

**USER GUIDE FOR TL FORGE 2024**

**ALEJANDRO SAINZ**

Contents

[Introduction 1](#_Toc162323915)

[Features 1](#_Toc162323916)

[Setup 1](#_Toc162323917)

[Requisites 1](#_Toc162323918)

[Configuration 1](#_Toc162323919)

[Quick Start 2](#_Toc162323920)

[How to use the asset in your project 3](#_Toc162323921)

[Demo Scenes 3](#_Toc162323922)

[2D Demo 3](#_Toc162323923)

[3D Demo 5](#_Toc162323924)

[Scripting support 5](#_Toc162323925)

[FAQs 7](#_Toc162323926)

# Introduction

Thank you for purchasing!

The Unity Tag and Layer Forge is an intuitive tool designed to enhance the workflow within the engine by customizing Tags, Layers and Physics Collisions directly from the inspector of a GameObject.

Feel free to contact for any enquiry.

Email: [alejasainzmartinez@gmail.com](mailto:alejasainzmartinez@gmail.com)

# Features

Direct Management from the Inspector: Perform all Tag and Layer operations without leaving the context of your GameObjects. Manage your project settings all from one place.

Efficiency: Save valuable development time with streamlined operations.

Clarity: Gain a better understanding of your project's collision matrix with an intuitive visual interface.

# Setup

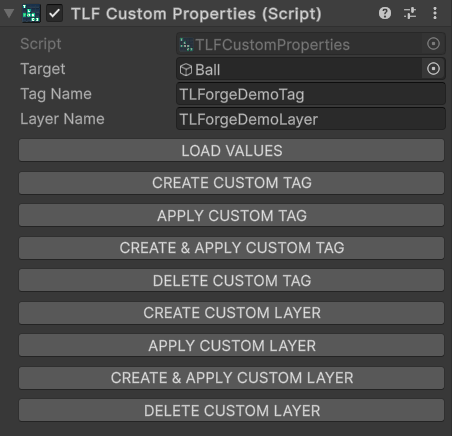
## Requisites

The Unity Tag and Layer Forge has been developed and tested using Unity 2023.2.7f1.

## Configuration

Just by importing the package the user will have enabled TL Forge functions on its Unity Editor environment.

# Quick Start

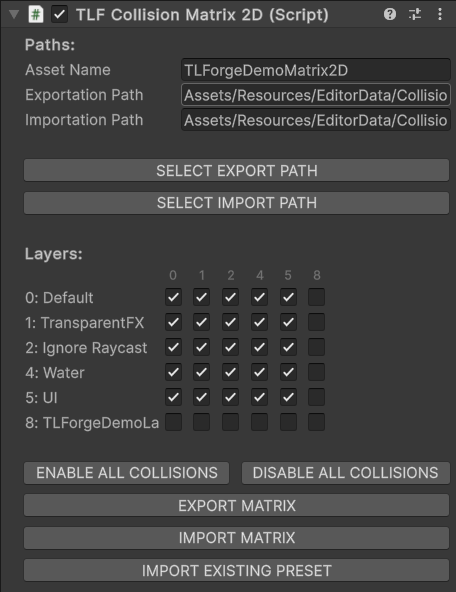
TL Forge presents three main components the user will use more often

**1-TLFCustomProperties** allow the user to define a **Tag Name** and a **Layer Name** to apply to that specific **Target** GameObject.

The component shows a series of buttons that handles from Tag/Layer creation to deletion. This means the corresponding name will be stored or removed from Unity.

Once a name gets created and applied the Target will automatically have that defined value.

This is the main core of the tool.

 **2-TLFLayerCollisionMatrix2D** allow the user to define the state of the project’s Physics 2D collision matrix by showing how the existing layers interact with each other.

Every layer interaction is showed as a checkbox the user can mark to set a new collision detection.

Layers can be associated by the indices showed vertically and horizontally.

This component makes possible to define presets of Matrices by exporting the custom matrix to a path defined in the field Exportation Path.

The user can define an **Asset Name** for the Matrix preset and an **Exportation Path** and an **Importation Path**.

The user can either defined both paths by writing or by using **SELECT EXPORT PATH** and **SELECT IMPORT PATH**.

Once a configuration satisfies the user needs is possible to create a Matrix preset by using **EXPORT MATRIX**.

Every time the user selects the GameObject with **TLFLayerCollisionMatrix2D** component the preset matrix defined will be imported.

