Debug Toolkit for Unity (UDT)

> *Thanks for using the UDT. Unity 6+ is supported for this asset.*

This documentation aims to get you ready to use and modify UDT for your awesome projects.

The current version of the toolkit 1.0.

If you are looking for the last patch note please check the Patch Notes. If you are looking for features in development take a look at the What's Next section. Discord

---

Features

UDT is composed of four elements :

- A modular console, for which can create custom commands but also log anything you want at runtime. Maybe you wish to have a simple way to kill your player, teleport to some place, show all the colliders that are in the scene or maybe you want to go frame by frame to know what is happening. Now its a simple command line.
- A system of optimized runtime Gizmos with an easy to use API. It makes your debugging way easier. Control them with the console, or as a standalone feature.
- A FreeCam to inspect your scene at runtime. With a simple command in the console activate or deactivate the FreeCam and navigate through your scene.
- A metrics system to now your fps, the number of batches, and your tris/vert usage.

Introduction to UDT

This toolkit aims to make your life easier while debugging your game.
In this version we focused on the run time debugging

We are aiming for a load of improvements in the future versions of this toolkit. If you want to now more about the next features please check the What's Next section.

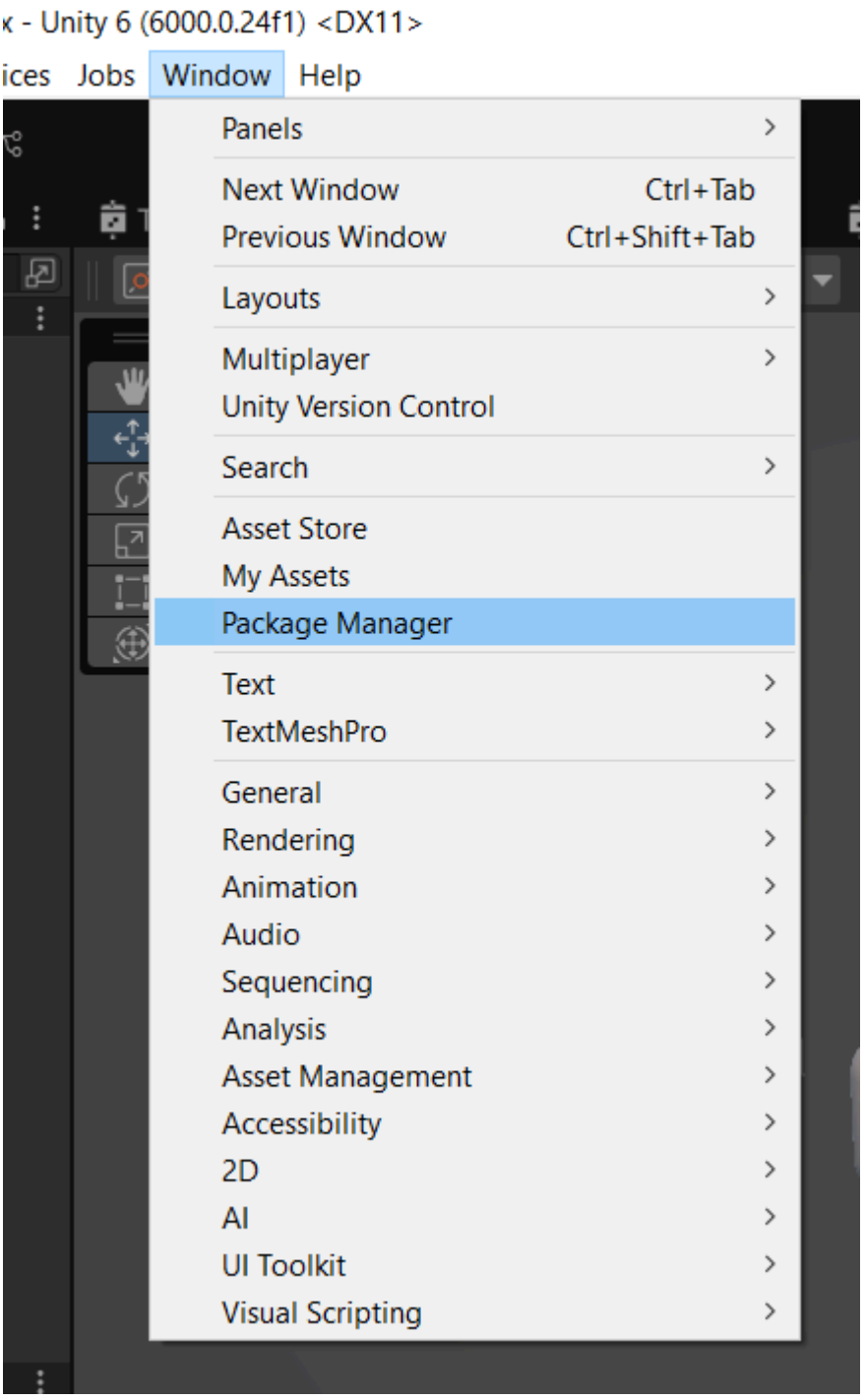
In the next few sections you are going to learn how to import UDT to your project and how to use it.

