of the node in kg
As of game version 0.35.0.0 the default weight of a node is 25 kg			
group name	**string** type	Groups a set of nodes into a group that can be used later in other sections	
This is mostly used to map visible model parts to groups of nodes for deformation. Nodes can also be assigned to multiple groups on an individual basis, as per the following example: `["f6ll", 0.76, -0.72, 0.83, {"group":["coupe_engine","coupe_windshield"]}],`			
nodeMaterial name	**string** type	options.nodeMaterial default	Physics material of the node, rubber, metal, etc.
The node physics material affects the default sound events generated by the node, as well as the type of emitted particles. The name should be "**NM_**" + name of the physics material, for example "**NM_METAL**". As of version 0.35.0.0 the following physics materials are available: METAL, PLASTIC, RUBBER, GLASS, WOOD, FOLIAGE, CLOTH, WATER, ASPHALT, ASPHALT_WET, SLIPPERY, ROCK, DIRT_DUSTY, DIRT, SAND, SANDY_ROAD, MUD, GRAVEL, GRASS, ICE, SNOW, FIRESMALL, FIREMEDIUM, FIRELARGE, SMOKESMALLBLACK, SMOKEMEDIUMBLACK, STEAM, RUMBLE_STRIP, COBBLESTONE, FOLIAGE_THIN, SPIKE_STRIP			

2. Modding – JBeam File Sections – Nodes

back

Optional arguments - Collision

These define what the node will collide with. Also used in every Jbeam file. Usually, `collision` and `selfCollision` are used together, while `staticCollision` is rarely used.

`collision` `name`	**`boolean`** `type`	`true` `default`	If the node can collide with anything
This argument affects all types of collision and has the **highest priority** - enabling any other collision type will not work if this one is disabled.			
`selfCollision` `name`	**`boolean`** `type`	`false` `default`	If the node can collide with the vehicle it belongs to
This argument only affects the self-collision type of dynamic collision. It is commonly set to false for nodes that would cause issues if they collided with the rest of the vehicle, but still need to collide with everything else, for example wheels and tires.			
`staticCollision` `name`	**`boolean`** `type`	`true` `default`	If the node can collide with map objects and terrain
This argument affects static and heightmap collision. It is set to false for wheel center nodes, which can't collide with the world as that would make the collision offset from the rim edges but still need to collide with other vehicles for correct crash deformation.			

2. Modding – JBeam File Sections – Nodes

back

Optional arguments – Friction

Define the node's **static** and **sliding friction**. `frictionCoef` is used in every Jbeam file and most of the time set to 0.5, `slidingFrictionCoef` much less, usually just in tires, as its default value is the same as the value set in `frictionCoef`.

`frictionCoef` name	**number** type	1 default	Static friction of the node
By default, this affects both static and dynamic friction, which for most vehicle parts are the same for optimization purposes.			
`slidingFrictionCoef` name	**number** type	`frictionCoef` default	Sliding friction of the node
When this argument is not set, the sliding friction will default to the same value as static friction.			

Optional arguments – Powertrain

These arguments are used by powertrain components of the car such as the engine.

`engineGroup` name	**string** type	A different kind of group, related to powertrain simulation
Used by components such as the engine block, intake, exhaust, radiator, fuel tank, etc.		

2. Modding – JBeam File Sections – Nodes

back

Fire simulation

flashPoint name	**number** type	Temperature (°C) at which the node catches on fire.	
In documentation this is often referred to as the auto-ignition temperature.			
baseTemp name	**number** type	tEnv default	Initial temperature (°C) of the node.
Using "thermals" as the baseTemp will tie the node's temperature to the engine temperature.			
smokePoint name	**number** type	flashPoint default	Temperature (°C) at which the node emits smoke.
burnRate name	**number** type	How quickly the node burns through its fuel (Actual burn rate also depends on how much energy is left to burn and the current temperature of the node)	
conductionRadius name	**number** type	0 default	Maximum distance (meters) that the node will transfer heat to other nodes through conduction.
selfIgnitionCoef name	**number** type	0 default	How much energy from collisions heats up the node.
chemEnergy name	**number** type	0 default	How much energy is in the node (J).
specHeat name	**number** type	1 default	Specific heat of the node in J/kg/°C (Along with the node's weight, this affects how quickly the node heats up).

2. Modding – JBeam File Sections – Nodes

back

Advanced example

Example of a typical node section, which begins by initializing all the properties that will be used for the following nodes.

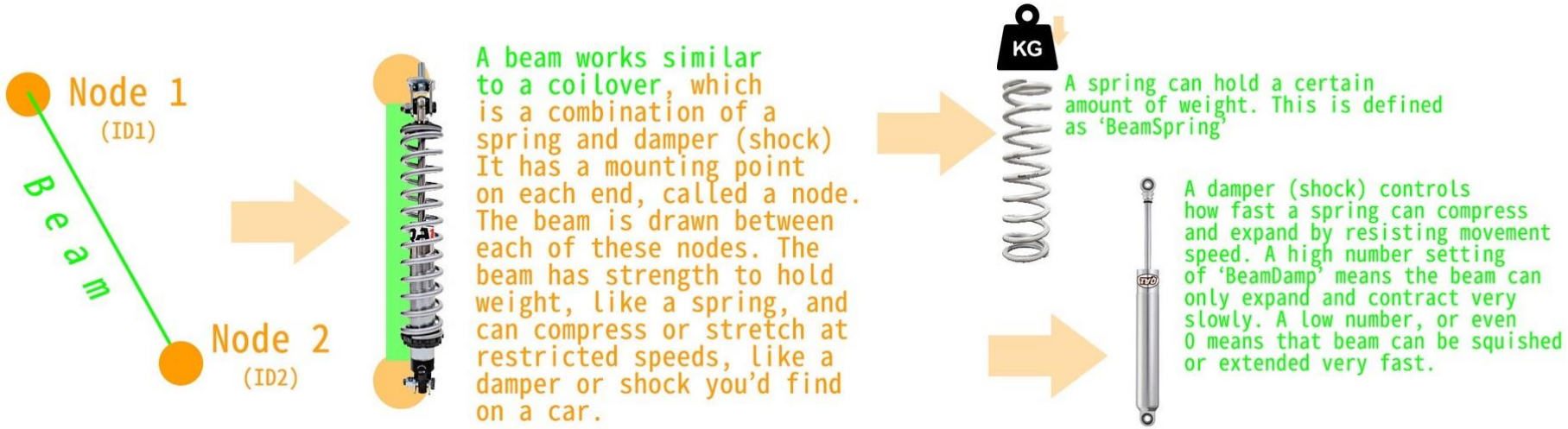
```
{
    "nodes": [
        ["id", "posX", "posY", "posZ"],
        {"nodeWeight":1.3},
        {"frictionCoef":0.6},
        {"nodeMaterial":"|NM_PLASTIC"},
        {"collision":true},
        {"selfCollision":false},

        // refnode
        ["fr4", 0.00000, -0.0862,  0.08203],

        {"group":"base"},
        ["fr1sr", 0.05127, 0.32888, 0.11144],
        ["fr1sl",-0.05127, 0.32888, 0.11144],

        {"frictionCoef":0.7},
        ["fr5sr",  0.05127, -0.17883, 0.11917],
        ["fr5sl", -0.05127, -0.17883, 0.11917],
        ["fr6sr",  0.05819, -0.25368, 0.11144],
        ["fr6sl", -0.05819, -0.25368, 0.11144],
        {"group":""},
      ],
}
```

2. Modding – JBeam File Sections – Beams

back

The origin of BeamNG's name – beams.

Beams are the spring-damper connections between nodes, forming the core structure of a vehicle. They enable deformation and breaking as needed.



There are multiple types of beams for all sorts of purposes, which you can see in more detail in the various examples. Standard beams act as simple springs with damping, with the ability to deform and break if needed, and are used for most applications.

2. Modding – JBeam File Sections – Beams

back

Required arguments

id1: name	string type	Name of the first node
Id2: name	string type	Name of the second node

Optional arguments – General

The Beams section has many optional arguments. We will divide them into categories, which will be sorted from most to least frequently used. These are the most basic arguments, used everywhere in Jbeam files, necessary for correct simulation.

beamType name	string type	NORMAL default	Beam type determines its behavior on compression and extension.

NORMAL - General use beams, don't have any special properties, used in most structures.
SUPPORT - The second most used type of beams. These only resist forces in compression, not in extension. With precompression, they start resisting forces when reaching the precompressed length. They can automatically break when reaching a set length. Usually used when multiple parts are linked together - they don't serve as attachments for those parts, but as limiters, preventing the nodes of one part to clip inside the surface of the other.

2. Modding – JBeam File Sections – Beams

back

Optional arguments – General

HYDRO - Defined in a separate section but sharing all normal beam properties, hydros can change length on demand, usually used for steering racks. They can also define the steering wheel lock angle of the car.

ANISOTROPIC - They have separate `beamSpring` and `beamDamp` values for expansion than for compression, can define transition zone between the two behavior variants, and can also automatically break when exceeding a certain length ratio. Used to simulate structures that are stiffer in expansion than in compression, such as tires (where they are automatically generated), ropes or soft tops.

BOUNDED - More complicated than anisotropic beams, these can have different `beamSpring` and `beamDamp` values not just for compression and extension, but also for low and high compression and extension speed. The contraction and expansion can be defined as a ratio or metric range, and the transition zone can be modified. They are computationally expensive to use and should usually only be used in suspension components such as dampers and bump stops.

2. Modding – JBeam File Sections – Beams

back

Optional arguments – General

LBEAM - Defined with three node IDs instead of two, they resist angle change between two beams created between those nodes, rather than a distance between two nodes. They can provide high lateral stiffness with low compression resistance. They are used for tires (where they are auto generated) and leaf springs.
PRESSURED - Special beams used to simulate air cylinders. Their forces are inversely proportional to their real time length. You can set the starting pressure in Pa or PSI, maximum pressure above which the beam breaks, surface of the cylinder and its volume coefficient. Used in tires (auto generated) and progressive air springs.
BROKEN - Broken beams stop resisting forces. You cannot set this type by typing it in Jbeam.

2. Modding – JBeam File Sections – Beams

back

Optional arguments – General

beamSpring name	**number** type	4300000 default	Rigidity of the beam (N/m).
Force required to change the length of the beam by a set amount. Excessively high stiffness compared to node weight can cause the beam to **vibrate** and cause **instability issues**. A beam with the `beamSpring` value of 0 will not be shown in the console as a duplicate. This allows for adding extra damping beams in the structure.			
beamDamp name	**number** type	580 default	Damping of the beam (N/m/s).
Damping helps reduce oscillations over time. However, if the damping is too high relative to the node's weight, it can lead to pulsing stress and instability in the structure.			
beamStrength name	**number** type	FLT_MAX default	Strength of the beam. (N).
How much force the beam can resist before breaking. A value of "**FLT_MAX**" will result in an unbreakable beam.			
beamDeform name	**number** type	220000 default	How much force (N) is required to deform the beam permanently.
A value of "**FLT_MAX**" will result in a beam that can't be permanently deformed.			

2. Modding – JBeam File Sections – Beams

back

Optional arguments – Breakable

Used only on breakable beams (the ones with `beamStrength` other than "**FLT_MAX**"). These kind of beams are used to simulate plastic or to attach various components of a vehicle together.

`breakGroup` `name`	**`string`** `type`		A beam gets automatically broken when another beam from the same breakGroup breaks
`breakGroupType` `name`	**`number`** `type`		Sets breakgroup behavior
If set to 0, this beam will break others in the `breakGroup`. if set to 1, this beam will NOT break others in the `breakGroup` but will be broken by the group.			
`optional` `name`	**`boolean`** `type`	`false` `default`	Deactivates errors when one of the beam's node is missing
This is used for cases where one of the nodes that make up the beam, is located in an optional component. It will also hide warnings related to duplicate beams.			

2. Modding – JBeam File Sections – Beams

back

Optional arguments – Deform Groups

Deform groups are used to trigger an action when a beam deforms. Most often used only for visual glass/lights damage and powertrain damage.

`deformGroup` name	**`string`** type	Identifies which deform group this beam is part of.
Used to trigger `flexbody` deform groups. Is also used for damage simulation on some powertrain components. A beam with a `deformGroup` will not be treated as a duplicate beam by the console. If a beam has both a `breakGroup` and a `deformGroup`, breaking it will spawn glass or wood particles depending on the `nodeMaterial` of both of its nodes. This is used for vehicle windows.		
`deformationTriggerRatio` name	**`number`** type	Level of deformation above which the `deformGroup` is triggered.
Typical values are very small numbers under 0.1. If the beam is shortened or lengthened by more than that amount, the `deformGroup` will be triggered. The existence of this property on a beam will create a deform trigger for the beam, causing extra code to be executed whenever any beam with this property is deformed. This can be performance intensive, so it should only be used where it's needed.		

2. Modding – JBeam File Sections – Beams

back

Optional arguments – Precompression

These arguments define a length change of the beam on spawn. They are most often used in **suspension** components.

`beamPrecompression` name	**number** Type	1 default	The length the beam will become as soon as it spawns.
2.0 would be twice the length, 0.5 would be half the length.			
`precompressionRange` name	**number** type	Beam length change on spawn in meters.	
Overrides `beamPrecompression`. A value of 0.2 would cause the beam to lengthen by 0.2 meters on spawn, while a value of -0.2 would cause the beam to shorten by 0.2 meters on spawn.			
`beamPrecompressionTime` name	**number** type	Time in seconds for precompressed beams to reach their requested length.	
Helps avoid deformation on spawn from precompressed beams violently going to their desired length.			

2. Modding – JBeam File Sections – Beams

back

Optional arguments – Drivetrain and suspension

Used only on certain suspension components and other critical vehicle parts.

name name	**string** type	Name of the beam.
Used by some systems to identify the specific beam to use.		
dampCutoffHz name	**number** type	Limits the vibration frequency (Hz) above which damping applies.
Only applies to normal, bounded, and l-beams. Used mostly with suspension components and other critical parts of the jbeam to help run higher damping before instability kicks in. Expensive to run, should only be used where needed.		

2. Modding – JBeam File Sections – Beams

back

Optional arguments – Suspension sound

These arguments are used on beams that work the roles of struts or shock absorbers in a vehicle. They should all be used together in line only on those beams.

soundFile name	**string** type		The FMOD event to play on beam compression.
The list of all usable event paths can be found in the World Editor: Window > Audio > SFX Previewer.			
volumeFactor name	**number** Type	1 default	Sets the relationship between the beam compression and sound volume.
decayFactor name	**number** Type	10 default	Sets the envelope decay/release factors of the sound.
pitchFactor name	**number** Type	0 default	Impulse-based pitch bend factor.
maxStress name	**number** Type	35000 default	The beam stress value (N/m^2) treated as full compression by the sound system.

2. Modding – JBeam File Sections – Beams

back

Bounded Beams

Bounded beams are a type of beam that can **vary** their **stiffness** and **damping** based on their **length** and **velocity**. They are mainly used for components like dampers and suspension limiters, as they allow for advanced behaviors such as multi-stage damping and bump stops.
However, because they are more complex, bounded beams can impact performance due to the increased computational cost. For this reason, their use should be kept to a minimum. In many cases, **precompressed** support beams can be used as a simpler alternative.

Unlike regular beams, you can place multiple bounded beams between the same two nodes to simulate different suspension elements, and the console will not report this as an error.

Required arguments

Same as generic beams.

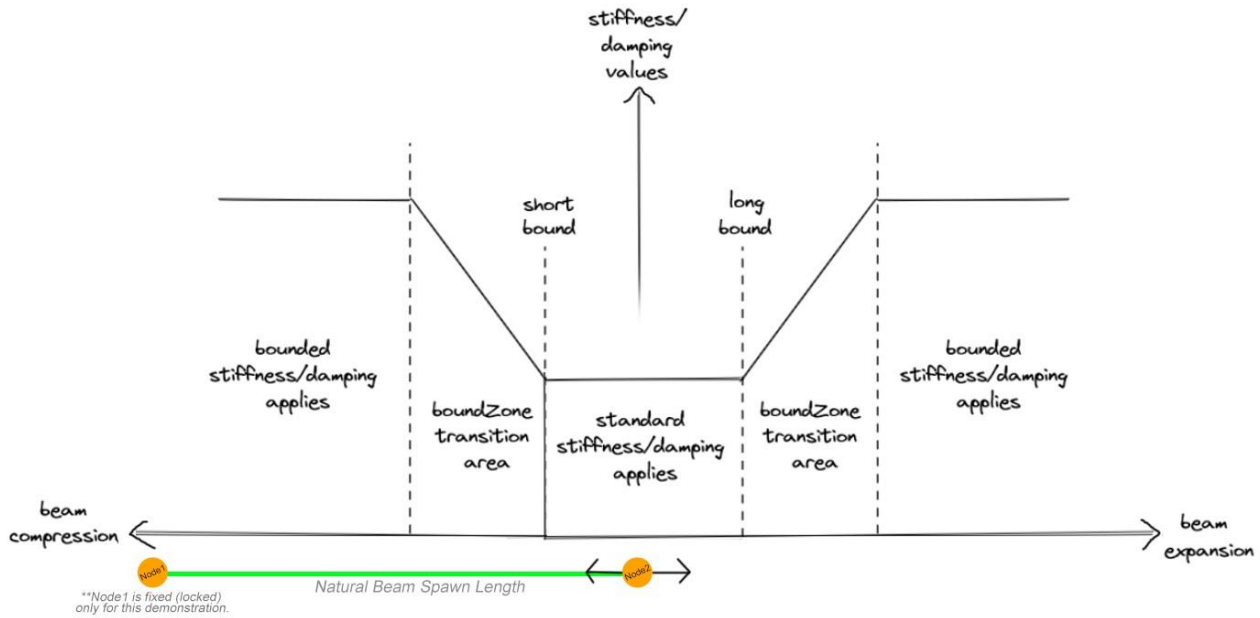
2. Modding – JBeam File Sections – Beams

back

Bounded Beams

Basic and advanced bound properties

The bounding effect lets a beam transition to a second set of stiffness and damping values when stretched or compressed past a limit. It's commonly used for bump stops, limit straps, or to restrict movement. The transition between normal and bound properties is gradual, controlled by the `boundZone` parameter.



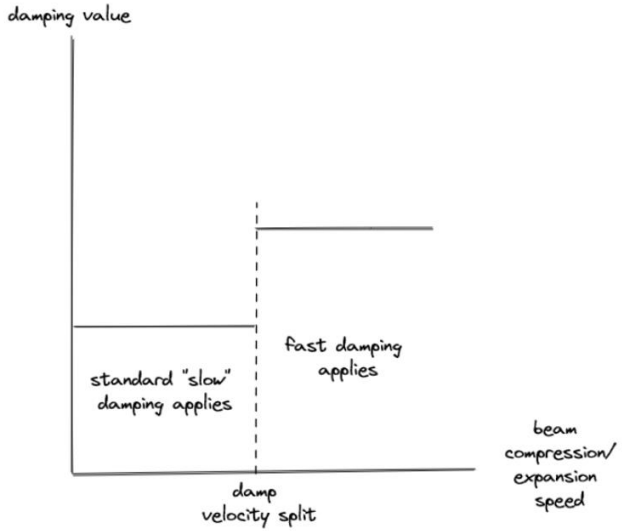
Damping can be split into bound/rebound and slow/fast. Regular damping applies during compression, while rebound damping works during **expansion**—useful for tuning weight transfer and suspension return.

2. Modding – JBeam File Sections – Beams

back

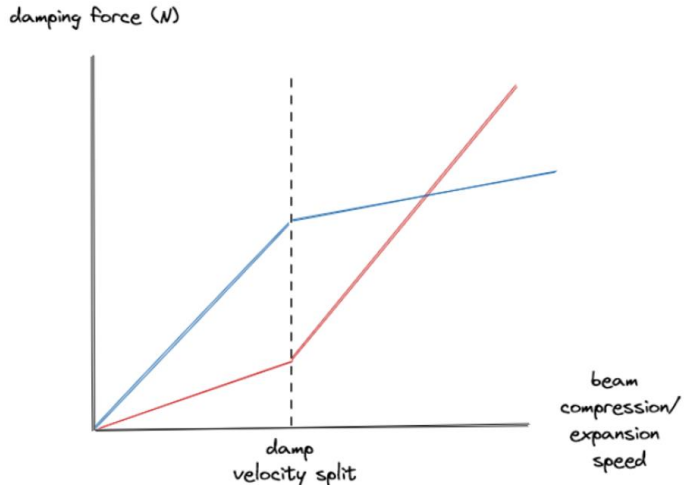
Bounded Beams

Basic and advanced bound properties



Fast damping is also split into bound and rebound, and it applies when the beam extends or compresses faster than a set velocity (defined by the `beamDampVelocitySplit`). On road and track cars, fast damping is usually lower than slow damping to make the suspension softer over harsh impacts like curbs or potholes, while slow damping still controls weight transfer. On off-road vehicles, fast damping is often closer to slow damping to better absorb large impacts and prevent bottoming out.