If the user made changes to the matrix as long as those changes are not saved by exporting the matrix again, the **IMPORT MATRIX** feature will load the matrix without changes.

Alternatively, the **IMPORT EXISTING PRESET** feature can load any of the created presets without having to define an Importation Path or an Exportation Path.

**3-TLFLayerCollisionMatrix3D** is a Physics based version of the TLFLayerCollisionMatrix2Dcomponent.

## How to use the asset in your project

# Demo Scenes

A Demo folder is Inside TLForge directory showing the dimensions with which the tool work.

## 2D Demo

A first scene called **“TLForge\_2D\_Demo”** presents a 2D based collision test on a canvas with UI buttons that access some of the TLForge features.

As a first step the user can select any GameObject on the scene and deploy the existing tags and layers to see a **TLForgeDemoTag** and a **TLForgeDemoLayer** already created for demonstration purposes.

Next check the collision matrix existent in the project by selecting the TLForge tool and looking for the **TLFLayerCollisionMatrix2D** component. The matrix shows all interactions with **TLForgeDemoLayer** deactivated.

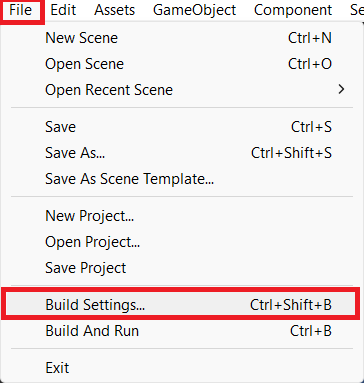
Hit Play mode and the demo will show a 2D bouncing circle falling and hitting the floor. This collision occurs because the layers on the ball and the floor (default and UI respectively) are actually interacting with each other. But if the user hit the **New Layer** button on the scene, the circle will be on **TLForgeDemoLayer** and then will fall below the floor. After that you can hit the **Reset** button to return the scene to its initial values.

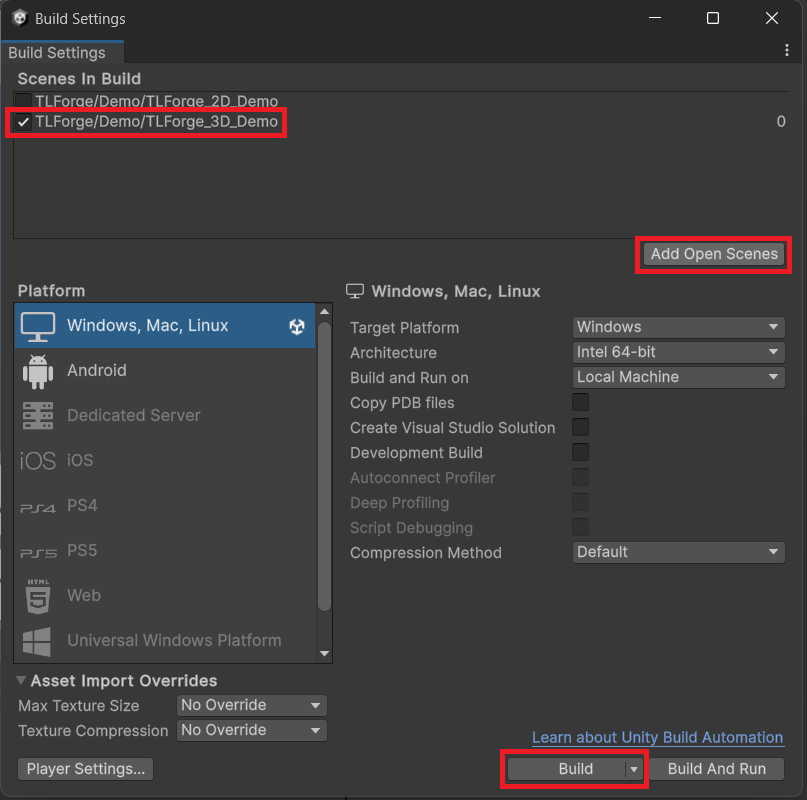
As mentioned before, TLForge handles different presets for the collision matrix, and to test that feature, hitting the **New Preset** button will use a different preset already stored on the base project. This second preset register collisions between **TLForgeDemoLayer** and **UI** / **Default**, therefore if now we hit **New Layer** button will see how the circle won’t overlap the floor anymore. You can now change between layers and the circle will stay still. Finally hit **Original Preset** and the circle will behave as it was when you run the demo.

The demo also allows the user to modify the circle’s tag through the UI using **New Tag** and **Original Tag** buttons.