Install UDT

> *Make sure that you are using Unity 6 or above.*

Package manager

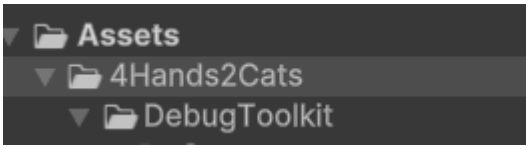
Get into the unity package manager at the top of the Unity Engine window and look for UDT in your assets



Click on the **download** button at the bottom right of the menu.

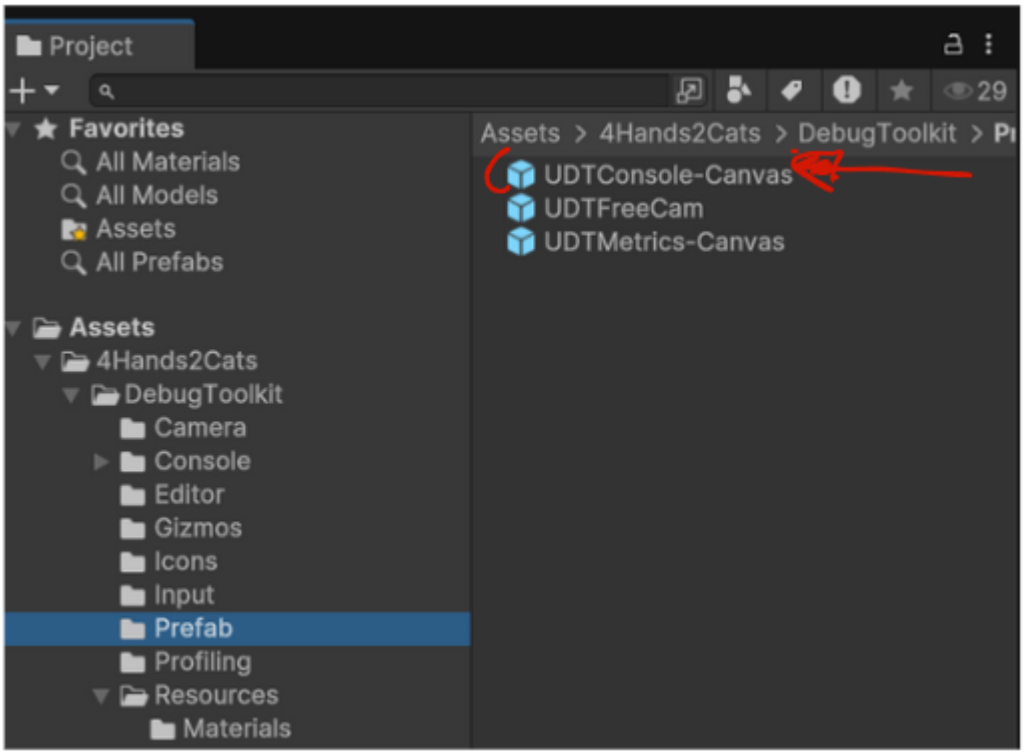
Once the files are downloaded, import them in your project by clicking on the **Import** button.

The toolkit is in the 4Hands2Cats folder. In the future if we do more assets they'll install in this folder as well.