Vanilla cars often use damping beams on suspension/steering parts to prevent oscillations during grip loss, with little slow damping and higher fast damping.
The resulting forces will always keep going up as the speed increases, however the resulting damping force curves will change if the values are progressive (in red), or digressive (in blue)



2. Modding – JBeam File Sections – Beams

back

Bounded Beams

Optional arguments – Main parameters

beamLongBound name	**number** type	1.0 default	How far the beam can expand before limit properties begin to apply.
0.0 would mean that the limit properties begin to apply as soon as the beam expands. 0.5 would mean that the beam can expand to 150% of its spawned length before limit properties begin to apply. 1.0 would mean that the beam can expand to 200% of its spawned length before limit properties begin to apply.			
beamShortBound name	**number** type	1.0 default	How short the beam can go before limit properties begin to apply.
0.0 would mean that the limit properties begin to apply as soon as the beam contracts. 0.5 would mean that the beam can contract to 50% of its spawned length limit properties begin to apply. 1.0 would mean that the beam can contract its entire length before limit properties begin to apply.			

`boundZone`: Distance (in meters) over which the transition from default to limit properties occurs after reaching a bound.
`longBoundRange`: Extension length (in meters) before limit properties apply (overrides `beamLongBound`).
`shortBoundRange`: Compression length (in meters) before limit properties apply (overrides `beamShortBound`).

2. Modding – JBeam File Sections – Beams

back

Bounded Beams

Optional arguments – Limit stiffness and damping

`beamLimitSpring`: Spring stiffness (N/m) applied when a bound is reached.
`beamLimitDamp`: Damping (N·s/m) applied during compression after reaching a bound.
`beamLimitDampRebound`: Damping (N·s/m) applied during extension after reaching a bound.

Optional arguments – Advanced damping

beamDampRebound name	**number** type	beamDamp default	The damping value applied only when the beam is expanding(N/m/s).

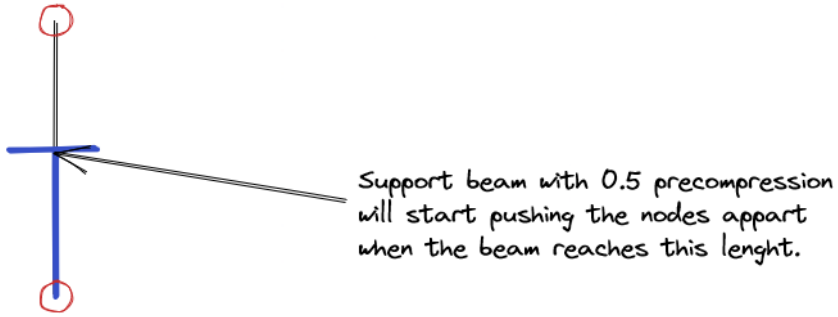
`beamDampFast`: Damping applied when the beam's movement speed exceeds `beamDampVelocitySplit`.
`beamDampReboundFast`: Fast rebound damping, active when extension speed exceeds the velocity split.
`beamDampVelocitySplit`: Speed threshold (m/s) at which `beamDampFast` and `beamDampReboundFast` take effect.
`beamDampVelocitySplitRebound`: Rebound-specific velocity split, overrides `beamDampVelocitySplit`.

2. Modding – JBeam File Sections – Beams

back

Support Beams

Support beams are beams that only resist to forces in **compression**.
When used with a precompression argument, they will only start resisting forces when being compressed smaller than their precompressed length.
Support beams are mostly used as **limiters**, to prevent structures from reversing themselves or getting stuck where they shouldn't.



Required arguments

Same as generic beams.

Optional arguments

beamLongBound name	**number** type	1.0default	When this bound is exceeded, the beam automatically breaks.
0.0 would mean the beam breaks if it expands at all. 0.5 would mean the beam breaks after expanding to 150% of its spawned length. 1.0 would mean the beam breaks after expanding to 200% of its spawned length.			

Support beams also support the same optional arguments as standard beams.

2. Modding – JBeam File Sections – Beams

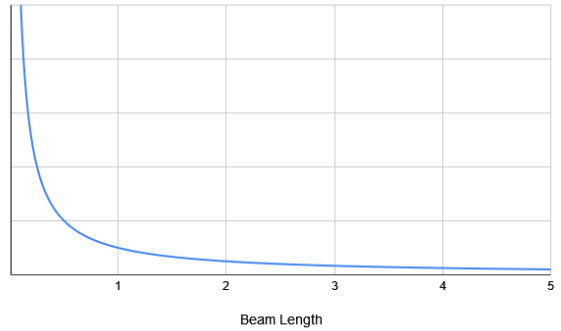
back

Pressured Beams

Pressured beams simulate progressive air springs that become stiffer as they compress. They are mainly used in tire sidewalls and specialized suspension setups.
Pressured beams inherit all optional parameters from standard beams.

Required arguments

Same as generic beams. The total force during extension = beamSpring + pressure-generated force. This prevents overextension and adds stability.



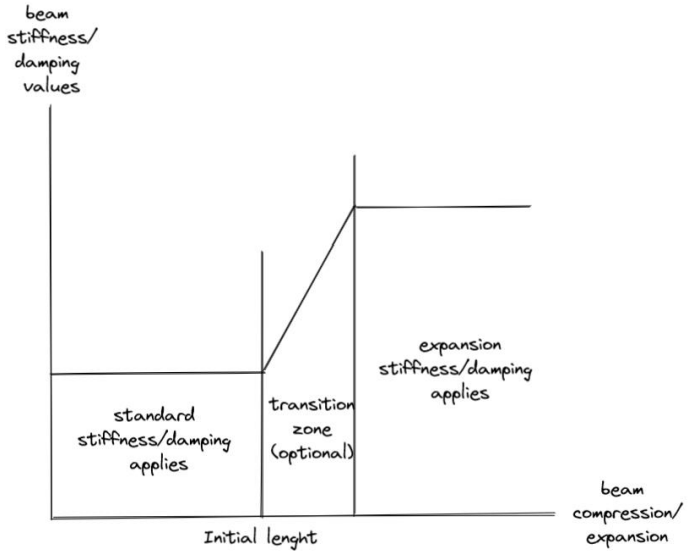
Optional arguments

`pressure`: Initial pressure in Pascals of the virtual air cylinder. Use this or `pressurePSI`, not both.
`pressurePSI`: Initial pressure in PSI of the virtual air cylinder. Use this or pressure, not both.
`surface`: Surface area (in m^2) of the virtual air piston. A larger surface creates more force for the same pressure. Default: 1.0.
`volumeCoef`: Ratio between beam length and the virtual cylinder length (1.0: same length; <1: pressure increases more gradually; >1: pressure increases more aggressively; 0: pressure stays constant and no pressure buildup).
`maxPressure`: Maximum pressure in Pascals before the beam breaks. Prevents instability by capping pressure as the beam compresses.
`maxPressurePSI`: Same as `maxPressure` but in PSI. Use one or the other, not both.

2. Modding – JBeam File Sections – Beams

back

Anisotropic Beams



Anisotropic beams have different spring and damping values depending on whether they are compressed or expanded beyond their original length. They are commonly used in tire sidewalls and rope-like elements, where you want compression flexibility but restrict overextension.

Required arguments

Same as generic beams.
The standard spring and damp arguments will be used when the beam is compressed.

Optional arguments

`springExpansion`: Spring stiffness (in N/m) when the beam is stretched beyond its spawned length. If not set, defaults to the normal `beamSpring`.
`dampExpansion`: Damping (in N·s/m) when expanded. If not set, defaults to `beamDamp`.
`beamLongBound`: Defines how far a beam can stretch before breaking, relative to its original length.
`transitionZone`: Defines a gradual ramp between standard and expansion values (instead of switching instantly).Expressed as a fraction of beam length.

2. Modding – JBeam File Sections – Beams

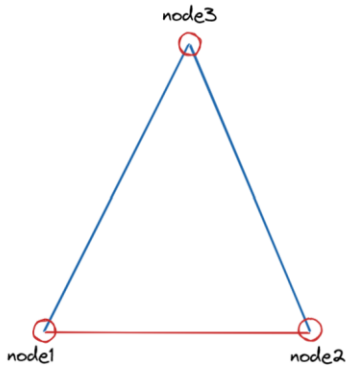
back

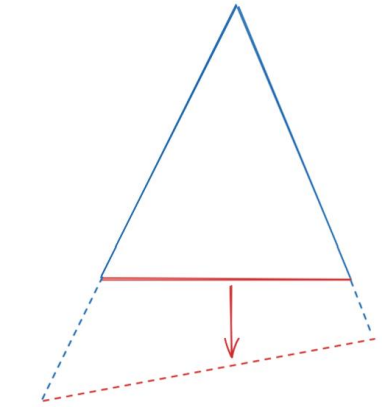
L-Beams

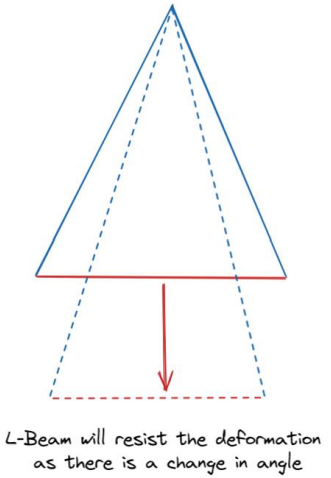
L-Beams are beams that are used to resist angle change between two beams. They are defined using three nodes, with the L-Beam being created between nodes 1 and 2. There must already be beams defined between nodes 1 and 3 and node 2 and 3.
L-Beams will apply a force only when the angle between the two standard beams change, and not if only the length of the l-beam changes.

Required arguments

Same as generic beams.







id3 name	string type	Name of the third node.
Any change in angle between beams [id1,id3] and [id2,id3] is then resisted by the L-Beam from [id1,id2]. `["node1","node2", {"id3:":"node3"}],`		

2. Modding – JBeam File Sections – Beams

back

Advanced example

```
{
   "beams": [
      ["id1:", "id2:"],

      // Beam telaio
      {"beamSpring":400000,"beamDamp":100},
      {"beamDeform":"FLT_MAX","beamStrength":"FLT_MAX"},
      ["w2l","w4l"],
      ["w2l","w2r"],

      // Molle-Ammortizzatori Anteriore
      {"beamSpring": "$spring_F", "beamDamp": "$damp_bump_F"},
      {"beamDeform": "FLT_MAX", "beamStrength": "FLT_MAX"},
      {"beamPrecompression": "$rideheight_F", "beamType": "|BOUNDED", "beamLongBound": 1.0, "beamShortBound": 1.0},

      ["s1r", "fw5r", {"beamDampRebound": "$damp_rebound_F", "soundFile": "art/sound/spring_compress2.ogg",
"volumeFactor": 1.8, "decayMode": 1, "decayFactor": 8, "pitchFactor": 20, "maxStress": 2000}],
      ["s1r", "fw3r", {"beamDampRebound": "$damp_rebound_F", "soundFile": "art/sound/spring_compress2.ogg",
"volumeFactor": 1.8, "decayMode": 1, "decayFactor": 8, "pitchFactor": 20, "maxStress": 2000}],

      {"beamSpring": 500000, "beamDamp": 1000},
      {"beamDeform": "FLT_MAX", "beamStrength": "FLT_MAX"},
      {"beamPrecompression": 1.0, "beamType": "|BOUNDED", "beamLongBound": 1.2, "beamShortBound": 0.85},
      ["fr1r", "fw4l"],
      ["fr2r", "fw4l"],
   ],
 [...]
}
```

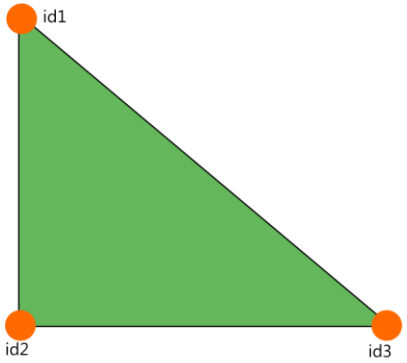
Example of typical beam sections, which begin by initializing all the properties that will be used for the following beams.

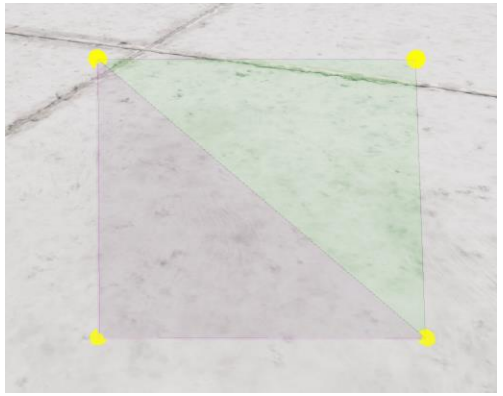
BeamNG

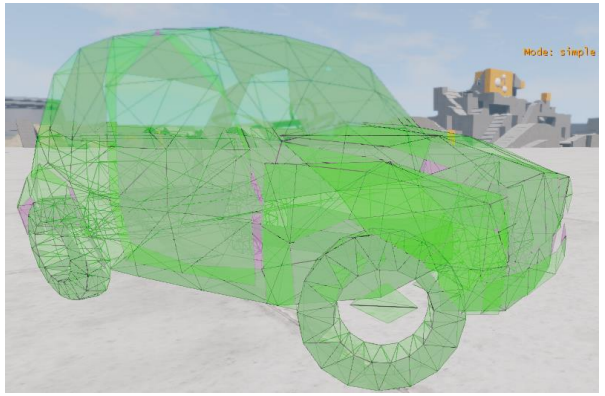
2. Modding – JBeam File Sections – Triangles

back

Triangles, also known as "coltris", are surfaces that fill-in the space between nodes. They are essential to allow for **collisions** between vehicles and are also used to define the **aerodynamic** properties of objects.
Triangles are defined by three nodes, chosen in a **counterclockwise** order.
Triangles enable collisions between jbeamed objects, **colliding only with nodes**—not with other triangles or static world elements. In debug view, the front side is shown in green and the rear in purple; exposing the front is preferred to prevent phasing during impacts. To flip a triangle, simply swap any two of its nodes.





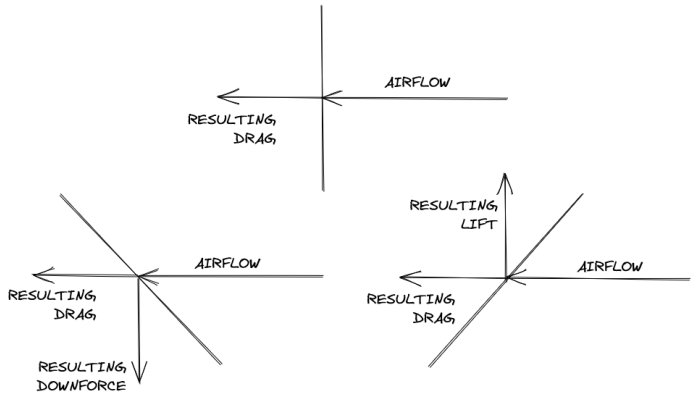


2. Modding – JBeam File Sections – Triangles

back

Aerodynamics

Regarding **aerodynamics**, BeamNG calculates drag on all triangles based purely on the speed of the airflow, the surface area and drag properties of the triangles. Drag and lift forces are then applied to the adjacent nodes.
Triangles do not influence each other, so a triangle at the back of the vehicle still generates full drag. Due to this, you may need to fine-tune the drag coefficient of different components depending on their exposure to airflow. The system also allows you to add surfaces to simulate ground effects or adjust the lift distribution on the vehicle. Lift is simulated based on the triangle's angle of attack relative to the airflow. An upward-tilted triangle generates lift, while a downward-tilted one creates downforce. A larger angle generates more lift, but beyond the stalling angle, lift decreases as the angle increases.



2. Modding – JBeam File Sections – Triangles

back

Required arguments

id1: Name	**string** type	The id of the first node that defines the triangle
id2: name	**string** type	The id of the second node that defines the triangle
id3: name	**string** type	The id of the third node that defines the triangle

Optional arguments

dragCoef name	**number** type	100 default	Drag coefficient of the triangle as a percentage of a flat plate of the same size.
Typical values are around 10 for most exposed body panels.			
liftCoef name	**number** type	dragCoef default	Lift coefficient of the triangle as a percentage of a flat plate of the same size.
Values between 80 and 120 are commonly used for spoilers.			

Quads place two triangles simultaneously, sharing all triangle properties. They require an `id4:` attribute and are defined in the same way. They are mainly used for perfectly rectangular surfaces but can make Jbeam files harder to understand when combined with triangles.

2. Modding – JBeam File Sections – Hydros

back

Hydros are beams whose length can be varied **on demand**. They're used for many things, including **steering racks**, hydraulic cylinders, and actuators for doors.
Their length is defined using **electrics** (more on this later).

A factor of 1 makes a command of 1 double the beam's length, while -0.5 halves it. Smaller factors reduce the amount of length change.

Required arguments

`id1:` name	**`string`** type	Name of the first node
`id2:` name	**`string`** type	Name of the second node

Optional arguments

`factor` name	**`number`** type	Extension/compression limit as a proportion of the hydro's length.
A value of 0.5 will result in the hydro halving in length at when the input is -1, and an extension to 1.5 times the hydro's initial length when the input is 1. Negative values will cause the hydro to **contract** instead of expanding with positive input values and vice versa.		

2. Modding – JBeam File Sections – Hydros

back

Optional arguments

inRate name	**number** type	2 default	How fast the hydro contracts.
outRate name	**number** type	inRate default	How fast the hydro expands.
steeringWheelLock name	**number** type	Defines how much steering wheel angle this hydro will match to.	
This is center to lock angle. Standard steering will be around 450~500, with older cars and trucks having slightly more angle, and racing cars having less angle. This drives the "steering" electric used for steering wheel props.			
lockDegrees name	**number** type	Sets the maximum steering angle (in degrees) that the hydro beam can achieve when fully actuated.	

Example

```
{
  "Buggy_Sterzatura": {
    "slotType": "Buggy_Sterzatura",
    "hydros": [
      ["id1:", "id2:"],
      ["fr2sr", "fw5r", {"factor":-0.08, "steeringWheelLock": 280, "lockDegrees": 25, "inRate": 1, "outRate": 2}],
      ["fr2sl", "fw5l", {"factor": 0.08, "steeringWheelLock": 280, "lockDegrees": 25, "inRate": 1, "outRate": 2}]
    ]
  }
}
```

2. Modding – JBeam File Sections – Information

back

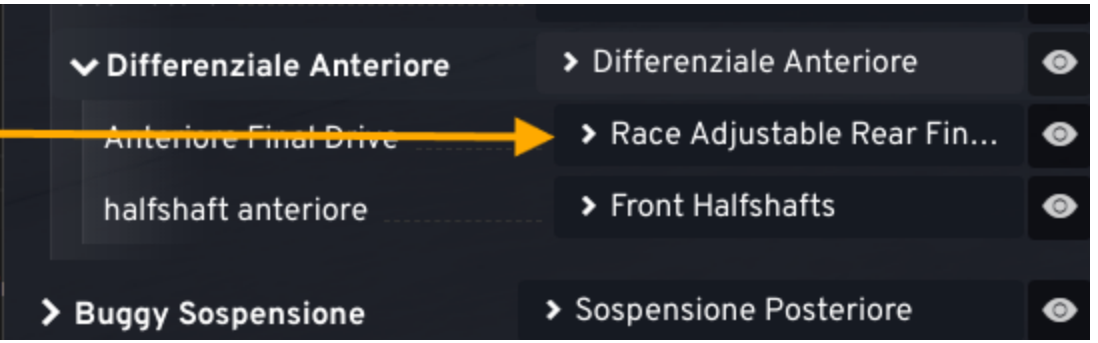
The **information** section contains data that is to be shown to the user in the User Interface (UI). The section is a simple dictionary with a few simple optional arguments:

authors name	**string** type	Name of the author(s).
Multiple authors can be separated by comma. "BeamNG" is reserved for official vehicles and should not be used.		
name name	**string** type	Visual name of the **part**.

Data in jbeam on the left, how it shows in-game on the right:

Part →
Section →

```
"Buggy_finaldrive_A": {
  "information": {
    "authors": "Stefano_Davide",
    "name": "Race Adjustable Rear Final Drive",
  },
  "slotType": "Buggy_finaldrive_A",
  "variables": [
    ["name", "type", "unit", "category", "default", "
    ["$finaldrive_F", "range", ":1", "Differentials",
```



2. Modding – JBeam File Sections – Slots

back

The **slot** and **slotType** sections are used to define where a component fits within the vehicle's parts tree. Each part also has a "slotType", which identifies where in the vehicle it fits, and allowing multiple parts to act as alternatives to each other. One unique slot type is "**main**", which refers to the "**root**"" component of the vehicle. The slots section defines which components are the "**children**" of that component based on their slotType.

Required arguments - `slotType`

`slotType` name	**`string/table`** type	The slot type, or list of slot types, of the component.
In almost all cases, this will be a string identical to the type of a slot in the parent component. However, it can also be a table specifying multiple slot types the part fits in.		

Required arguments - `slots`

`type` name	**`string`** type	The internal name of the slot.	
The parts in that slot should have this value as their slot type.			
`default` name	**`string`** type	The component that gets loaded by default if no part is defined in the car's config file.	
`description` name	**`string`** type	`type` default	The name of the slot in the part selector.