To test one last thing, after hitting **Exit** button to stop the demo go to:

File 🡪 Build Settings 🡪 Add Open Scenes 🡪 TLForge/Demo/TLForge\_3D\_Demo 🡪 Build.





After you select a desired directory and the demo build gets created you can run the build and check everything working just as it was on the Unity Editor.

## 3D Demo

A second scene called **“TLForge\_3D\_Demo”** presents a 3D based collision test on a canvas with UI buttons that access some of the TLForge features.

The resemblance with the previous scene is notorious, therefore if you followed the previous steps, you only need to know the Unity handles collisions not only in layers, but also by dimensions. That is why on this scene the main difference with the previous one is that you need to work with **TLFLayerCollisionMatrix3D** component instead of the 2D one.

# Scripting support

Because it is not possible to remove or add tags and layers in project builds, only the functions that the user will most likely access through code (to execute something both in the Editor and in a Build) will be covered.

If you want to modify the current Layer or Tag of a GameObject:

// Required fields

private TLFCustomProperties cProperties;

private GameObject newTarget;

// Creates if don’t exist (only on Editor, on Builds just finds the Tag) and applies

// the Tag contained on ‘Tag Name’ to the ‘Target’, both fields of the

// TLFCustomProperties component.

cProperties.CustomizeThenApplyTag ();

cProperties.ApplyCustomizedTag(); // Only applies the Tag

// Creates if don’t exist (only on Editor, on Builds just finds the Tag) and applies

// the Tag contained on ‘Tag Name’ to the ‘newTarget’ value.

cProperties.CustomizeThenApplyTag(newTarget);

cProperties.ApplyCustomizedTag(newTarget); // Only applies the Tag to ‘newTarget’

// Finds the Tag "TLForgeDemoTag" and applies it to the ‘Target’ of the

// TLFCustomProperties component.

cProperties.CustomizeThenApplyTag("TLForgeDemoTag");

// Finds the Tag "TLForgeDemoTag" and applies it to the ‘newTarget’ value.

cProperties.CustomizeThenApplyTag(newTarget, "TLForgeDemoTag");

// Creates if don’t exist (only on Editor, on Builds just finds the Layer) and applies

// the Layer contained on ‘Layer Name’ to the ‘Target’, both fields of the

// TLFCustomProperties component.

cProperties.CustomizeThenApplyLayer();

cProperties.ApplyCustomizedLayer(); // Only applies the Layer

// Creates if don’t exist (only on Editor, on Builds just finds the Layer) and applies

// the Layer contained on ‘Layer Name’ to the ‘newTarget’ value.

cProperties.CustomizeThenApplyLayer(newTarget);

cProperties.ApplyCustomizedLayer(newTarget); // Only applies the Layer to ‘newTarget’

// Finds the Layer "TLForgeDemoLayer" and applies it to the ‘Target’ of the

// TLFCustomProperties component.

cProperties.CustomizeThenApplyLayer("TLForgeDemoLayer");

// Finds the Layer "TLForgeDemoLayer" and applies it to the ‘newTarget’ value.

cProperties.CustomizeThenApplyLayer(newTarget, "TLForgeDemoLayer");

If you want to modify the Collision Layer Matrix preset:

// Required fields

private TLFCollisionMatrix2D matrix2D;

private TLFCollisionMatrix3D matrix3D;

// Import the ‘TLForgeDemoMatrix2D’ json stored in ‘EditorData/CollisionData/2D/’

matrix2D.Import("EditorData/CollisionData/2D/", "TLForgeDemoMatrix2D");

// Import the ‘TLForgeDemoMatrix3D’ json stored in ‘EditorData/CollisionData/3D/’

matrix3D.Import("EditorData/CollisionData/3D/", "TLForgeDemoMatrix3D");

# FAQs

**Why an error shows up every time the TLForge GameObject is selected?**

If an error with a message “There is no CollisionMatrix to import at Assets/EditorData/CollisionData/LayerCollisionMatrix.json” shows up every time the TLForge is selected is because the tool is loading some inexistent preset Matrix in your project. The error won’t show up anymore if a correct preset is setted.

Every time a new TLForge gets created, the LayerCollisionMatrix is setted as a default preset, but is very likely that don’t exist in the project, therefore a new valid preset needs to be created.

**Why the Target of the TLFCustomProperties can never be null?**

a

**Why if a layer gets deleted while it’s interactions where activated and then another layer gets added into project by default the matrix mark an interaction there?**

a