Quick start guide

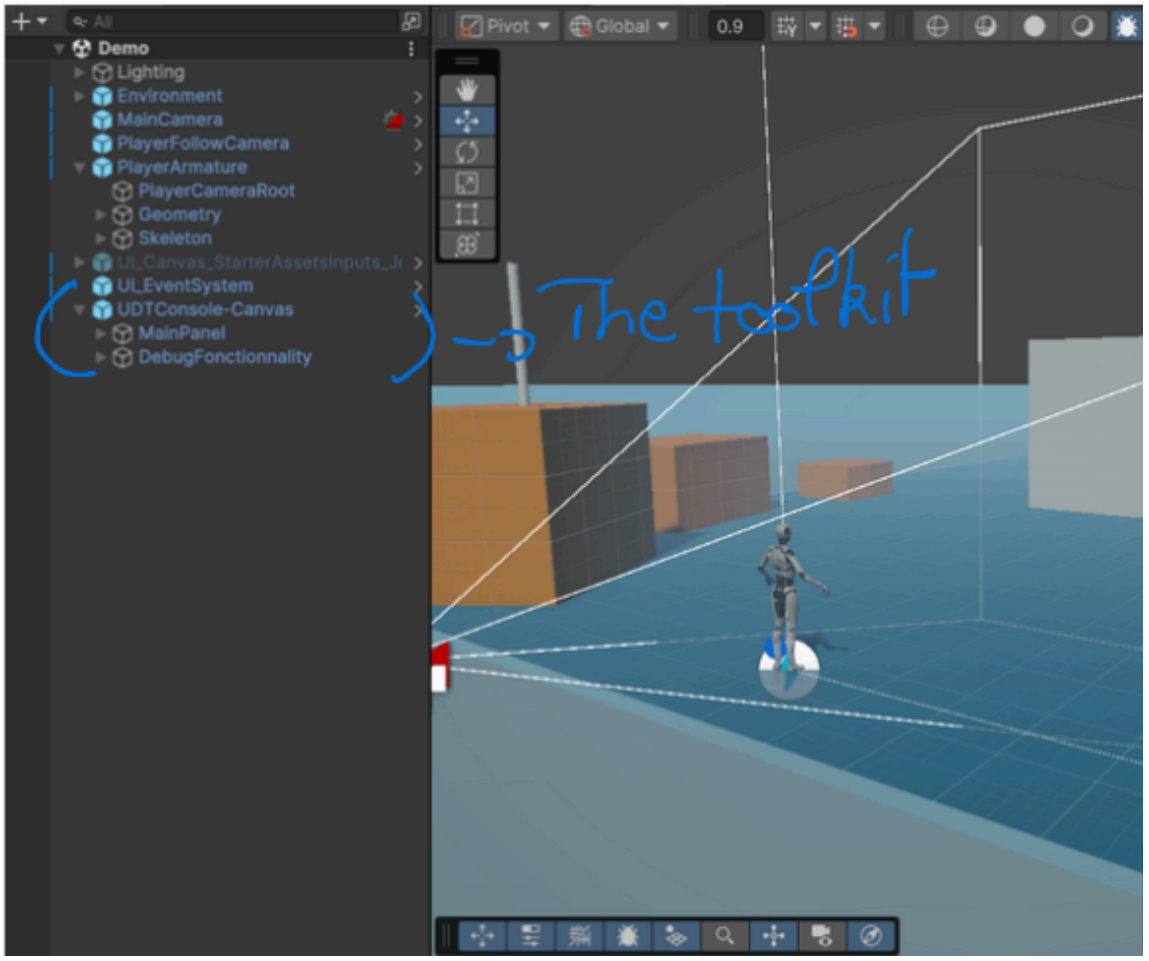
As you just seen in the demo scene, the toolkit is pretty simple to include in your project. You just have to drag and drop the **UDTConsole-Canvas** prefab from the prefab folder inside your scene.



Don't forget to remove it for production builds.

Demo Scene Tour

The demo scene is based on the third person template of Unity. We've added our debug toolkit in this scene.



To use this demo scene just press play. The Console and the all the other features are on the **F12** key.

To navigate in the scene use the **WSDQ** keys and the mouse to control the camera.

Console

The In-game console is the central piece of UDT. It controls every features of the toolkit and more.

> *This section is the documentation of the usage of the console as is. In the customization section you'll find explanations on how to make your own commands for the console.*

Refer to the Quick Start section to learn how to enable the in game console for your project.

Built in commands

There is a set of built in commands that you can use to control the toolkit and enable/disable features.

Like in any console you can use the top and bottom arrow of your keyboard to navigate through the command you've already written.

Simple

- Type *help/Help/-h/-h* in the console to get a list of all the available commands.