2. Modding – JBeam File Sections – Slots

Optional arguments

coreSlot name	**boolean** type	If enabled, the choice to have the part empty is removed from the part selector/garage.

How slots appear in game

```
"slotType" : "main",

"slots": [
    ["type", "default", "description"]
    ["Buggy_Telaio","Buggy_Telaio","Buggy Telaio"],
    ["Buggy_Motore","Buggy_Motore","Buggy Motore"],
    ["Buggy_Batteria","Buggy_Batteria","Buggy Batteria"],
    ["Buggy_Sospensione_A","Buggy_Sospensione_A","Buggy Sospensione Anteriore"],
    ["Buggy_Sospensione_P","Buggy_Sospensione_P","Buggy Sospensione Posteriore"],
```


```
"Buggy_Sospensione_A": {
  "information": {…
  },
  "slotType": "Buggy_Sospensione_A",
  "scalebeamSpring": 0.40,
  "scalebeamDamp": 0.2,
  "slots": [
    ["type", "default", "description"],
    ["Buggy_Ruote_A", "Buggy_Ruote_A", "Ruote_Anteriori"],
    ["Buggy_Differenziale_A", "Buggy_Differenziale_A", "Differenziale Anteriore"],
    ["Buggy_Sterzatura", "Buggy_Sterzatura", "Sterzatura"]
  ],
```

```
"Buggy_Sterzatura": {
  "information": {
    "authors": "Stefano_Davide",
    "name": "Sterzatura"
  },
  "slotType": "Buggy_Sterzatura",
  "hydros": [
    ["id1:", "id2:"],
    ["fr2sr", "fw5r", {"factor":-0.08, "steeringWheel
    ["fr2sl", "fw5l", {"factor": 0.08, "steeringWheel
  ]
}
```

2. Modding – JBeam File Sections – Flexbodies

back

Flexbodies are **visual models** that **deform** based on node movement and are assigned to specific nodes using nodegroups. The game will calculate deformation by mapping every vertex to the nearby nodes, and moving them accordingly with the movement of those nodes.
All the meshes should be in one or multiple **.dae** files either in your car's folder or the common folder.
An additional feature are **deform groups**: they trigger material changes when certain beams deform, often used for lights and windows.

```
"flexbodies": [
     ["mesh", "[group]:", "nonFlexMaterials"],
     ["my_mesh", ["my_group"]],
],
```

Required arguments

`mesh` name	**string** type	Defines the name of the mesh.
This is the same name as in **Blender**.		
`[group]:` name	**string** type	Defines the id of the node group this mesh is linked to.
A mesh can be linked to multiple node groups.		

`nonFlexMaterials` is a deprecated legacy feature that is not used or usable anymore.

2. Modding – JBeam File Sections – Flexbodies

back

Optional arguments

deformGroup name	string type	Defines the deform group that will be used for this mesh.
This name should match the deform group defined in the beams section.		

Example

```
    "flexbodies": [
        ["mesh", "[group]:", "nonFlexMaterials"],

        ["anteriore", ["telaio"]],
        ["giunto_bracci_anteriore3", ["telaio"]],
        ["giunto_molle_anteriore",   ["telaio"]],
        ["base_sensori1", ["telaio"]],
        ["base_sensori2", ["telaio"]],
        ["base_sensori3", ["telaio"]],
        ["base_sensori4", ["telaio"]],
        ["base_sensori5", ["telaio"]],
        ["base_sensori6", ["telaio"]],
        ["base_sensori7", ["telaio"]],
        ["base_sensori8", ["telaio"]],
        ["base_sensori9", ["telaio"]],
        ["giunto_molle_posteriore",   ["telaio"]],
        ["presa_alettone", ["telaio"]],
        ["alettone_posteriore", ["alettone"]],
    ],
```

2. Modding – JBeam File Sections – Camera

back

There are three different sections of **cameras** within jbeam. The `cameraExternal`, referring to the orbit camera, `cameraChase` referring to the chase camera, and `cameraInternal` which is used for the dash, hood and other custom cameras.
Normally, `cameraExternal` and `cameraChase` should both be defined **within the same component as the refnodes** (usually the frame or body of a vehicle depending on its construction).

Required arguments - `cameraExternal`

The external camera refers to the **orbit camera**. It is defined using the refnode as the center point and can be rotated freely by the player.

`distance` name	**number** type	How far the camera is from the vehicle (m).	
`distanceMin` name	**number** type	How close from the vehicle the camera can get (m).	
`offset` name	**dictionary** type	`{“x”:0, “y”:0, “z”:0}` default	The offset of the camera’s focus point (m).
If set at 0,0,0, the camera’s focus point will be the refnode.			
`fov` name	**number** type	Default field of view of the camera (degrees).	

2. Modding – JBeam File Sections – Camera

back

Required arguments - `cameraChase`

The **chase camera** has a mostly fixed rotation behind the car. It is defined using the refnode as the center point.

`distance` name	**number** type	How far the camera is from the vehicle (m).	
`distanceMin` name	**number** type	How close from the vehicle the camera can get (m).	
`defaultRotation` name	**dictionary** type	`{“x”:0, “y”:0, “z”:0}` default	The rotation of the chase camera (degrees).
`offset` name	**dictionary** type	`{“x”:0, “y”:0, “z”:0}` default	The offset of the camera’s focus point (m).
If set at 0,0,0, the camera’s focus point will be the refnode.			
`fov` name	**number** type	Default field of view of the camera (degrees).	

Example

```
"cameraChase":{
      "distance":2.0,
      "distanceMin":0.5,
      "defaultRotation":{"x":0,"y":-10,"z":0},
      "offset":{"x":0, "y":0.00, "z":0.35},
      "fov":65,
},
```

2. Modding – JBeam File Sections – Props

back

Props are meshes that do not affect physics themselves but can be animated. Attached to the physics skeleton via nodes. They are used for **decorative** or mechanical elements like driveshafts. Props follow node movement, can rotate or move based on defined parameters and react to physics events through `breakGroups` and `deformGroups`, allowing them to visually respond to damage.

Required arguments

`func`: The name of a variable or function (usually from the vehicle's Lua code or predefined animations) that controls the prop.
`mesh`: The name of the mesh to use for this prop.
`idRef`: The main node the prop is attached to (anchor point)..
`idX:,idY`: Nodes used to define the orientation of the prop in the X and Y directions.
`rotation`: Axis of rotation (e.g. {x:0, y:1, z:0} means it rotates around the Y axis).
`translation`: Optional movement of the prop in 3D space (usually {x:0, y:0, z:0}).

Optional arguments

`baseRotation`: The default rotation of the prop (in degrees) before any animation is applied.
`min / max`: Limits of movement (e.g. angle for rotation).
`offset`: Starting value for the animation.
`multiplier`: Scales the value received from func.
`optional`: Disables errors when one or more nodes of the prop were not found.

2. Modding – JBeam File Sections – Props

back

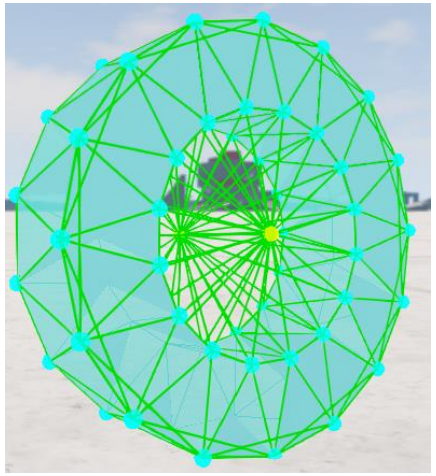
Example

```
"props": [
        ["func","mesh","idRef:","idX:","idY:","baseRotation","rotation","translation","min","max","offset","multiplier"],

        ["driveshaft_A","trasmissione_A","fr4r","fr4","fr3",{"x":90, "y":0, "z":0},{"x":0, "y":1, "z":0},{"x":0, "y":0,"z":0},
         -360, 360, 0, 1,{"breakGroup":"driveshaft_A","deformGroup":"driveshaft_A","optional":true}],
],
```

Make sure to set the origin of the prop meshes to the geometry origin in Blender, so BeamNG can properly handle their rotation and translation.

2. Modding – JBeam File Sections – Pressure Wheels

back



The `pressureWheel` is the **primary method** for creating **wheels** in BeamNG. It is a real time physical tire model made from **nodes**, **beams**, and **triangles**.
The pressure wheel system generates a wheel automatically using two nodes as an axle. The wheel's position and orientation are based on the position and direction of these nodes.
Due to the high number of parameters, it's recommended to copy existing pressure wheel setups from official JBeam files and adjust only what's needed.

Structure of the wheels

Wheels consist of two main parts:

- The **hub** connects to the axle using rigid beams and represents the rim's outer structure.
- The **tire** surrounds the hub with its own set of nodes and is connected to the hub via multiple beams. Together, they form a pressure group.

The beams between the hub and the axle are fairly rigid **standard** beams.
The tire nodes are linked to the hub node and each other with a **mix of beams** and allow for tread deformation and sidewall flex.

2. Modding – JBeam File Sections – Pressure Wheels

back

Structure of the wheels

The radial beams are the main weight bearing nodes on the tire. They are generated as **anisotropic** beams, allowing them to be quite soft in compression, to simulate the squishiness of tires, while still being strong in extension, keeping the tires from expanding at higher speeds. Reinforcement beams provide lateral stiffness by linking tire nodes across the wheel, allowing controlled sidewall flex during cornering.

Tire friction parameters – Friction velocity

Each ground surface in BeamNG has a static and sliding friction parameter, along with a `stribekVelocity` which affects the transition between static and sliding friction. The tire's various friction parameters are defined as multipliers of those various friction parameters.

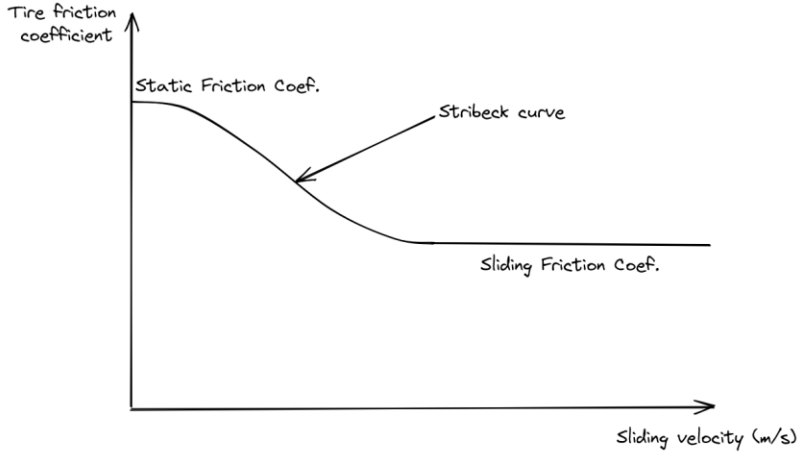
`frictionCoef` and `slidingFrictionCoef` are friction multipliers when the tire is rolling and sliding, with the second that is lower than the first one. To simulate the gradual transition from static to sliding friction in flexible rubber tires, the **Stribeck curve** is used. The `stribeckVelocity` parameter controls how quickly this transition happens—higher values make it more gradual. The `stribeckExponent` smooths the curve's shape, making it more bell-like.

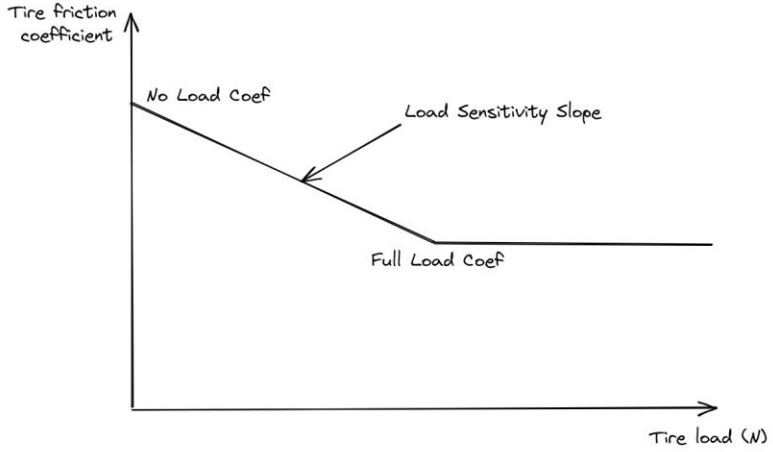
2. Modding – JBeam File Sections – Pressure Wheels

back

Tire friction parameters – Tire load

Unlike standard friction models where friction is directly proportional to load, tire friction behaves differently due to their flexibility. The `frictionCoefficient` is slightly higher at lighter loads, allowing for a more progressive response during braking and cornering.
Three parameters control tire load sensitivity: `noLoadCoef` (typically just above 1) defines friction at zero load, `fullLoadCoef` (usually just below 1) defines friction at high load, `loadSensitivitySlope` controls how quickly friction drops from `noLoadCoef` to `fullLoadCoef` as load increases.
As the force is calculated per node, the normal force used is the force on each individual node, not the force on the wheel itself.





2. Modding – JBeam File Sections – Pressure Wheels

back

Required arguments

```
"pressureWheels": [
        ["name","hubGroup","group","node1:","node2:","nodeS","nodeArm:","wheelDir"],
        [...]
]
```

name name	**string** type	The name of the wheel.
The standard naming for 4 wheeled vehicles is FR, FL, RR and RL. This will be used when referring to the wheel in other sections, like the powertrain.		
hubGroup name	**string** type	The nodegroup of the hub nodes.
Works in a similar way to standard node groups.		
group name	**string** type	The nodegroup of the tire nodes.
Works in a similar way to standard node groups.		
node1: name	**string** type	The first axle node.
The standard naming scheme for 4 wheeled vehicles is fw1rr, fw1ll, rw1rr and rw1ll.		
node2: name	**string** type	The second axle node.
The standard naming scheme for 4 wheeled vehicles is fw1r, fw1l, rw1r, and rw1l.		

2. Modding – JBeam File Sections – Pressure Wheels

back

Required arguments

nodeS name	**string** Type	Name of the stabilizing node.
Each node from the hub will attach to this node with a beam. It's mostly legacy and not generally used. Putting 9999 in will disable it.		
nodeArm name	**string** type	Along with nodeCoupling, this node will be used to apply **braking counter torque** to the suspension and body.
nodeArm should should located roughly where the brake caliper should be. Keep in mind that this node needs to be far enough from the nodeCoupling, and heavy enough. A node that is too close, or too light for the set braking torque can cause instability in the braking system, resulting in poor braking performance and brakes that overheat when the parking brake is applied. For a typical car, they should be at least 2kg. For a large truck, as much as 5-10kg.		
wheelDir name	**string** type	The direction that drive torque is applied.
The correct value (1 or -1) depends on which order you define the axle nodes.		

2. Modding – JBeam File Sections – Pressure Wheels

back

Advanced example

The `pressureWheels` structure is the most complex and parameter-rich. For this reason, in this section we prefer to show explanations only for the parameters set in the example.

```
    "pressureWheels": [
    ["name","hubGroup","group","node1:","node2:","nodeS","nodeArm:","wheelDir"],
    {"enableTireSideSupportBeams":true}, // Enables additional side support beams inside the tire to increase rigidity and prevent
deformation

    //general settings
    {"radius":0.1000}, // Radius of the tire nodes
    {"hubRadius":0.0515}, // Radius of the rim
    {"wheelOffset":-0.01}, // Offset from the original position (Left/right)
    {"hubWidth":0.04}, // Width of the rim
    {"tireWidth":0.0752}, // Width of the tire
    {"numRays":12}, // The amount of nodes to make the circle, more may result in smoother driving, but at the cost of performance,
weight & stability

    //hub options
    {"hubBeamSpring":1200000, "hubBeamDamp":125}, // Spring and damping values for the beams connecting the hub nodes
    {"beamSpring":1200000, "beamDamp":125}, // General spring and damping for the internal wheel structure
    {"hubBeamDeform":"FLT_MAX", "hubBeamStrength":"FLT_MAX"}, // Hub beams won't deform or break (maximum values)
    {"beamDeform":"FLT_MAX","beamStrength":"FLT_MAX"}, // Beams are indestructible (ideal for debugging or rigid setups)
    {"hubNodeWeight":0.20}, // Weight of the rim nodes, total rim weight = numRays * nodeWeight * 2
    {"hubNodeMaterial":"|NM_PLASTIC"}, // Material used for the hub (influences physical properties like friction and deformation)
    {"hubFrictionCoef":0.2}, // Friction of the hub nodes against other objects
[...]
```

2. Modding – JBeam File Sections – Pressure Wheels

back

Advanced example

```
[...]
    //tire options
    {"wheelTreadBeamSpring":180000,"wheelTreadBeamDamp":50,"wheelTreadBeamDeform":"FLT_MAX","wheelTreadBeamStrength":10000000}, //
Spring and damping values for beams across the tread (edge to edge of the tire)
    {"wheelPeripheryBeamSpring":180000,"wheelPeripheryBeamDamp":40,"wheelPeripheryBeamDeform":"FLT_MAX","wheelPeripheryBeamStrength":100
00000}, // Spring/damp for peripheral beams connecting the tire circumference
    {"springExpansion":180000, "dampExpansion":18}, // Expansion springs/damping pushing the tire radially outward
    {"nodeWeight":0.1}, // Weight of tire nodes
    {"nodeMaterial":"|NM_RUBBER"}, // Rubber material for tire behavior and grip
    {"frictionCoef":1.3}, // Tire friction for each node, increasing will add more grip to your vehicle

    //Pressure
    {"pressurePSI":10}, // Tire pressure in PSI; defines the internal "inflated" force pushing the nodes outward
    {"maxPressurePSI":1500}, // Maximum pressure allowed before the tire explodes
    {"reinforcementPressurePSI":23}, // PSI above which reinforcement beams activate to maintain shape and avoid deformation
    {"pressureSpring":360100}, // Internal spring force that simulates air pressure in the tire
    {"reinforcementPressureSpring":2044000}, // Spring force for reinforcement beams that activate above reinforcement PSI
    {"pressureDamp":50}, // Damping effect for internal air pressure simulation
    {"reinforcementPressureDamp":53}, // Damping for reinforcement pressure beams
[...]
```

2. Modding – JBeam File Sections – Pressure Wheels

back

Advanced example

```
[...]
    //Brake
    {"brakeTorque":10}, // Braking torque applied to this wheel
    {"parkingTorque":0}, // Torque applied when the parking brake is active
    {"enableBrakeThermals":false}, // Disables heat simulation for brakes
    {"brakeDiameter":false}, // Placeholder for brake diameter (not active)
    {"brakeMass":false}, // Placeholder for brake mass (not active)
    {"brakeType":false}, // Placeholder for brake type (not active)
    {"rotorMaterial":false}, // Placeholder for rotor material (not active)
    {"brakeVentingCoef":false}, // Placeholder for brake venting coefficient (not active)

    //front
    {"propulsed":0}, // Indicates if the wheel is driven by the engine (0 = not powered)
    {"selfCollision":false}, // Wheel nodes don't collide with each other (common for front wheels)
    {"collision":true}, // Enables collision with external objects
    {"axleBeams":["axle_AD"]}, // Beam(s) that simulate the axle for this wheel
    ["AD", "ruota_AD", "", "fw1rr", "fw1r", 9999,  "fw3r", 1], // Actual wheel definition: name, group, node references, rotation
direction

    {"axleBeams":["axle_AS"]}, // Axle beam for the second front wheel
    ["AS", "ruota_AS", "", "fw1ll", "fw1l", 9999,  "fw3l",-1], // Second front wheel setup

    {"axleBeams":[]}, // Clears axle beams for following wheels (if any)
    {"enableABS":false}, // Disables ABS simulation for this wheel
    {"selfCollision":true} // Enables self-collision again for following wheels or parts
],
```

2. Modding – JBeam File Sections – Variables

back

Variables are used for all the settings that you'd like the user to have access to in the "**Tuning**" setting.
Variables can either be used by themselves, or as part of a function. They can be used pretty much anywhere in the jbeam that uses a number as it's entry value.

Required arguments

```
"variables": [
     ["name", "type", "unit", "category", "default", "min", "max", "title", "description"],
     [...],
]
```

name name	**string** type	The internal name of the variable, that will be used when referring to your variable in the rest of your jbeam (the $ as first character is usually used).
type name	**string** type	The type of variable("range" is the only supported type).
unit name	**string** type	The units that will be shown in the tuning menu.
Leaving this blank will result in the slider being from 0% to 100% and won't show the min and max values in the tuning menu.		

2. Modding – JBeam File Sections – Variables

back

Required arguments

category name	**string** type	The category under which the variable will appear in the tuning menu.
This can be anything you want. When loading the car, the game will take all variables with the same category and group them together.		
default Name	**number** type	The default value of the variable.
min name	**number** type	The minimum value of the variable.
max name	**number** type	The maximum value of the variable.
title name	**number** type	The name of the variable that will be shown in the tuning screen.
description name	**string** type	A description that will be shown if the user hovers his mouse above the variable.

The editable variable is identified by the $ symbol in front of its name, and once defined this way, it can be referenced in the scripts using the same name.

2. Modding – JBeam File Sections – Variables

back

Optional arguments

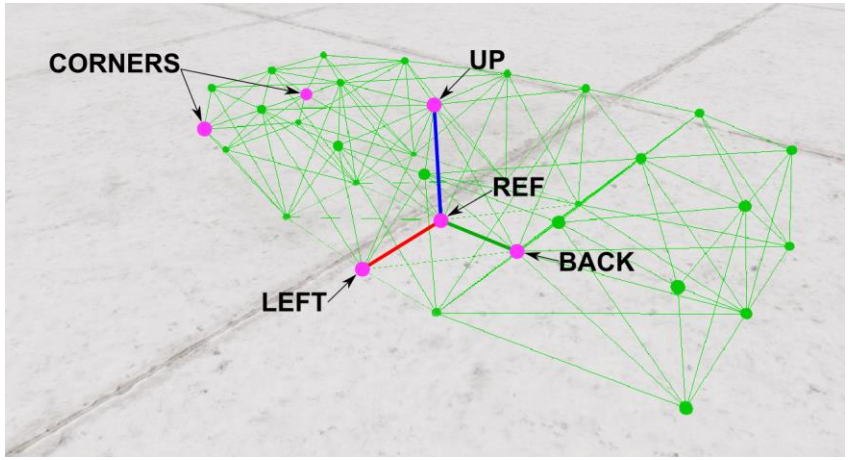
subCategory name	**string** type	The sub-category under which the variable will appear in the tuning menu.
This can be anything you want. When loading the car, the game will take all variables with the same category and sub-category and group them together. This is often used to split Front and Rear variables.		
stepDis name	**number** type	The size of the steps on the slider.
Leaving the Unit argument blank will result in the step being 1% which cannot be changed here.		

Advanced example

```
"variables": [
        ["name", "type", "unit", "category", "default", "min", "max", "title", "description"]
        ["$rideheight_R", "range", "", "Sospensione", 1.10, 0.98, 1.5, "Altezza Sospensione", "Aumenta o diminuisci l'altezza delle
sospensioni", {"stepDis":0.05,"subCategory":"Posteriore"}]
        ["$spring_R","range", "N/m", "Sospensione", 10000, 1000, 30000, "Rapporto Rigidita", "Rigidita sospensione", {"stepDis":500,
"subCategory":"Posteriore"}]
        ["$damp_bump_R", "range", "N/m/s", "Sospensione", 1300, 10, 3000, "Smorzamento in compressione", "Rapporto di compressione
lenta", {"stepDis":100, "subCategory":"Posteriore"}]
        ["$damp_rebound_R", "range", "N/m/s", "Sospensione", 1300, 10, 3000, "Smorzamento in estensione", "Rapporto di estensione
lenta", {"stepDis":100, "subCategory":"Posteriore"}]
],
```

2. Modding – JBeam File Sections – Refnodes

back



Refnodes are **fundamental** and are used to define the orientation and position of the car, as well as the position of the cameras. They consist in a set of 4 nodes that define a coordinate system, along with 2 nodes that define the front of the vehicle.

A single set of refnodes **must be present on the vehicle at all times**, meaning this should be part of your chassis or body jbeam. A vehicle with no refnodes will fail to spawn.

While refnodes do not need to be precisely centered in the vehicle, all refnodes (excluding the corner nodes) need to be **perfectly aligned with each other in their respective direction**, otherwise you might experience camera and/or spawning issues. You might need to do some adjustments to your jbeam for this to be possible, and in some cases the addition of nodes that serve solely as `refNodes` might be required.

2. Modding – JBeam File Sections – Refnodes

back

Required arguments

```
"refNodes":[
    ["ref:", "back:", "left:", "up:", "leftCorner:", "rightCorner:"]
    [...],
],
```

ref: name	**string** type	A node near the center and lowest point of the vehicle body.
This node doesn't need to be in the 0,0,0 position but should be close to it.		
back: name	**string** type	A node directly behind the ref node (Y axis).
left: name	**string** type	A node directly left of the ref node (X axis).
up: name	**string** type	A node directly above the ref node (Z axis).
leftCorner: name	**string** type	A node at the front left corner of the vehicle.
Used to trigger checkpoints.		
rightCorner: name	**string** type	A node at the front right corner of the vehicle.
Used to trigger checkpoints.		

2. Modding – JBeam File Sections – Controller

back

Controllers are used to simulate various special functions of a vehicle using lua. It includes things like digital gauges, **drive modes**, lightbars, stability control, etc.
Every vehicle needs at least one **main** controller, with the main options being `dummy` or `vehicleController`.
Vanilla controllers can be found inside "\lua\vehicle\controller" in the game's install folder.

Vehicle Controller

The vehicle controller is the main controller for any powered vehicle in BeamNG. It manages the **powertrain controls** and interfaces with the different shift logic controllers for the various transmission types. It is the main source of electrics data for engine and transmission information. The **vehicleController** in our case is used to control the behavior of the vehicle's engine.

The way these sections are used will be covered in the second part.

```
"controller": [    //defined in the main jbeam of the vehicle
     ["fileName"],
     ["vehicleController", {}],
],
```

```
"vehicleController": {    //defined in the engine jbeam file
     "shiftLogicName": "electricMotor",
     "motorNames": ["engine"],
     "topSpeedLimitReverse": 15,
     "defaultRegen": 0.0,
     "brakeRegenCoef": 0.0,
},
```

2. Modding – JBeam File Sections – Energy storage

back

Energy storage is used to identify a tank of fuel, a set of batteries, or a tank of nitrous.

Required arguments

type name	**string** type	The type of energy stored.	
The available options are `fuelTank`, `n2oTank` and `electricBattery`. Each of those has unique parameters.			
name name	**string** type	The name of the energy storage.	
In case of a vehicle with multiple fuel tanks, each name must be unique.			
batteryCapacity name	**number** type	0 default	Battery capacity at full charge (kWh).
startingCapacity name	**number** type	capacity default	Initial charge of the battery.

Example

```
"energyStorage": [
      ["type", "name"],
      ["electricBattery", "mainBattery"]
],
"mainBattery": {
      "batteryCapacity": 10,
      "startingCapacity": "$fuel",
},
```

2. Modding – JBeam File Sections – Powertrain

back

A **powertrain** is the system of parts that make a vehicle move, including the **engine**, **transmission**, **driveshaft**, **axles**, **differentials** and **wheels**.

Overview of customizable powertrain components

Engine Types:

- **Internal Combustion Engine (**ICE) – Traditional engine types powered by fuel (gasoline, diesel, etc.).
- **Electric Motor** – Uses electricity instead of fuel, offering instant torque and regenerative braking.
- **Hybrid Powertrain** – A combination of an ICE and an electric motor, balancing efficiency and power.
- **Turbine Engine** – A high-power engine often used in experimental or jet-powered vehicles.

Transmission Types:

- **Manual Transmission** (MT) – Driver-controlled gearbox with a clutch.
- **Automatic Transmission** (AT) – Gear shifts are controlled automatically.
- **Continuously Variable Transmission** (CVT) – Uses belts and pulleys to offer smooth, stepless gear changes.
- **Dual-Clutch Transmission** (DCT) – A faster-shifting automatic transmission with two clutches.
- **Sequential Transmission** – Found in race cars; allows clutchless up/down shifting.
- **Electric Direct Drive** – Used in electric vehicles, where no traditional transmission is needed.

2. Modding – JBeam File Sections – Powertrain

back

Overview of customizable powertrain components

Differentials:

- **Open Differential** – Standard type, distributing power evenly but allowing independent wheel speeds.
- **Limited-Slip Differential (LSD)** – Reduces wheel slip by redistributing torque between wheels.
- **Locking Differential** – Forces both wheels on an axle to rotate at the same speed for better off-road traction.
- **Torque-Vectoring Differential** – Actively adjusts power to different wheels for better handling.

Transfer Cases (for AWD/4WD Vehicles):

- **Part-Time 4WD** – Allows switching between 2WD and 4WD manually.
- **Full-Time AWD** – Power is always distributed to all four wheels.
- **Selectable Transfer Case** – Lets the driver switch between different drive modes (2WD, AWD, 4WD High/Low)

Driveshafts & Halfshafts:

- **Driveshaft** – Transfers power from the engine/transmission to the wheels.
- **Halfshaft (Axles)** – Connects differentials to the wheels and transmits power.

2. Modding – JBeam File Sections – Powertrain

back

Overview of customizable powertrain components

Clutches & Torque Converters:

- **Clutch** – Used in manual transmissions to engage/disengage power from the engine to the gearbox.
- **Torque Reactor** – The point in the drivetrain where torque reaction forces are balanced: parts **before** it twist one way and parts **after** it twist the other way. Essential for realistic drivetrain behavior in BeamNG.

Final Drive & Gearing:

- **Final Drive Ratio** – Determines how engine power is translated into wheel rotation speed and torque.
- **Gear Ratios** – Affect acceleration, top speed, and fuel efficiency.

To demonstrate how to customize a powertrain, we will use the vehicle we created as an example.

2. Modding – JBeam File Sections – Powertrain

back

Required arguments

```
"powertrain": [
     ["type", "name", "inputName", "inputIndex"],
     [...]
],
```

type name	**dictionary** type	The kind of component (electricMotor, combustionEngine, differential, clutch, gearbox, etc.).
name name	**dictionary** type	A unique name for this component.
inputName name	**dictionary** type	The name of the element **providing input** to this component.
dummy means no mechanical input (it's an independent, primary source).		
inputIndex name	**number** type	Used when the input component has multiple outputs (like a differential).
For dummy this is always 0.		

Engine definition

This section defines all the physical and logical behavior of the motor. It must be located inside the vehicle's **motor jbeam file**.

2. Modding – JBeam File Sections – Powertrain

back

Required arguments – Torque

```
"torque": [
        ["rpm", "torque"],
        [...],
]
```

rpm name	**number** type	Engine revolutions per minute.
torque name	**number** type	Torque (in Nm) produced at that rpm.

Optional arguments

`maxRPM`: This is the maximum RPM the motor can spin at. Above this limit, it will protect itself and cut power.

`inertia`: Rotational inertia of the motor — the higher the value, the more "sluggish" the motor feels when accelerating or decelerating. Lower = quicker response.

`friction`: Static internal resistance of the motor (torque that opposes movement).

`dynamicFriction`: Resistance during rotation.

`electricalEfficiency`: How efficient the motor is.

`regenThrottle`: At what throttle position regen starts.

2. Modding – JBeam File Sections – Powertrain

back

Optional arguments

`maxRegenTorque`: Maximum torque the motor can absorb when regenerating.
`electricsThrottleFactorName`: Name of a parameter in the electrics system that scales available throttle (like a limiter).
`energyStorage`: List of batteries the motor draws power from.

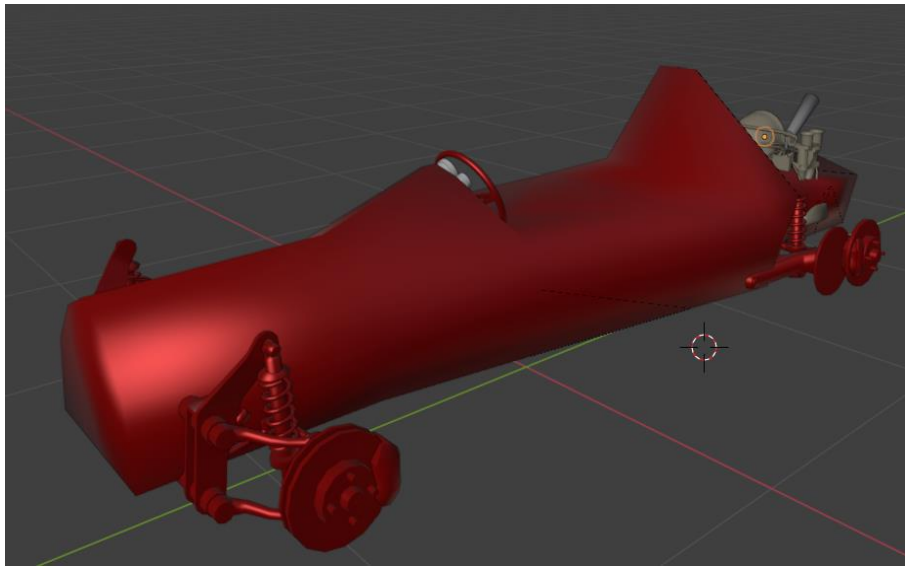
`waterDamage`: Which node groups cause engine damage when submerged in water.
`engineBlock`: Node group representing the physical engine body.
`breakTriggerBeam`: When this beam breaks, the engine is considered damaged or dead.
`uiName`: What the engine is called in the user interface.

3. Tutorials

To better understand modding, we completed the official Autobello tutorial from the official repository:

[Autobello Tutorial]

We **highly recommend** completing the Autobello tutorial before starting any vehicle modding. It provides a solid understanding of how Blender and BeamNG work together, and how the node and beam physics system functions.



Part 2: Vehicle creation

Inside the mod "Desert Buggy V3" you will see these files:

1. Jbeam scripts

- Buggy.jbeam
- Buggy_Telaio.jbeam.jbeam
- Buggy_Sospensione_A.jbeam
- Buggy_Sospensione_P.jbeam
- Buggy_Motore.jbeam
- Buggy_Batteria.jbeam
- Buggy_Ruote_A.jbeam
- Buggy_Ruote_P.jbeam

2. Informations script

- Info_Desert_Buggy.json
- Desert_Buggy_Race.pc
- Desert_Buggy_Offset.pc
- Info.json
- main.materias.json
- Name.cs

3. Vehicle design

- Default.jpg
- Desert_Buggy_Race.png
- Desert_Buggy_Offset.png
- Buggy.dae

Part 2: Vehicle creation

In the following slides, we will explain the steps needed to understand the purpose of the files in the mod and how to create them from scratch. The process will be divided into the following phases:

1. **Creating the 3D Model (.dae file) – Blender Phase**
   - Vehicle creation in Blender
   - Material and UV map assignment in Blender
   - Creation and use of the main.material file
2. **Creating the JBeam Scripts – JBeam Phase**
   - Hierarchy of the JBeam structure
   - Definition of beams and nodes
   - Definition of flexbodies and refnodes
   - Powertrain structure and vehicleController
   - Definition of wheels and hydros
3. **Defining the Package Structure – Configurations Phase**
   - Packages and Vehicles
   - Info.json, .pc and png files

1) Creating the 3D Model (.dae file) – Blender Phase

back

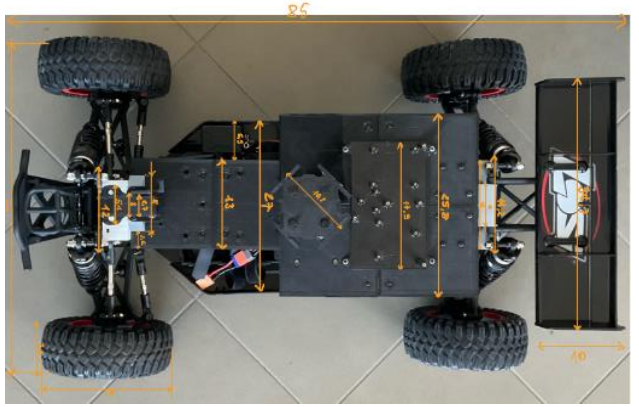
1) Vehicle creation in Blender

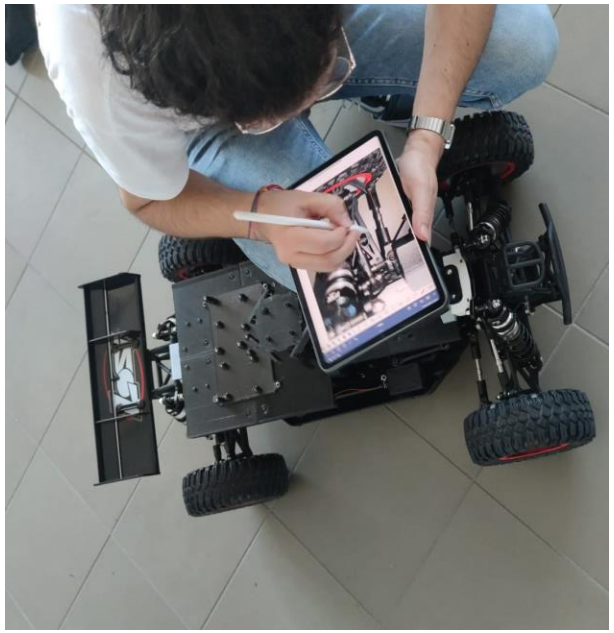
Before opening Blender, it is important to take a preparatory step:

- **Obtain 1:1 scale measurements**

If you're recreating a well-known vehicle, you can easily find its dimensions and specifications online. However, whenever possible, you can also take the measurements yourself, as we did.

Take a lot of photos of the vehicle and save the measurements on a pdf files are also a good practice, but to create the vehicle it's very important to take 6 photos:





1) Creating the 3D Model (.dae file) – Blender Phase

back

1) Vehicle creation in Blender

- 1 photo from the left side and 1 photo from the right side
- 1 photo from the front side and 1 photo from the rear side
- 1 photo from the top side and 1 photo from the bottom side

After gathering all the necessary measurements, we can proceed by opening Blender and starting a blank file.
This guide does not cover the basics of the Blender environment. Therefore, if you are new to Blender, we recommend watching a tutorial before continuing.

Blender Tutorial

After understanding how Blender works and how to import a **.dae** file into your environment, we can begin creating the vehicle.

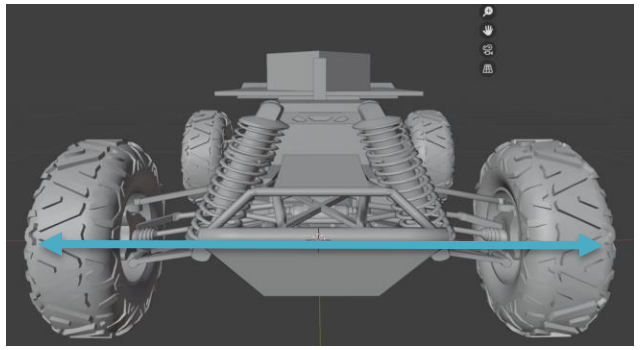
1) Creating the 3D Model (.dae file) – Blender Phase

back

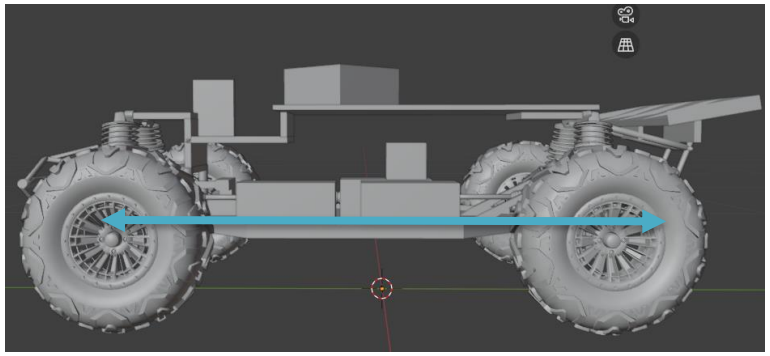
1) Vehicle creation in Blender

The first thing to do is to decide the setting of the axis, we suggest to use:

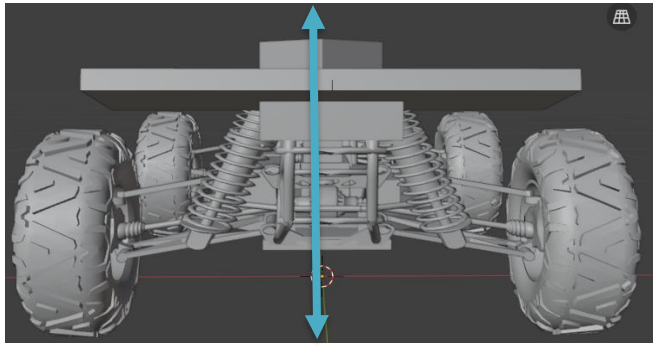
X-axis to the:



Y-axis to the:



Z-axis to the:



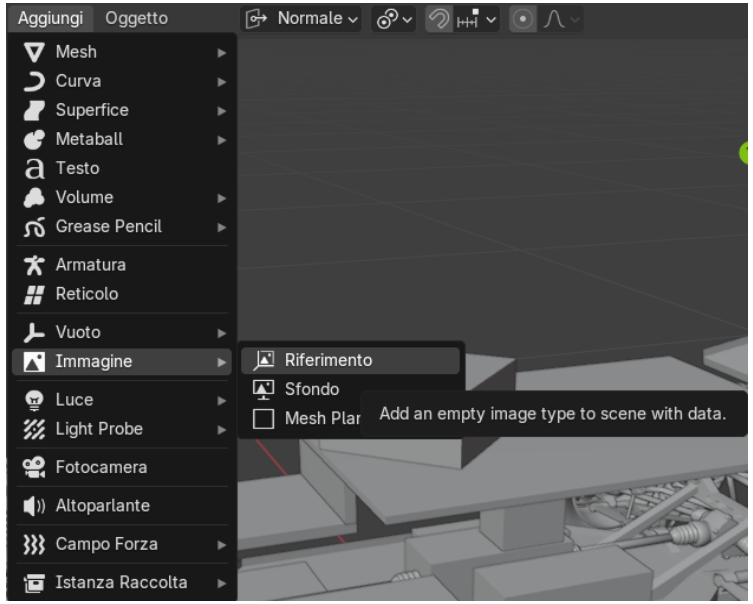
Place the center of the **X-Y axis** at the exact center of the vehicle. This will simplify some operations later on (refnodes). For the **Z-axis**, do not set it at the center of the vehicle. Instead, use only the positive part of the axis, considering the floor at level zero.
Also, we suggest to use negative numbers for the left part of the vehicle (X-axis) and for the rear part of the vehicle, but you can choose also different setting.