Booleans

- Type *metrics/Metrics/-m/-M* followed by *enable/e* or *disable/d* to enable or disable the metrics.
- Type *freecam/Freecam* followed by *enable* or *disable* to enable or disable the freecam.
- Type *light/-l* followed by *enable/e* or *disable/d* to enable or disable all the lights in the scene.
- Type *shadows/-s* followed by *enable/e* or *disable/d* to enable or disable all the shadows in the scene.
- Type *Collider/collider/-c/-C* followed by *enable/e* or *disable/d* to enable or disable the in game gizmos rendering for the collider (beware on large scene there might be a small freeze when enabling). This command also activates all the other in game gizmos [going to change in next version]
-
- Type *Gizmos/gizmos/-g/-G* followed by *enable/e* or *disable/d* to enable or disable the in game gizmos.

Vector

- Type *Time/time/-t/-T* followed by a float value between 0 and 100 to change the time scale of your game.
- Type *Frame/frame/-f/-F* followed by a float value x to navigate x frames in the future. [Note : If the number of x is too high you might wait a long time].

Enum

- Type *graphics/Graphics/-g/-G* followed by *low/l* or *medium/m* or *high/h* or *ultra/u* to change the quality settings. [Note : This scriptable object of this command will have to be changed if your quality settings are not following the standard ones of an URP project (*verylow/low/medium/veryhigh/ultra*)].

Debugs logs

Free Cam

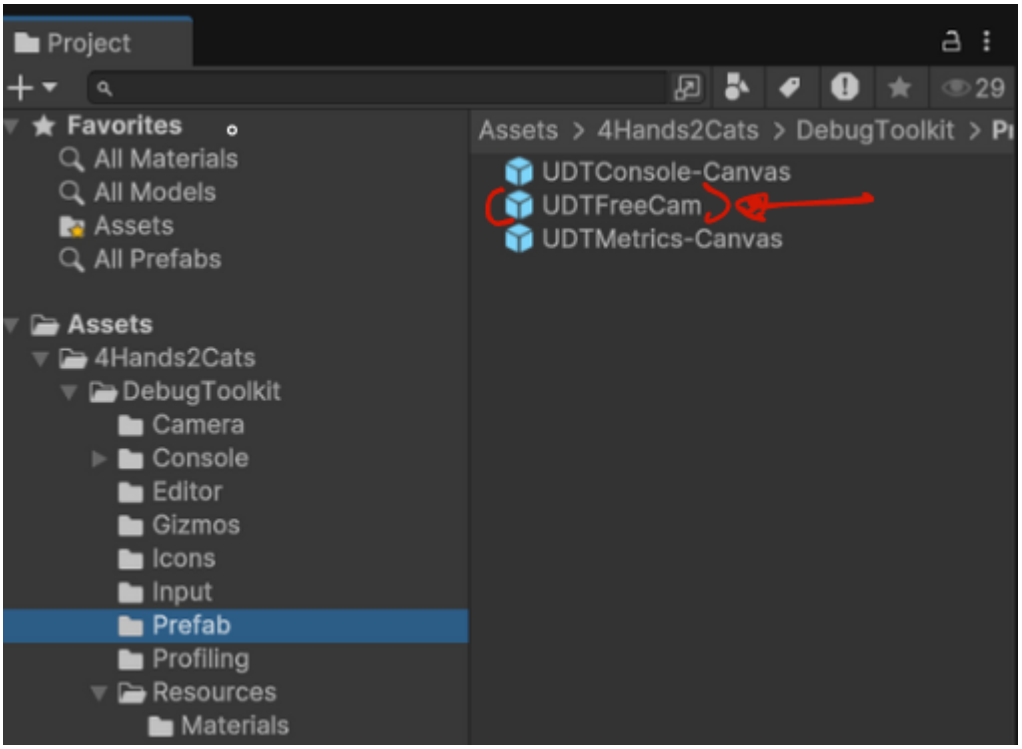
The free cam is here to give you a tool that's similar to the navigation in the scene panel of unity, but at runtime.

To use type *freecam/Freecam* followed by *enable* or *disable* to enable or disable the freecam in the console after opening it using **F12**.

Though this embeded in the package, the freecam comes as stand alone feature. Feel free to use it for your gameplay if you want.

> *Future versions of the cam should be compatible with cinemachine.*

The prefab for this one is in the same folder as the console.



To use as a stand alone feature just drag and drop the prefab in your scene.