1) Creating the 3D Model (.dae file) – Blender Phase

back

1) Vehicle creation in Blender

Before creating the vehicle's parts, we can import the photos taken previously into the Blender environment to use them as reference images for the project. It's important to do it to have a practice reference of the vehicle.



To add a reference image to your blender project you can select "add" then "image".

Then place the images you took precisely at the center of the axis, ensuring they are properly aligned, and scale every photos to set the right measurement.

You can use the **measurement tool** in Blender to check the dimensions and confirm proper alignment.

1) Creating the 3D Model (.dae file) – Blender Phase

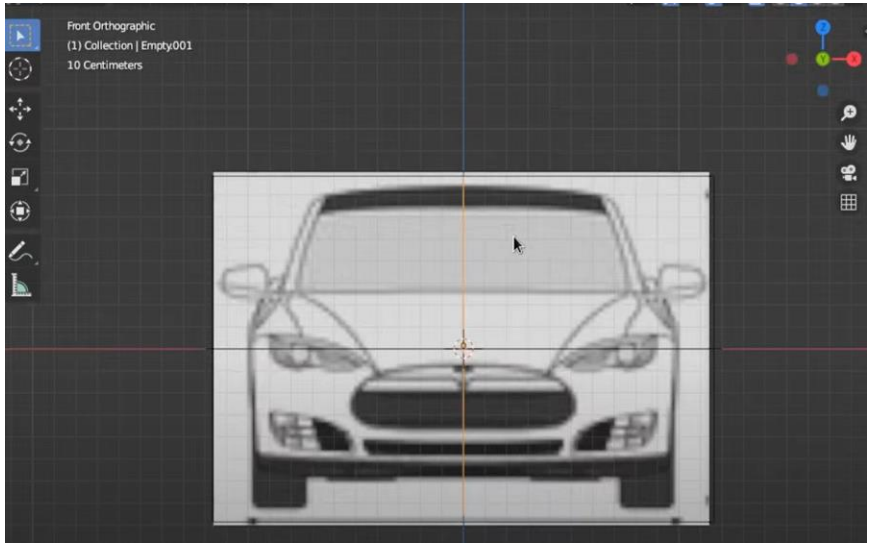
back

1) Vehicle creation in Blender

You can also adjust them, so they are visible from only one side, like viewing a cube. You can find detailed instructions on how to set this up in the first part of this tutorial:

Reference Tutorial

After the photo placing you will see something like this for every face of the vehicle.



After placing the photos, you will see something like this for each face of the vehicle.
Be careful with the photos, especially those taken with cameras, as they may cause scaling issues. We recommend using them only as a visual reference and always verifying measurements with the values you recorded earlier.

1) Creating the 3D Model (.dae file) – Blender Phase

back

1) Vehicle creation in Blender

Before starting the creation of the vehicle's parts, we need to establish an overview of the components we need to model.
In the **Scene menu** at the top right of the screen, it is essential to organize the project into separate collections, for example:

- **Front Suspension** folder
- **Rear Suspension** folder
- **Wheels** folder
- **Chassis and Support** folder
- **Engine and Transmission Components** folder

Keep in mind that the vehicle should be **symmetrical along the Y-axis**. Additionally, most vehicles use the same suspension system for both the front and rear, meaning many parts can be duplicated and mirrored in the correct direction to create the opposite side of the vehicle.
Inside the collection we can define all the pieces we need to create.

1) Creating the 3D Model (.dae file) – Blender Phase

back

1) Vehicle creation in Blender

Now we can start creating the vehicle's parts. You can also import **.dae** files from other BeamNG vehicles into a separate Blender file.
These files can be found inside the **content** folder of the game.

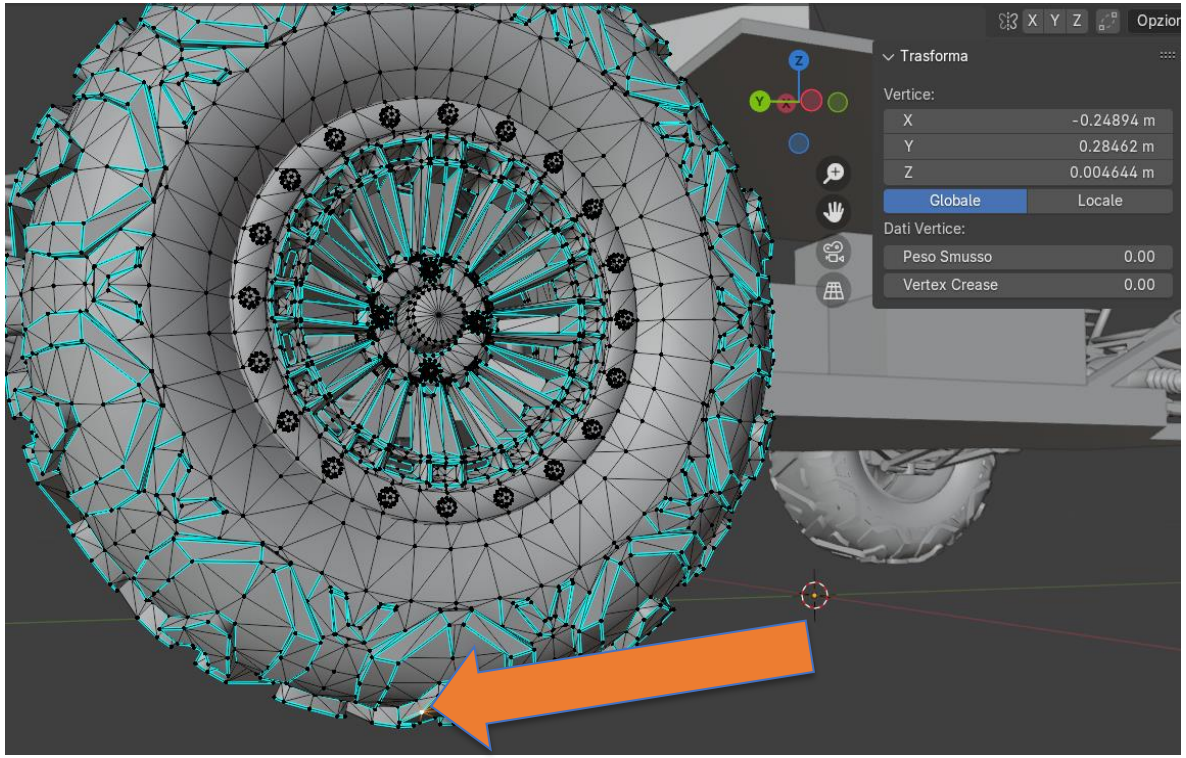
The 3D models included in the game files are well-structured and highly detailed. We recommend using them as inspiration or even directly borrowing certain parts from the **.dae** files of the vanilla vehicles.
To add a part to your project, simply **copy** (**Ctrl + C**) **and paste** (**Ctrl + V**) the desired model in Blender, and it will automatically be placed in your file, here you can **scale** that by pressing the **S** key.
Unfortunately, for our project we couldn't find a perfect reference within the vanilla files. However, we did use some specific components, such as **wheels** and **coilovers** (**springs** and **damper groups**). If you are working on a standard vehicle instead of a toy car like ours, you will find plenty of examples and references not only within the game but also online.

1) Creating the 3D Model (.dae file) – Blender Phase

back

1) Vehicle creation in Blender

We recommend starting by importing the wheels. This will help you get a sense of the vehicle's overall size and determine the height of its base from the ground. Make sure that the wheels are positioned as close as possible to the zero level to ensure proper alignment with the floor.



You can select a bottom point of the wheel and ensure that its **Z-axis position** is as close to zero as possible. This will help properly align the vehicle with the ground.

Use the reference images to position the wheels correctly and verify the distances between them to ensure they are placed accurately along the **X** and **Y** axes.

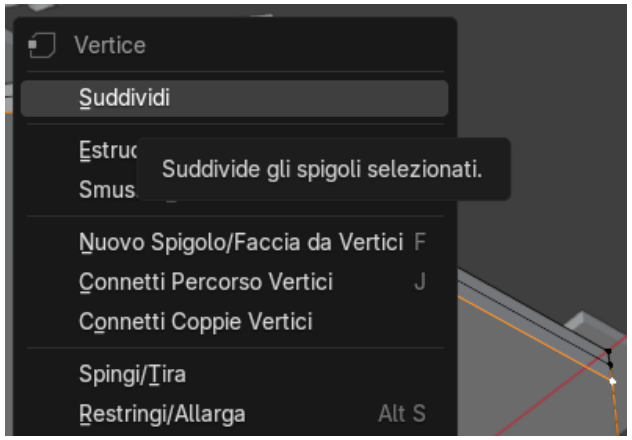
1) Creating the 3D Model (.dae file) – Blender Phase

back

1) Vehicle creation in Blender

Once the wheels are done, we can proceed to create the base of the vehicle. You can use the reference images and the wheels as a guide for placement.
We recommend starting from the base with a simple plane. Align the plane to the desired **Z** level and then deform it to shape the base structure of the vehicle by working with the plane's vertices.

To create a new vertex for your base you can use **subdivide** after selecting 2 vertices:



Then, you can move the newly created vertex anywhere you want and generate new faces or connections by selecting the vertices and pressing **F**.
Another useful tool is the **E** (**Extrude**) function, which allows you to create new vertices or surfaces in the desired direction.

1) Creating the 3D Model (.dae file) – Blender Phase

back

1) Vehicle creation in Blender

To summarize, here are the essential tools for creating the base of the vehicle:

- **Make Edge/Face** (**F** key): Creates a surface or a connection between the selected vertices.
- **Subdivide**: Generates a new vertex between two already selected.
- **Extrude** (**E** key): Extrudes the selected surface or vertices. Hold the **X**, **Y**, or **Z** key to extrude in a specific direction.
- **Merge**: You can merge some vertices or some mesh in a single part, that can be useful to connect vertices from different parts, select the vertices and press the right button of the mouse, to do it select the parts or the vertexes and press the mouse right button, then **Merge**.

After creating the base, carefully check the position of every vertex to ensure symmetry. These tools are useful for modeling vehicle parts, especially rigid and polygonal components. For more complex shapes, start with a simple structure like a cylinder, then use **Duplicate** (**Shift+D**), **Scale** (**S**), and connect the vertexes of the mesh to the others vehicle's part to refine the design. You can also import parts from vanilla vehicles or other files to streamline the process!

1) Creating the 3D Model (.dae file) – Blender Phase

back

1) Vehicle creation in Blender

After creating the wheels and the base, we can proceed with the construction of:

- **Suspension**

To create the suspension, start by modeling the lower arms, followed by the upper parts of the arms. Next, place the coilover into the suspension system and ensure that everything closely resembles the real vehicle.
It's not crucial to replicate the exact details of the real vehicle, but it is important to maintain the correct scale. The components must be positioned and oriented properly to ensure realistic movement within the game.

- **Chassis**

you can create it without worrying too much about the smoothness. We recommend using the same tools described previously to build the structure.
Focus on maintaining symmetry and ensuring all parts are properly connected. You can use the **F** key to join selected elements together.

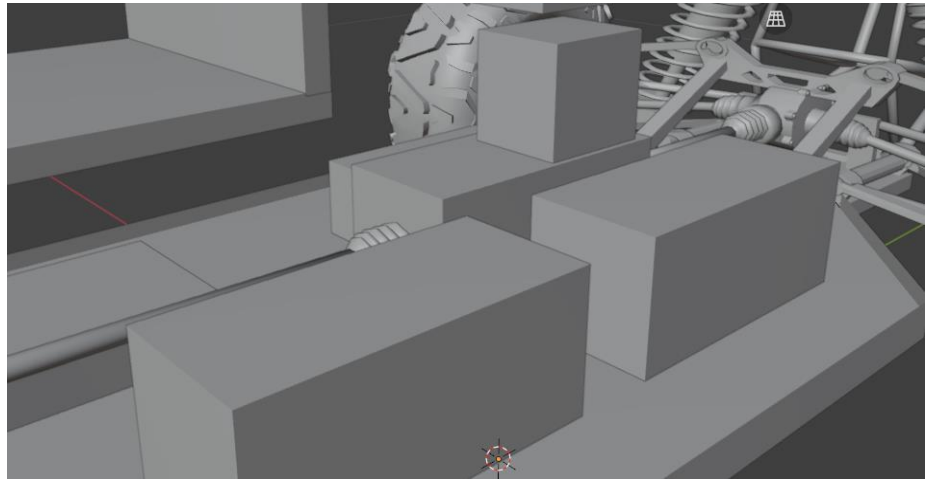
1) Creating the 3D Model (.dae file) – Blender Phase

back

1) Vehicle creation in Blender

- **Engine**

As we will see later in the JBeam creation process, creating a detailed engine mesh is not essential since the **torque** is defined by a function. This means the mesh does not impact the power system. You can even represent the engine using simple cubes, as we did.



- **Transmission**

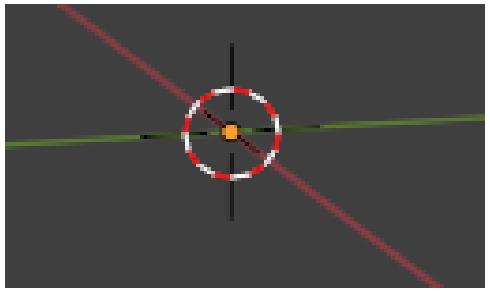
The same principle applies to the **transmission**, but we recommend using a slightly more detailed mesh to make it easier to determine the correct position and orientation of the transmission nodes and beams. You can also use meshes from vanilla vehicles to set up differentials, half-shafts, and driveshafts.

1) Creating the 3D Model (.dae file) – Blender Phase

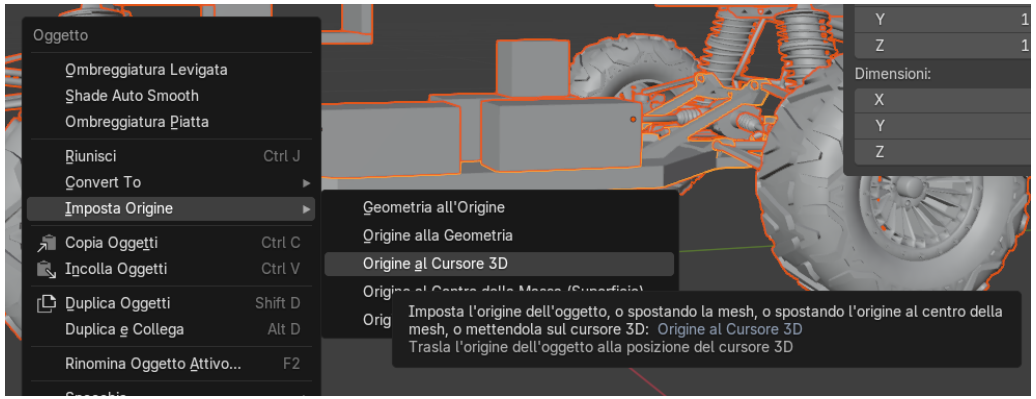
back

1) Vehicle creation in Blender

After creating the vehicle, it's essential to align the origins of all its parts with the world origin. To do this efficiently, we'll use the 3D cursor.



Press **Shift + S** and select "**Cursor to World Origin**". Select all vehicle parts by pressing A, then right-click and choose "**Set Origin**" → "**Origin to 3D Cursor**".



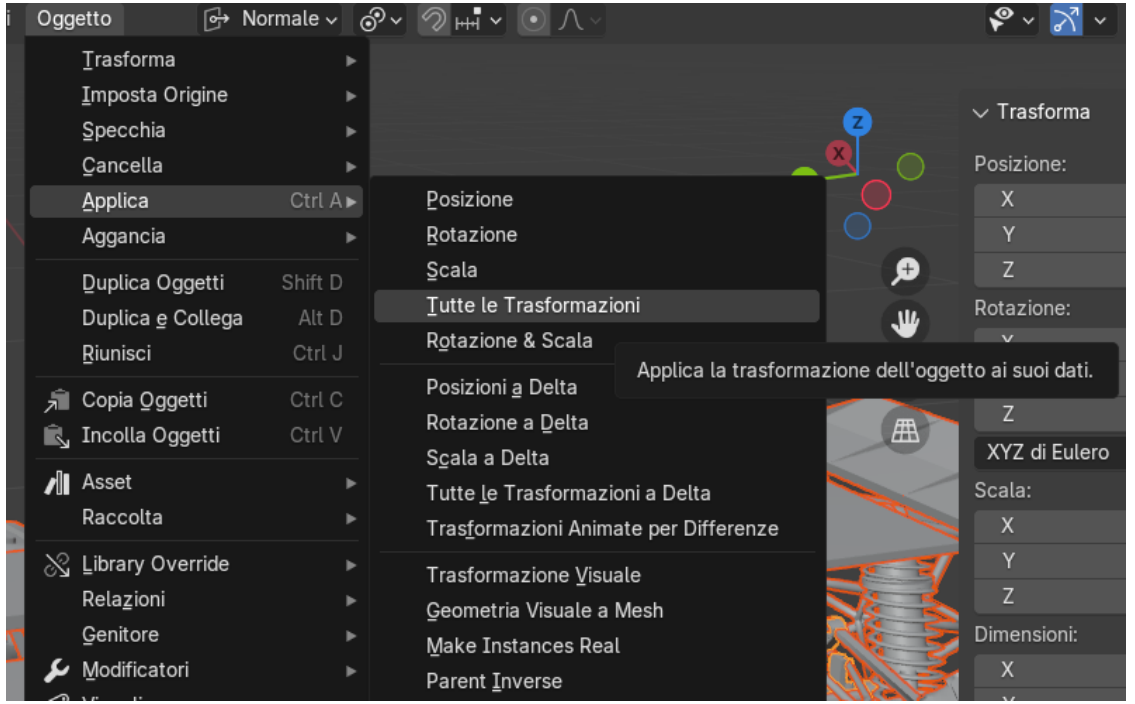
This ensures that every part shares a common origin, allowing BeamNG to correctly interpret the vehicle's position and orientation.

1) Creating the 3D Model (.dae file) – Blender Phase

back

1) Vehicle creation in Blender

We also need to **apply all transforms** (scaling, rotation, etc.) to the model.
This process resets the scaling and rotation values without changing the shape or position of your model.

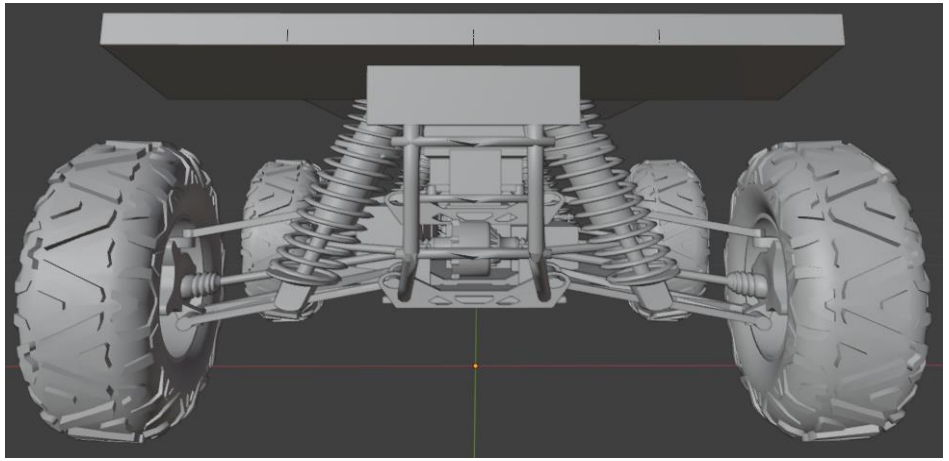


To apply transformations:

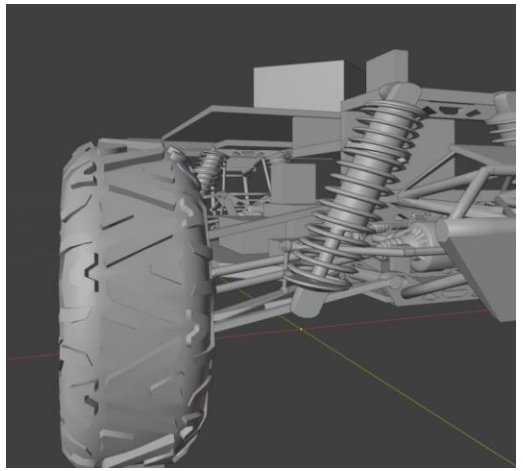
- Select everything by pressing **A**
- Go to the **Object** menu
- Look for "**Apply**" and apply **all transforms**

This ensures that BeamNG interprets your model correctly.

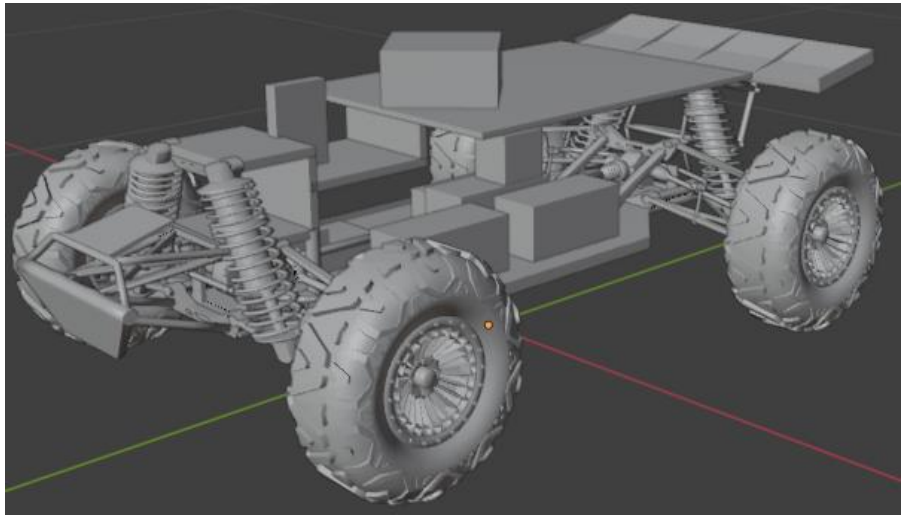
Rear full-body shot



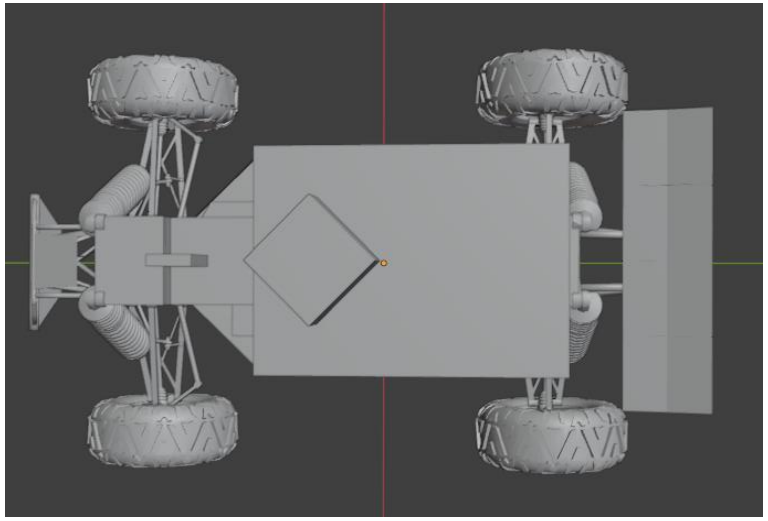
Detailed shot



Full-body shot



Top-down shot



1) Creating the 3D Model (.dae file) – Blender Phase

back

2) Material and UV map assignment in Blender

After creating the vehicle, we're **not done with Blender yet**! We still need to:

- **Assign materials** to all vehicle parts
- **Set up the UV map** for proper texturing

The process for UV mapping and material assignment is the same as we previously explained for track building. We simply need to use **UV Intelligent Projection** and assign a material to each vehicle part.

What name should I give to the materials?
The material names in Blender **must match exactly** with the material names defined in the main.material.json file!

But I don't have a material file! Where can I find it?
No worries—you can use your own file or download the provided main.material.json from the shared drive or you can check for others material files online or in the vanilla vehicles.

1) Creating the 3D Model (.dae file) – Blender Phase

back

3) Creation and use of the main.material file

You have two options:

- **Modify the material names inside main.material** to match the names assigned in Blender.
- **Assign the exact same material names in Blender** as defined in main.material.json

These materials allow you to customize your vehicle with realistic textures :

- **FVee_MainColor** – Main body color with clear coat and metallic properties.
- **FVee_SecondaryColor** – Secondary color with a clear coat and metallic finish.
- **FVee_RollCage** – Material for the roll cage with a generic color mask.
- **FVee_BlackPlastic** – Black plastic material with a slightly rough surface.
- **FVee_BareMetal** – Bare metal texture with a high metallic factor and some roughness.
- **FVee_autobello** – Autobello vehicle material with detailed textures
- **FVee_autobello_engine** – Engine material with high metallic properties and texture maps.
- **FVee_autobello_interior** – Interior material with color, normal, and reflectivity maps.
- **FVee_autobello_interior_b** – Additional interior material with metallic and roughness textures.

1) Creating the 3D Model (.dae file) – Blender Phase

back

3) Creation and use of the main.material file

Now we can finally export the vehicle and use the **.dae** file. Place the **.dae** file in a new folder along with the **main.material.json** file.
The creation process is not entirely linear; at times, you may need to go back and redo certain steps. We recommend organizing your work into **collections**, as explained earlier, and using references during the initial stages of the vehicle's creation.

Now, we have finally completed the **graphics** part of the vehicle.

Wait, are you saying this is only for the graphics?

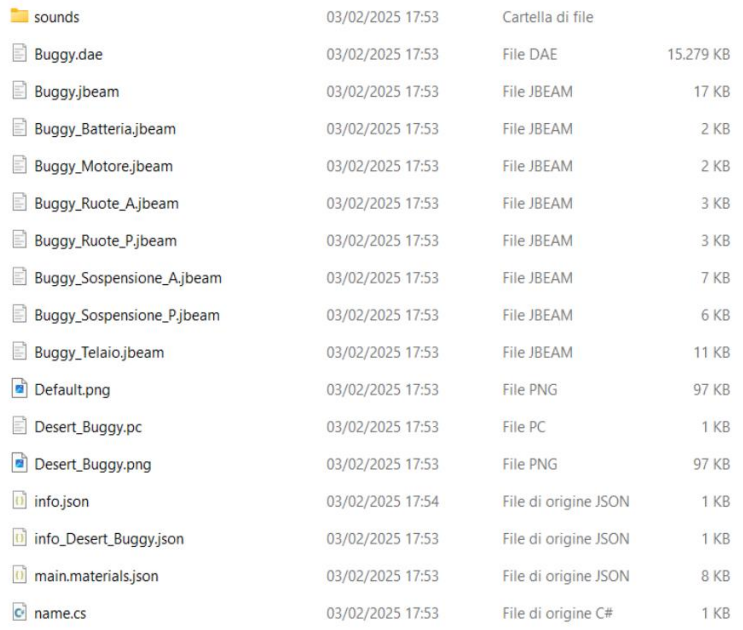
Well, yes. Having a good **3D model** is essential for a realistic appearance, but most of the **physics** are handled by the **JBeam files**.

2) Creating the JBeam Scripts – JBeam Phase

back

1) Hierarchy of the JBeam structure

The **JBeam files** define the **soft-body physics** of the vehicle but do not represent its exact configuration. Instead, they are divided into different components. However, the game automatically compiles all JBeam files into a **single structure**, so you don't need to worry about managing multiple files. Any **node** or **beam** defined in one file will be recognized across **all files**!



sounds	03/02/2025 17:53	Cartella di file	
Buggy.dae	03/02/2025 17:53	File DAE	15.279 KB
Buggy.jbeam	03/02/2025 17:53	File JBEAM	17 KB
Buggy_Batteria.jbeam	03/02/2025 17:53	File JBEAM	2 KB
Buggy_Motore.jbeam	03/02/2025 17:53	File JBEAM	2 KB
Buggy_Ruote_A.jbeam	03/02/2025 17:53	File JBEAM	3 KB
Buggy_Ruote_P.jbeam	03/02/2025 17:53	File JBEAM	3 KB
Buggy_Sospensione_A.jbeam	03/02/2025 17:53	File JBEAM	7 KB
Buggy_Sospensione_P.jbeam	03/02/2025 17:53	File JBEAM	6 KB
Buggy_Telaio.jbeam	03/02/2025 17:53	File JBEAM	11 KB
Default.png	03/02/2025 17:53	File PNG	97 KB
Desert_Buggy.pc	03/02/2025 17:53	File PC	1 KB
Desert_Buggy.png	03/02/2025 17:53	File PNG	97 KB
info.json	03/02/2025 17:54	File di origine JSON	1 KB
info_Desert_Buggy.json	03/02/2025 17:53	File di origine JSON	1 KB
main.materials.json	03/02/2025 17:53	File di origine JSON	8 KB
name.cs	03/02/2025 17:53	File di origine C#	1 KB

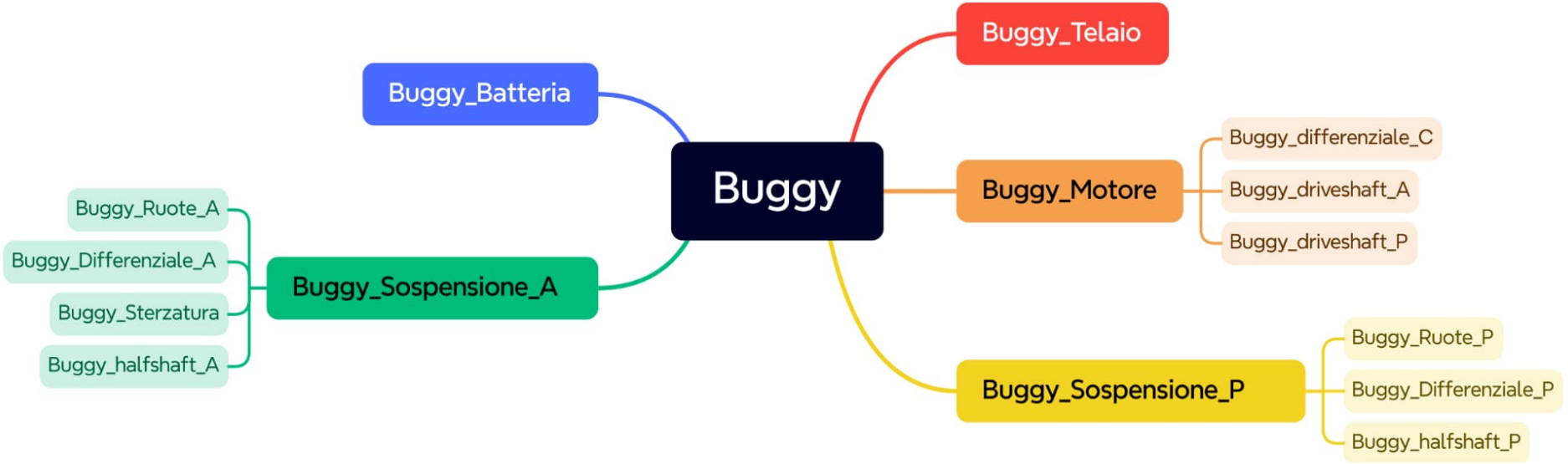
- **Buggy.jbeam**: main jbeam file and base definition.
- **Buggy_Telaio.jbeam**: defines the chassis.
- **Buggy_Motore.jbeam**: represents the engine.
- **Buggy_Batteria.jbeam**: represents the battery.
- **Buggy_Ruote_A.jbeam**: defines the front wheels.
- **Buggy_Ruote_P.jbeam**: defines the rear wheels.
- **Buggy_Sospensione_A.jbeam**: defines front suspension.
- **Buggy_Sospensione_P.jbeam**: defines rear suspension.

2) Creating the JBeam Scripts – JBeam Phase

back

1) Hierarchy of the JBeam structure

The first step in creating the **JBeam structure** is to define the **hierarchy** of the vehicle's components. Below, you can see a map of our vehicle. These are not **JBeam files** themselves but rather representations of the **structures** defined within those files and how they are **connected** to each other.
Inside the files, you need to define this structure, and by using the "**slot**" function, you can establish **parent-child** connections between components.



2) Creating the JBeam Scripts – JBeam Phase

back

1) Hierarchy of the JBeam structure

For example:

{"Buggy":{
...

"slots": [
["type", "default", "description"]
["Buggy_Telaio","Buggy_Telaio","Buggy Telaio"],
["Buggy_Motore","Buggy_Motore","Buggy Motore"],
["Buggy_Batteria","Buggy_Batteria","Buggy Batteria"],
["Buggy_Sospensione_A","Buggy_Sospensione_A","Buggy Sospensione Anteriore"],
["Buggy_Sospensione_P","Buggy_Sospensione_P","Buggy Sospensione Posteriore"],
],
...

The **JBeam structure** we created is **simple** and based on **our** vehicle, but you will need to **design the structure specifically for your own vehicle**. You can also take a look at **other mods** or **vanilla vehicles** for reference. There is no single **correct answer**, only different **types of structures**, but make sure to **organize everything properly** for the best results. Also, you can follow the dae collections used in Blender.

2) Creating the JBeam Scripts – JBeam Phase

back

2) Definition of beams and nodes

After defining the **hierarchy**, we can start creating the **nodes and beams** structure. We recommend starting from the **base of the vehicle**, but, as before, it's a good idea to take inspiration and use as a base of work **other mods** or **vanilla vehicles**. Our vehicle was originally based on the **RG-R/C mod** (a toy car mod), but we made several modifications:

- **We reduced** the number of **beams and nodes** used.
- **We completely changed** the **suspension** and **chassis** structure.
- **The placement of the nodes and the beam usage was completely overhauled**.
- **We redesigned the entire powertrain**.
- **We created a new triangle structure and adjusted the vehicle's dynamics**.

For this reason, we **don't recommend starting from scratch**. Instead, take an **existing base** and **adapt the structure** to fit your vehicle. Once the vehicle works properly, you can start **adding features** or **tweaking beams and nodes** to refine the handling.

2) Creating the JBeam Scripts – JBeam Phase

back

2) Definition of beams and nodes

The **mass of the nodes** and the **beam options** are not easy to define at first. Instead, you will refine these values through a **tuning process**, which consists of small adjustments and testing different variations.

This is why it's so important to **start from a solid base**. Creating everything from scratch is possible, but it can take a significant amount of time. Some **vanilla vehicles** have been developed and improved over **several years (but some of them has problem like the not perfect symmetry)**!
With **mods**, the situation is different. Some mods are built **from zero** or from a **very minimal base**, like the **RG-R/C mod**. However, if you examine its **beam and node structure**, you will likely notice **errors**, such as **missing beams, misplaced nodes**, or **unrealistic values** (for example, the toy car in this mod weighs over **240 kg**, which is far from the real-life counterpart).
Before choosing a **base for your work**, be aware of its **limitations** and decide on the **level of realism** you want to achieve.

2) Creating the JBeam Scripts – JBeam Phase

back

2) Definition of beams and nodes

If you're reproducing a **normal car**, this task will be easier due to the vast number of **mods** and **vanilla vehicles** available for reference.
However, if you're recreating a **toy car**, you shouldn't waste time checking mods or vanilla vehicles. We suggest using **your own vehicle** as a base (it's also one of the lightest vehicles in the world).

The most important aspect of the **nodes** and **beams structure** is **symmetry**. You must ensure that the beam structure is the same on both sides, and that the nodes are placed symmetrically.
If symmetry is not maintained, the vehicle will behave incorrectly in the game, causing it to veer off course due to improper weight distribution and forces. This issue becomes more noticeable as the vehicle's speed increases.

Now we can start explain how we setup the nodes and beam structure:

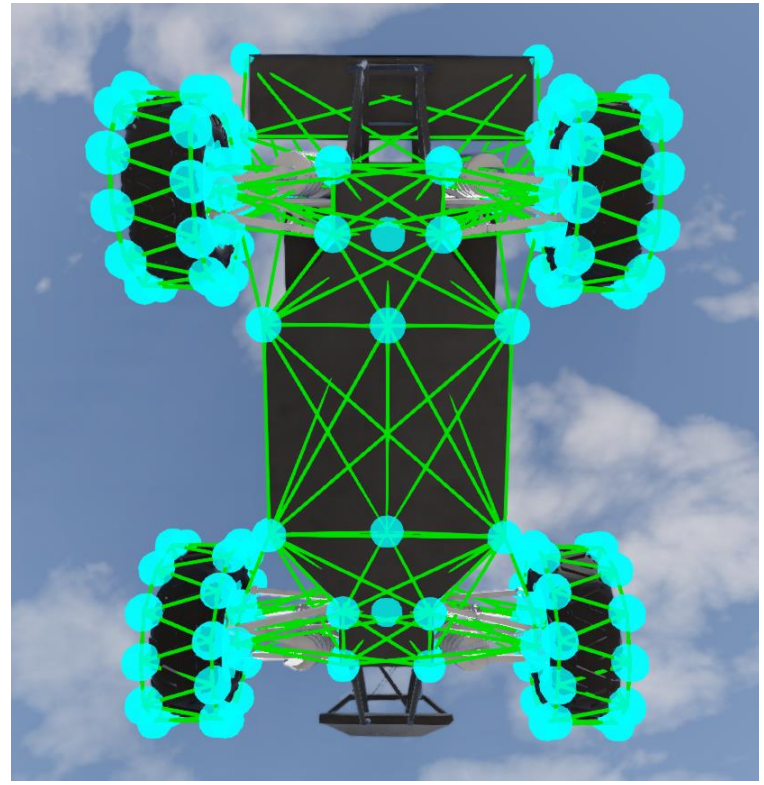
2) Creating the JBeam Scripts – JBeam Phase

back

2) Definition of beams and nodes

- **Buggy**

In the **main file**, the **base of the vehicle** is defined. This is a crucial component because it needs to be **very rigid** but **not excessively** so to maintain **stability**.
Additionally, it is essential to ensure a **symmetrical structure** to prevent **unexpected behavior** in-game.



As shown in the image, the **masses** are **evenly distributed** across the entire base surface, providing **stability** and **realistic driving dynamics**.

By using the Jbeam distribution property, some nodes of the base are be moved in other structures to define engine, differentials and other components with nodes and beams.

2) Creating the JBeam Scripts – JBeam Phase

back

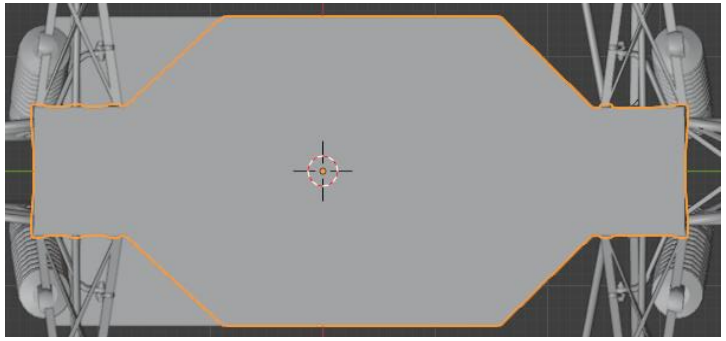
2) Definition of beams and nodes

- **Buggy**

To create the base, you can follow the real vehicle's structure and place the nodes in the exact positions of the Blender model's vertices.
We suggest creating three rows of nodes:

- **Central nodes**, named like fr1, fr2, …
- **Left nodes**, named like fr1l, fr2l, …
- **Right nodes**, named like fr1r, fr2r, …

After creating the nodes, we can connect each adjacent node with beams. Since there are no upper nodes yet to support the lower part of the vehicle, the nodes will tend to collapse and be unstable when tested in-game.



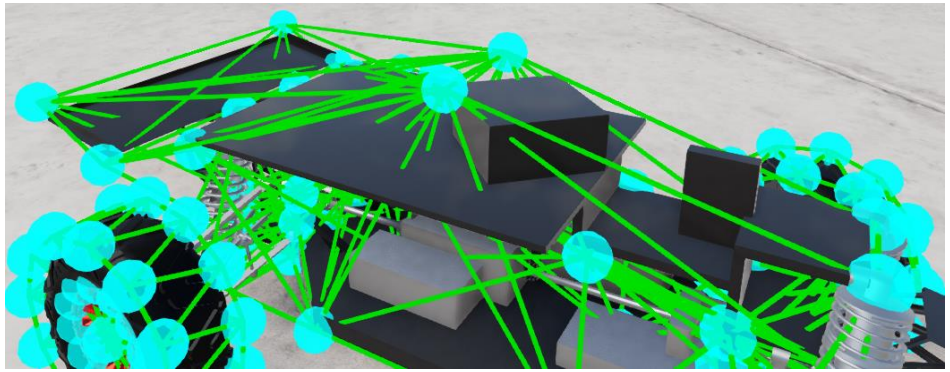
2) Creating the JBeam Scripts – JBeam Phase

back

2) Definition of beams and nodes

- **Buggy_Telaio**

In the upper part of the vehicle, we use **two upper nodes** (**fr4ur** and **fr4ul**) to support the entire chassis structure and the rear wing. Additionally, we use **four nodes** (**fr4t, fr3tl, and fr3tr**) to connect the rear and front sections of the vehicle.
These nodes are crucial, as their **stability** and ability to **absorb forces** generated by the wheel movement directly affect the overall stability and behavior of the vehicle.



The **rear wing** consists of **four nodes** forming a square. It is connected to both the **rear suspension** and the **upper nodes** (**fr4ur** and **fr4ul**).

2) Creating the JBeam Scripts – JBeam Phase

back

2) Definition of beams and nodes

- **Buggy_Sospensione_A**

The suspension consists of several nodes:

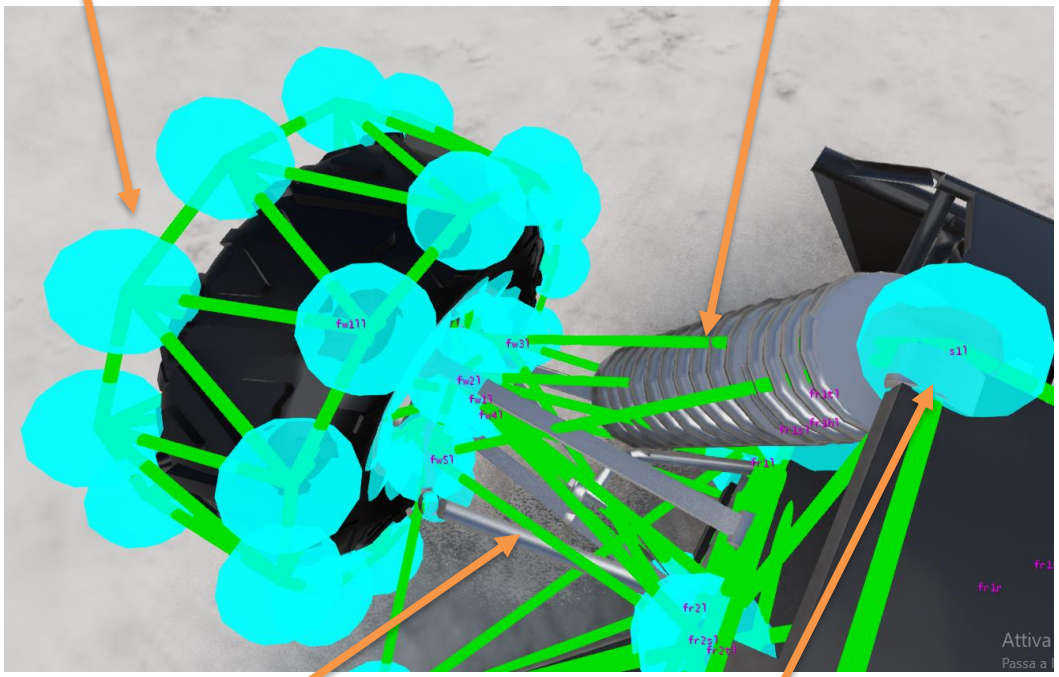
- **6 nodes per wheel**
- **1 node for the coilover**
- **7 nodes for the arm structure**

The most important beams are the **two coilover beams**. They connect the **coilover node** to **two nodes on the wheel**. These beams are crucial because their **values** directly affect the vehicle's dynamics.

All the wheels of the vehicle are done like that to follow the symmetry rule. We used some "variable" to modify in game some suspension value.

The tires are defined with the function: pressurewheel

Coilover beams: characterised by stiffness, damping, and ride heigh



Steering beam

coilover node

2) Creating the JBeam Scripts – JBeam Phase

back

2) Definition of beams and nodes

- **Buggy_Sospensione_A**

In the game, you can modify various suspension properties of the vehicle through the **"Vehicle Config"** menu. Adjusting these settings allows you to change the vehicle's dynamics and overall performance.

- **Increasing stiffness** improves vehicle stability and handling but reduces its ability to navigate uneven terrain.
- **Increasing damping** makes the vehicle less stable but allows for smoother suspension dynamics.
- **Raising the ride height** helps the vehicle traverse certain terrains more easily but reduces overall performance.

2) Creating the JBeam Scripts – JBeam Phase

back

2) Definition of beams and nodes

- **Buggy_Sospensione_A**

In the Jbeam files the coilovers beams are described by the following lines of

```
// Molle-Ammortizzatori Anteriore
{"beamSpring": "$spring_F", "beamDamp": "$damp_bump_F"},
{"beamDeform": "FLT_MAX", "beamStrength": "FLT_MAX"},
{"beamPrecompression": "$rideheight_F", "beamType": "|BOUNDED", "beamLongBound": 1.0, "beamShortBound": 1.0},

["s1r", "fw5r", {"beamDampRebound": "$damp_rebound_F", "soundFile": "art/sound/spring_compress2.ogg", "volumeFactor": 1.8,
["s1r", "fw3r", {"beamDampRebound": "$damp_rebound_F", "soundFile": "art/sound/spring_compress2.ogg", "volumeFactor": 1.8,
["s1l", "fw5l", {"beamDampRebound": "$damp_rebound_F"}],
["s1l", "fw3l", {"beamDampRebound": "$damp_rebound_F"}],
```

- **Spring & Damping Settings:** use variables to adjust stiffness and bump damping
- **Structural Properties:** the beams are indestructible and can be adjust the ride height
- **Connections:** Defines the **shock absorber beams** between suspension (`s1r`, `s1l`) and wheel nodes (fw5r, fw3r, etc.).

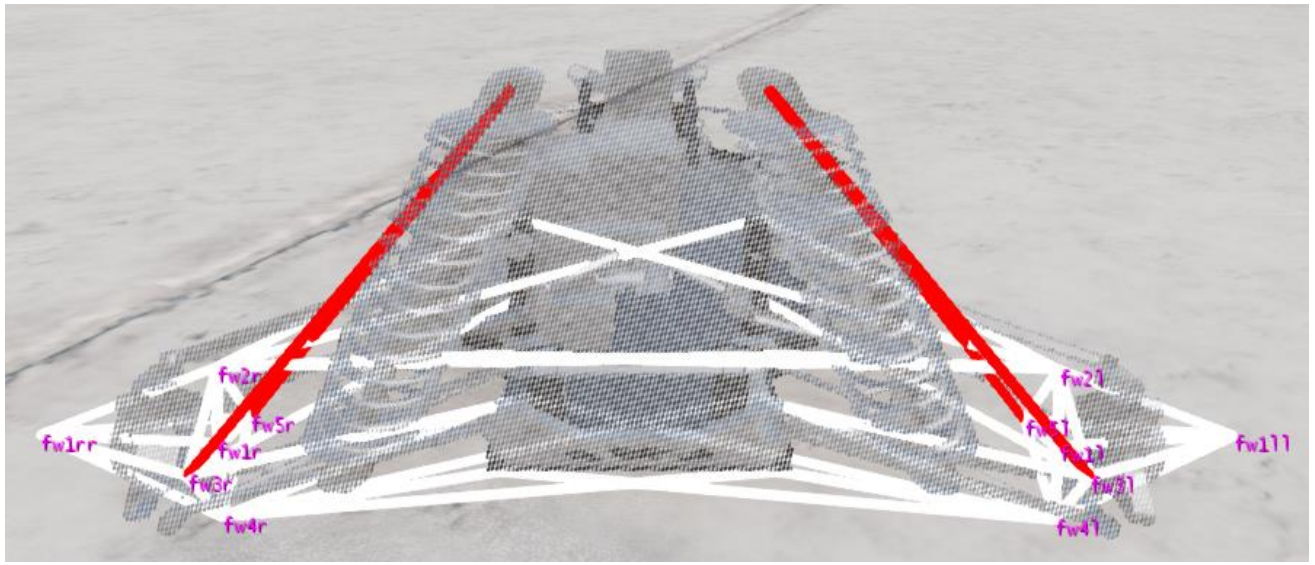
2) Creating the JBeam Scripts – JBeam Phase

back

2) Definition of beams and nodes

- **Buggy_Sospensione_A**

By using the debug tool of the game, we can show to you the suspension's beam structure and how the wheels nodes are defined:



There are also low force beams to make the structure more stable

```
{"beamSpring": 500000, "beamDamp": 1000},
{"beamDeform": "FLT_MAX", "beamStrength": "FLT_MAX"}
{"beamPrecompression": 1.0, "beamType": "|BOUNDED",

["fr1r", "fw4l"],
["fr2r", "fw4l"],
["fr1tr", "fw2l"],
["fr2tr", "fw2l"],
["fr1l", "fw4r"],
["fr2l", "fw4r"],
["fr1tl", "fw2r"],
["fr2tl", "fw2r"]
```

We can see in red the coilover beams defined in the previous slide and in white the beams structure to connect the wheels to vehicle's body.

2) Creating the JBeam Scripts – JBeam Phase

back

2) Definition of beams and nodes

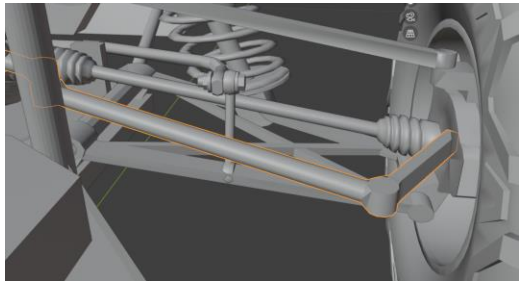
- **Buggy_Sospensione_A**

Make sure that the nodes used for the wheels are very stable to withstand the forces acting on them during different driving phases.
You can also use the rows of nodes to structure the suspension arms, modeling them as beams with **low damping** but **high strength and stiffness** to accurately simulate the real vehicle's structure.

```
"beams": [
  {"beamSpring": 9000000, "beamDamp": 1},
  {"beamDeform": "FLT_MAX", "beamStrength": "FLT_MAX"},
  {"beamPrecompression": 1.0, "beamType": "|NORMAL", "beamLongBound": 1.0, "beamShortBound": 1.0},
```

Additionally, ensure that the entire structure is **symmetrical** between the left and right sides of the vehicle, while keeping the necessary beams free for the steering mechanism of the front suspensions.

We decide to use the beam ["fr2sr", "fw5r"] for the right steering part and the beam ["fr2sl", "fw5l"] for the left steering part.



2) Creating the JBeam Scripts – JBeam Phase

back

3) Definition of flexbodies and refnodes

Now that we have completed the **nodes and beams definition**, we need to create the **connection between the nodes, beams, and the mesh** from the **.dae files**.
To achieve this, we will use the "**flexbodies**" function, which allows us to link a **mesh** to a **group of nodes and beams**. The smallest group that can define a mesh is a **single beam**, as seen in the case of **halfshafts**.
There isn't much to **create from scratch** here—what matters is following the **base structure** and using a **tuning process** to determine the best **node and beam groups** to assign.
Additionally, we can rely on the **hierarchy structure** established earlier. For example, the **suspension arms** are connected to the **wheel hubs**, meaning they must move together.
However, they also need to follow the **vehicle's base**, so some **arm nodes** must also be linked to the base structure.

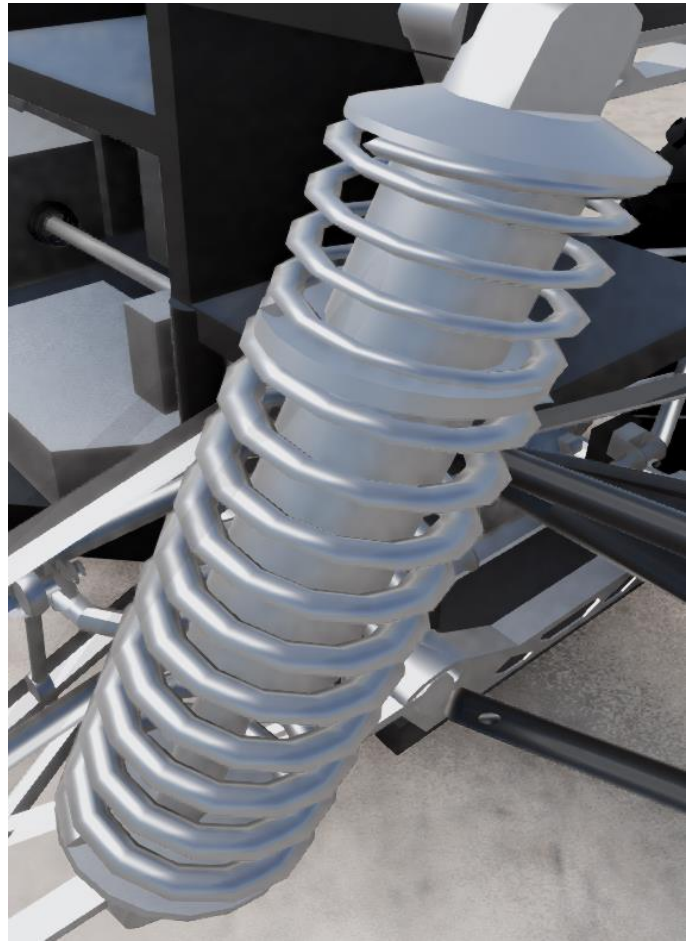
```
  [ "molla_AD" , [ "molla_AD" ]],
  ["braccio_sterzo_AS", ["braccio_AS", "hub_AS"]],
  ["braccio_sterzo_AD", ["braccio_AD", "hub_AD"]]
],
```

2) Creating the JBeam Scripts – JBeam Phase

back

3) Definition of flexbodies and refnodes

With some work and by following the base structure and hierarchy you've set up, you will be able to correctly configure the meshes. Remember: if a mesh doesn't spawn, it's because you haven't assigned the mesh group to the correct nodes!



For example, the coilover AD is generated by the **group** "molla_AD", but if you don't assign this group to both the coilover node and the wheel nodes connected by the coilover beam, it won't spawn!

Also, remember that the game retains the **axis origin** from Blender and its **position relative to the origin but the beams and nodes group associated define the behavior of the mesh.**
It's not essential for the **node groups** to perfectly match the **mesh structure**—their purpose is simply to **spawn the mesh** and **control its behavior**.

2) Creating the JBeam Scripts – JBeam Phase

back

3) Definition of flexbodies and refnodes

- **Refnodes**

refnodes are essential for spawning the vehicle and ensuring proper control.
You need to define the refnodes in the main file (Buggy.jbeam) and use very stable nodes to guarantee stable handling and driving behavior.
It's best to assign refnodes using the base of the vehicle as a reference:

- **Ref node:** Ensure it is at the center of the vehicle and near the axis origin set in Blender. Example: (0.0, 0.0, height of the base)
- **Back node:** Positioned behind the ref node.
- **Left node:** Positioned to the left of the ref node.
- **Top node:** Positioned above the ref node.

```
"refNodes":[
    ["ref:", "back:", "left:", "up:"]
    ["fr4", "fr5", "fr4l", "fr4t"]
```

There are additional optional nodes, but they are not necessary for our vehicle.

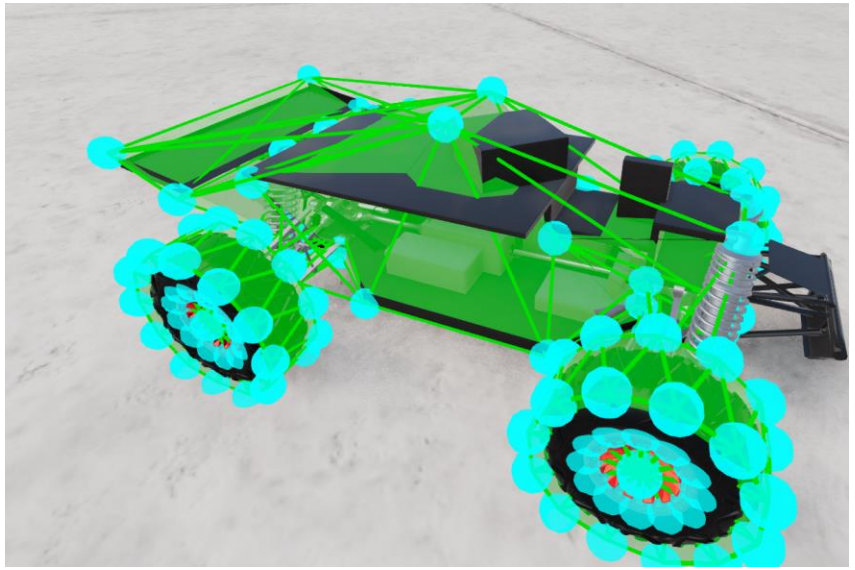
2) Creating the JBeam Scripts – JBeam Phase

back

3) Definition of flexbodies and refnodes

- **Triangles**

After defining the **nodes and beams**, it is crucial to set up **triangles** across the entire surface of the vehicle to ensure proper **collision behavior**.
It is very important that the **green side** of the triangles has its **normal vector facing outward** to ensure correct impact reactions.



As shown in the image, you **don't need to perfectly match** the exact **mesh definition**—a simplified structure is sufficient for proper physics simulation.

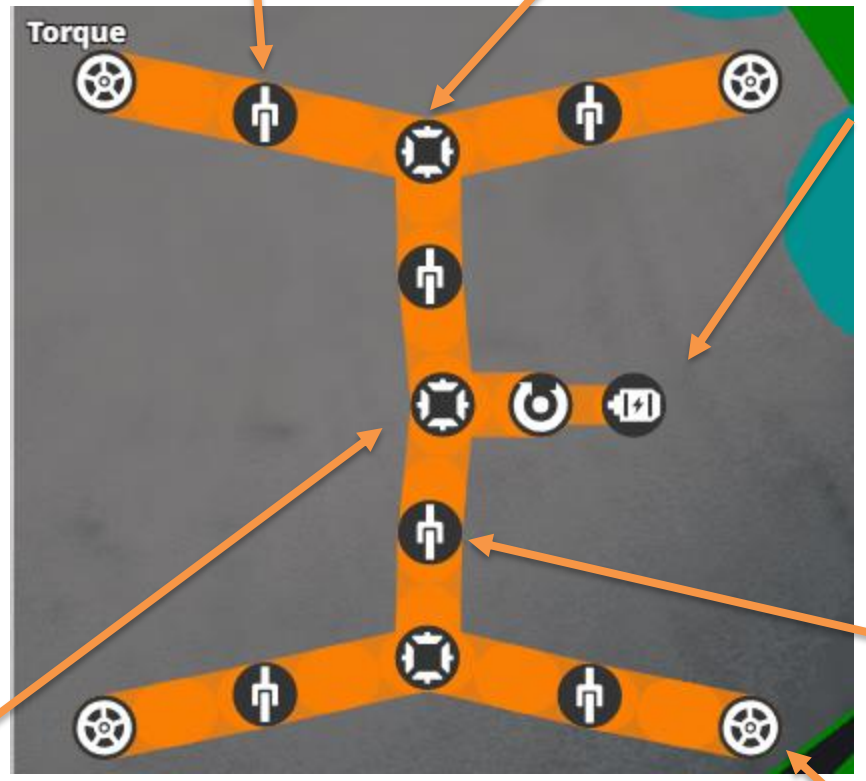
2) Creating the JBeam Scripts – JBeam Phase

back

4) Powertrain structure and vehicleController

The powertrain system is what makes the vehicle move—you need to define all the **components** of your powertrain system.
The real vehicle is a **4x4** with a **single electric motor**, **three differentials**, and **multiple shafts** connecting everything together.
We successfully replicated the entire **powertrain structure** of the real vehicle, as well as its **performance**.

You can create every powertrain possible, from the camion with so many wheels or a racing rear traction car.



2) Creating the JBeam Scripts – JBeam Phase

back

4) Powertrain structure and vehicleController

For the own vehicle we decide to use:

- **Electric Motor**
- **Reactor Torque**
- **3 Open Differentials** with the gear-ratio modifiable
- **Electric Battery** with level energy modifiable
- **2 driveshaft**
- **4 halfshaft** destructible
- **4 wheels**

```
"powertrain": [
  ["type", "name", "inputName", "inputIndex"],
  ["electricMotor", "engine", "dummy", 0]
],
```

```
"powertrain": [
  ["type", "name", "inputName", "inputIndex"],
  ["differential", "differenziale_anteriore", "driveshaft_A", 1, {"diffType":"open",
],
```

To define the electric motor, you need to define the torque in every RPM range. Through a tuning process, we succeeded in setting the right values for the vehicle. Increasing the gear ratio will improve acceleration but reduce top speed.
The battery is also configured to have realistic durability and allows the user to develop algorithms to optimize energy consumption for specific tasks.

2) Creating the JBeam Scripts – JBeam Phase

back

4) Powertrain structure and vehicleController

The electric motor is defined by the Jbeam section of the engine:

- **Torque Curve:** Specifies torque output at different RPMs. The motor provides **30 Nm** up to **9000 RPM**, then gradually decreases.
- **Max RPM:** The motor's redline is set at **13,000 RPM**.
- **Inertia & Friction:** Defines how quickly the motor responds and resists motion.
- **Electrical Efficiency:** Set to **95%**, affecting power consumption.
- **Regeneration:** Disabled (maxRegenTorque: 0.0).
- **Battery:** The motor uses "mainBattery" as its power source.

```
"engine": {
  //Engine configuration
  "torque": [
    ["rpm", "torque"],
    [0, 30],
    [500, 30],
    [1000, 30],
    [1500, 30],
    [2000, 30],
    [3000, 30],
    [4000, 30],
    [5000, 30],
    [6000, 30],
    [7000, 30],
    [8000, 30],
    [9000, 30],
    [10000, 20],
    [11000, 20],
    [12000, 20],
    [13000, 20],
    [14000, 10],
    ],


  "maxRPM":13000,
  "inertia":0.10,
  "friction":5,
  "dynamicFriction":0.001,
  "electricalEfficiency":0.95,
  "regenThrottle": 0.0,
  "maxRegenTorque": 0.0,
```

2) Creating the JBeam Scripts – JBeam Phase

back

4) Powertrain structure and vehicleController

We can define the powertrain **chain scheme** shown earlier in the following way:

```
["type", "name", "inputName", "inputIndex"],
["electricMotor", "engine", "dummy", 0]
```

```
["type", "name", "inputName", "inputIndex"],
["torsionReactor", "torsionReactor", "engine", 1],
["differential", "differenziale_centrale", "torsionReactor", 1,
```

```
["type", "name", "inputName", "inputIndex"],
["shaft", "driveshaft_A", "differenziale_centrale", 1,
```

```
["type", "name", "inputName", "inputIndex"],
["differential", "differenziale_anteriore", "driveshaft_A", 1,
```

```
["type", "name", "inputName", "inputIndex"],
["shaft", "halfshaft_AS", "differenziale_anteriore", 1,
["shaft", "halfshaft_AD", "differenziale_anteriore", 2,
```

The powertrain **section** must be located in every **part that needs to be linked with the chain**

```
"torqueReactionNodes:":["fr4l","fr4r","fr4"],
```

These are the **reference nodes** used to apply torque reaction forces. Choosing the right nodes ensures **realistic chassis flex and behavior**.

2) Creating the JBeam Scripts – JBeam Phase

back

5) Definition of wheels and hydros

After defining the Nodes and Beams and the powertrain, we need to set up the wheels and steering of the vehicle. Unlike other nodes and beams, the wheels are created using the BeamNG function "**pressureWheel**".
We can divide the wheel characteristics into six modifiable sections:

- **General settings**
- **Hub options**
- **Tire options**
- **Pressure options**
- **Brake options**
- **Definition and connection to the hub's nodes**

There are many possible values to modify, so this is just one way to categorize them. You can create your own classification and adjust more characteristics beyond the default values.

2) Creating the JBeam Scripts – JBeam Phase

back

5) Definition of wheels and hydros

In the first three sections, you must define the height and dimensions of the wheel hub, as well as the characteristics of the tire rubber.
The spring stiffness of the hub's beams and the tire's beams can be adjusted through a tuning process. We recommend starting with a low value and gradually increasing it step by step. This approach helps to avoid instability during the testing phase.

```
//general settings
{"radius":0.1000}, //Radius of the tire nodes
{"hubRadius":0.0515}, //Radius of the rim
{"wheelOffset":-0.01}, //Offset from the original position(Left/right)
{"hubWidth":0.04}, //Width of the rim
{"tireWidth":0.0752}, //Width of the tire
{"numRays":12}, //The amount of nodes to make the circle, more may result in smoother driving, but at the cost of performance, weight & stability

//hub options
{"hubBeamSpring":1200000, "hubBeamDamp":125},
{"beamSpring":1200000, "beamDamp":125},
{"hubBeamDeform":"FLT_MAX", "hubBeamStrength":"FLT_MAX"},
{"beamDeform":"FLT_MAX","beamStrength":"FLT_MAX"},
{"hubNodeWeight":0.20}, //Weight of the rim nodes, to get the total weight of the rim, calculate numRays X nodeWeight X 2
{"hubNodeMaterial":"|NM_PLASTIC"},
{"hubFrictionCoef":0.2},

//tire options
{"wheelTreadBeamSpring":180000,"wheelTreadBeamDamp":50,"wheelTreadBeamDeform":"FLT_MAX","wheelTreadBeamStrength":10000000},
{"wheelPeripheryBeamSpring":180000,"wheelPeripheryBeamDamp":40,"wheelPeripheryBeamDeform":"FLT_MAX","wheelPeripheryBeamStrength":10000000},
{"springExpansion":180000, "dampExpansion":18},
{"nodeWeight":0.1},
{"nodeMaterial":"|NM_RUBBER"},
{"frictionCoef":1.3}, //Tire friction for each node, increasing will add more grip to your vehicle
```

2) Creating the JBeam Scripts – JBeam Phase

back

5) Definition of wheels and hydros

Then, you must define the pressure values and brake characteristics. Our vehicle's braking system is almost entirely disabled since it does not have a braking system on the wheels. The wheel definition is the final part of the file, where you must specify the node references for the wheel:

- **AD**: Wheel name
- **ruota_AS**: Group for flexbodies
- **fw1rr**: Most lateral node reference
- **fw1r**: Less lateral node reference
- **fw4r**: Turning node reference
- **fw3r**: Braking node reference
- **-1/1**: Wheel rotation direction

The wheel name must be used in the powertrain system with the function "connectedWheel": "AS".

```
//Pressure
{"pressurePSI":10}, //Tire pressure in PSI
{"maxPressurePSI":1500},
{"reinforcementPressurePSI":23},
{"pressureSpring":360100},
{"reinforcementPressureSpring":2044000},
{"pressureDamp":50},
{"reinforcementPressureDamp":53},

//Brake
{"brakeTorque":10},
{"parkingTorque":0},
{"enableBrakeThermals":false},
{"brakeDiameter":false},
{"brakeMass":false},
{"brakeType":false},
{"rotorMaterial":false},
{"brakeVentingCoef":false},

//front
{"propulsed":0}
{"selfCollision":false}
{"collision":true}
{"axleBeams":["axle_AD"]}
["AD", "ruota_AD", "", "fw1rr", "fw1r", 9999,  "fw3r", 1],
{"axleBeams":["axle_AS"]}
["AS", "ruota_AS", "", "fw1ll", "fw1l", 9999,  "fw3l",-1],
{"axleBeams":[]}
{"enableABS":false}
{"selfCollision":true}
```

2) Creating the JBeam Scripts – JBeam Phase

back

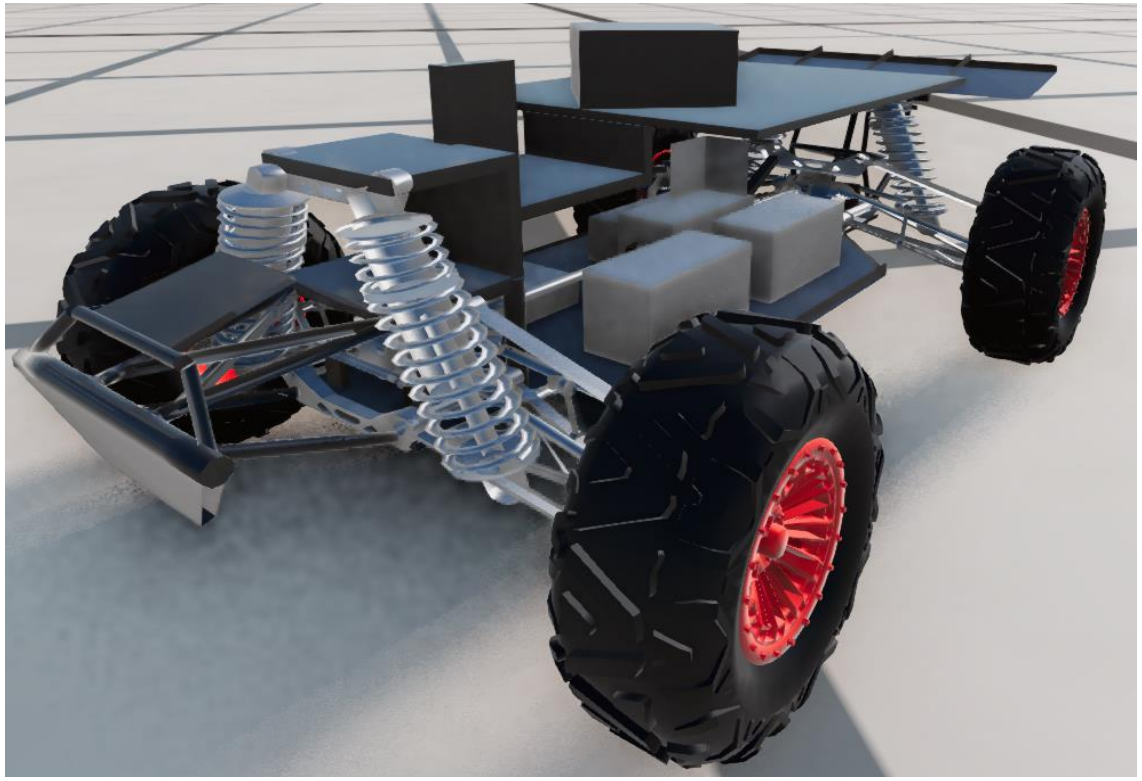
5) Definition of wheels and hydros

Now we need to define how the vehicle's cornering works. In our project, inside the Suspension file, you will find the steering configuration.
This sets up the steering system for the buggy, with specific parameters that define the responsiveness and limits of the steering mechanism, including steering wheel lock, input/output speed, and sensitivity adjustments.
The "hydros" section defines the connections for the steering system using nodes. Each link includes:

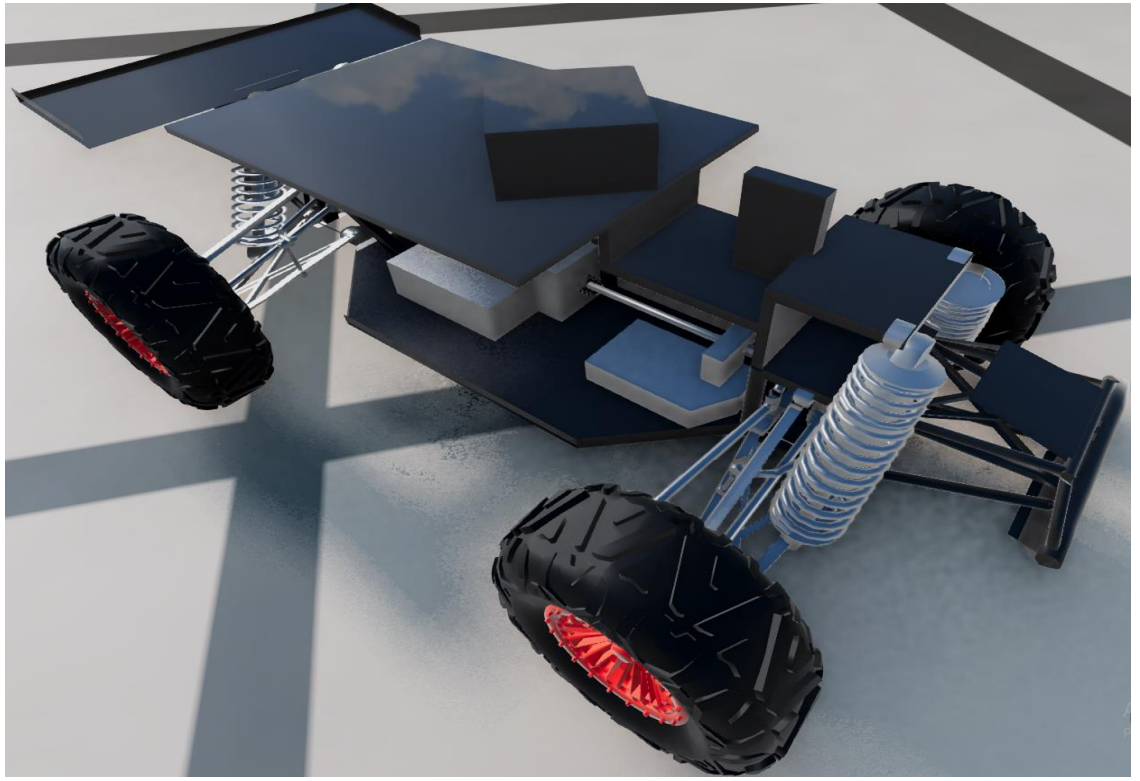
- "**id1" and "id2**": Node identifiers for the steering beams.
- "**factor**": Adjusts steering sensitivity, with negative values reversing the steering direction.
- "**steeringWheelLock**": Limits the steering wheel's rotation.
- "**lockDegrees**": Sets the maximum steering angle.
- "**inRate" and "outRate**": Control the speed of steering input and return to neutral.

2) Creating the JBeam Scripts – JBeam Phase

Full-body shot



Top-down shot



2) Creating the JBeam Scripts – JBeam Phase

Vehicle's Dynamic

3) Defining the Package Structure – Configurations Phase

back

1) Packages and Vehicles

After defining the nodes and beams structure and completing the JBeam construction, we need to create and configure additional files to provide BeamNG with essential information about the vehicle package and configurations.

A mod is not merely a single vehicle but rather a package that defines the vehicle within it. The package establishes the overall structure, including the number of components that make up the vehicles it contains and the hierarchical relationships between these parts.
Within the package, multiple versions of a vehicle can be defined by modifying individual components or adjusting specific variables.

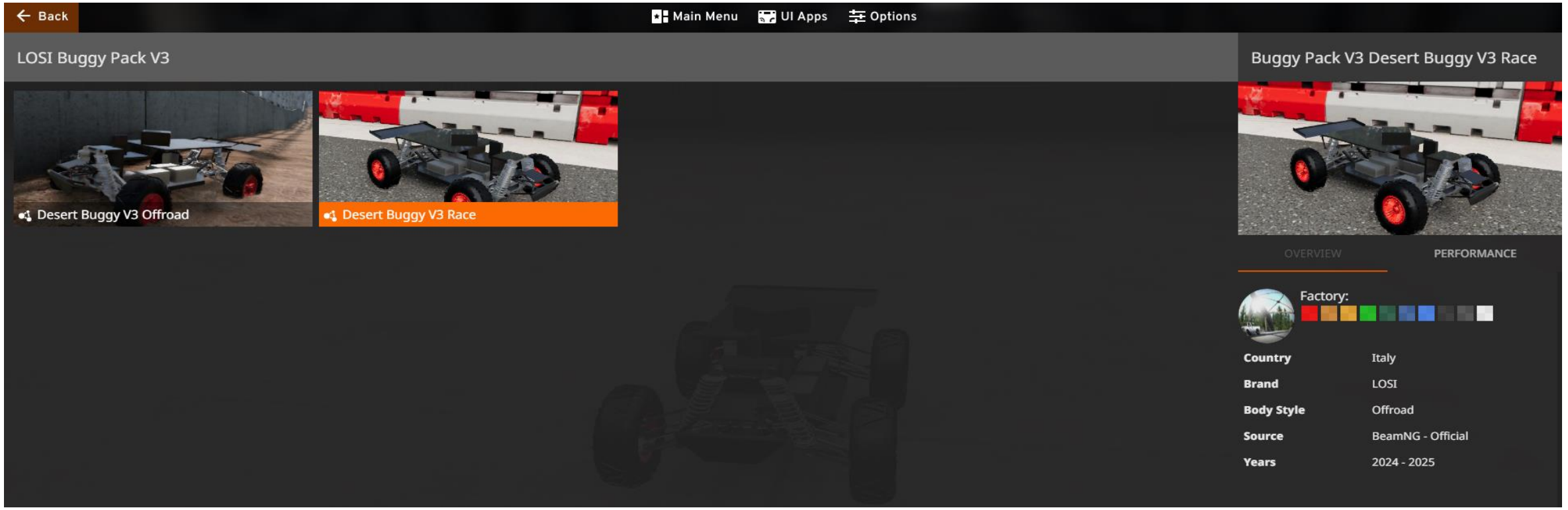
This approach allows for vehicle configuration changes without the need for reverse engineering the code, ensuring a more flexible and structured modification process.

In fact, you can see 2 two version of the vehicle inside selecting the package inside the garage menu:

3) Defining the Package Structure – Configurations Phase

back

1) Packages and Vehicles



We can define multiple versions of the vehicle simply by creating additional files within the mod folder.

3) Defining the Package Structure – Configurations Phase

back

2) Info.json, .pc and png files

In a **BeamNG** mod, the **info.json**, **.pc**, and **.png** files serve different purposes to properly integrate the vehicle into the game. Here's what each file does:

info.json (Vehicle Information File)
This file contains metadata about the vehicle, such as:

- **Vehicle name** (displayed in the game menu)
- **Author** (who created the mod)
- **Description** (a brief overview of the vehicle)
- **Available configurations and colors**

It helps **BeamNG** organize and display the mod correctly in the vehicle selection menu. There is **one main** info.json **file** for the package and **different** info.json **files** for each vehicle configuration. In our mod, we also use main.material as the texture file to define the available colors and to set the **black** and **red** colors of the real vehicle. To do this, you need to define the colors inside the package **info.json** file, just like we did.

3) Defining the Package Structure – Configurations Phase

back

2) Info.json, .pc and png files

You can take these colors as a base for your work.

```
"colors":{
    "White": "1 1 1 1.524",
    "Charcoal": "0.191 0.191 0.191 1.536",
    "Black": "0.106 0.108 0.109 1.645",
    "Classic Green": "0.000 0.258 0.1451 1.552",
    "Yellow": "0.949 0.625 0.0263 1.552",
    "Orange": "0.83323 0.465697 0.0507703 1.524",
    "Red": "1 0 0 1.8",
    "Dark Blue": "0.034 0.233 0.561 1.404",
    "Light Blue": "0.236 0.495 0.967 1.723",
    "Bright Green": "0.0198987 0.766101 0.0298481 1.7"
}
```

3) Defining the Package Structure – Configurations Phase

back

2) Info.json, .pc and png files

.pc File (Vehicle Configuration File)
The **.pc** file defines a specific **preset configuration** of the vehicle, including:

- **Parts selection** (e.g., wheels, engine, suspension, body panels)
- **Color settings** (defaultpaintName1…)
- **Tuning parameters** (gear ratios, suspension stiffness, etc.)

Players can save and load different configurations of the same vehicle using this file.

.png File (Vehicle Thumbnail Image)
This is the **preview image** that appears in the vehicle selection menu.

- It should show the vehicle clearly.
- The file name usually matches the **.pc** file to ensure the correct image is displayed for each preset.

Vehicle's Analisys

With some limits, we achieved replicating the real vehicle dynamics and structure, some properties are:

Objectives achieved

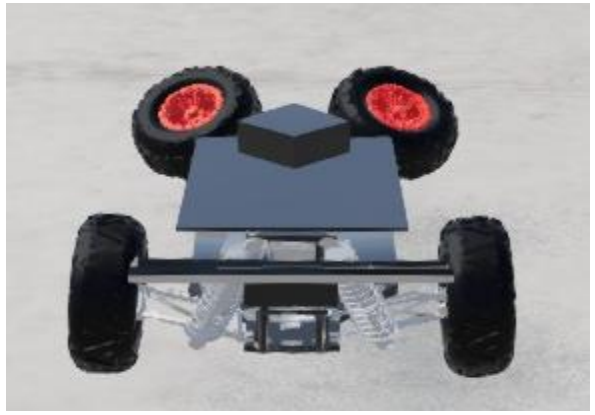
- 1:1 scale and good stability
- Independent suspensions
- Max speed**:** 80 km/h
- Braking pitch
- Electric engine and battery
- 4x4 traction and 3 differentials system
- Good cornering and entry speed near to the real vehicle

To achieve

- Weight not close to that of the real vehicle (120kg vs 20kg)

To better understand the vehicle's properties, it is important to consider the key aspects of its creation. Our goal was to achieve a balance between **stability**, **the number of nodes and beams**, **weight**, and **driving realism.**

Considerations



- **Stability**

Ensures the vehicle behaves correctly during different driving phases, such as cornering at various speeds, braking, and accelerating. A lack of stability can make the vehicle **undrivable** or even cause it to **malfunction or break apart**.

- **Number of nodes and beams**

A more complex vehicle will have a greater number of nodes and beams, making it more realistic-similar to the cars already present in the game, provided they are used correctly.
Our focus was on the **suspension dynamics**, particularly movements such as **pitching during braking** and **cornering at low speeds**.
However, increasing the number of nodes and beams also **raises debugging difficulty** and introduces a **greater number of variables to manage**.

Without compromising driving realism, we optimized the design by removing **20 nodes** from the initial version!

Considerations

- **Weight**

The first version of the vehicle weighed over **240 kg**, but in the second version, we successfully **reduced this significantly**. Weight plays a crucial role in **stability**:

- **Increasing the weight of the nodes** helps reduce vehicle oscillations, making the car more stable.
- A well-balanced weight distribution also simplifies the **modding process** and improves overall handling.

We successfully achieved a well-balanced weight distribution like can be seen in the image on the left.

- **Comparison to Other Vehicles:**

There are no small vanilla vehicles in the game, and the only comparable mod is a much heavier and larger vehicle. Our focus was to create a vehicle with a practical and balanced size and weight that performs well under various conditions.

In summary

- **Approach**

Our approach to creating the vehicle was driven by careful consideration of **stability**, **node and beam structure**, and **weight**. We made strategic adjustments to ensure the vehicle felt realistic while also being **stable**, **manageable**, and **optimized for performance**.
This approach allowed us to maintain realism in the driving experience without compromising performance or the vehicle's ability to navigate different challenges.

- **Sensor's integration**

We also considered the **integration of sensors** in the vehicle's design, ensuring that their **placement and scale** align with the **real-world proportions**. This allows for a more accurate and realistic sensor setup, optimizing performance in various driving scenarios. By maintaining the real vehicle scale, we ensure the sensors' behavior and data, such as distance measurements, are realistic and effective for testing.