> *Note : If you use it as stand alone feature it'll not be control anymore by the console.*

Gizmos

As you now there is already an API in unity to draw gizmos for debugging purposes. But you cannot draw gizmos for run time. Those gizmos are harvesting the power of the **GL** API to show performance friendly gizmos at runtime for quite anything.

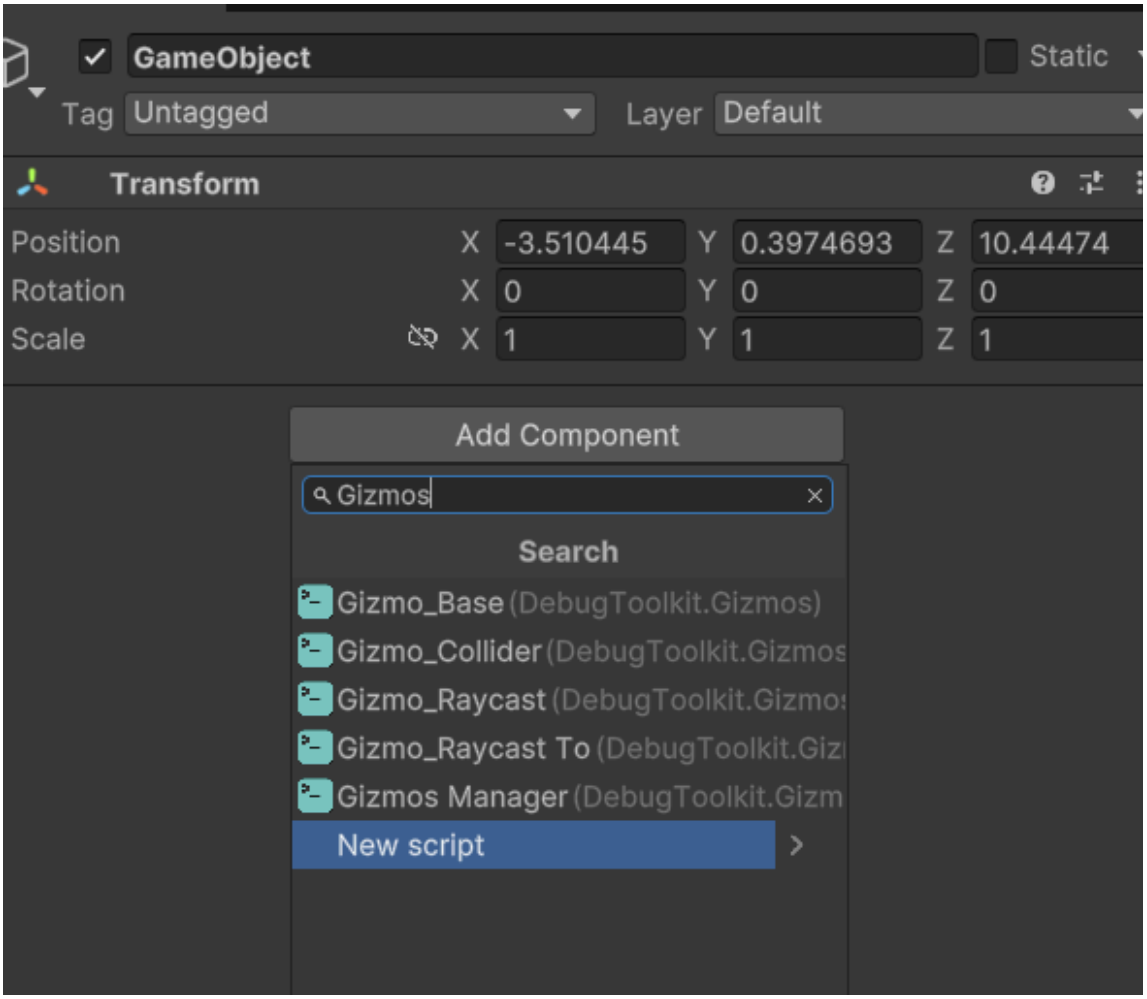
There are two way of using the gizmos. Manual and automatic using the console.

- The manual way consist of adding the gizmos components to your gameobjects
- The automatic way is used to show all colliders in the scene. To show all colliders simple type the command : *Collider/collider/-c/-C* followed by *enable/e* or *disable/d* to enable or disable the in game gizmos rendering for the collider (beware on large scene there might be a small freeze when enabling). This command also activates all the other in game gizmos [going to change in next version] int the ingame console.

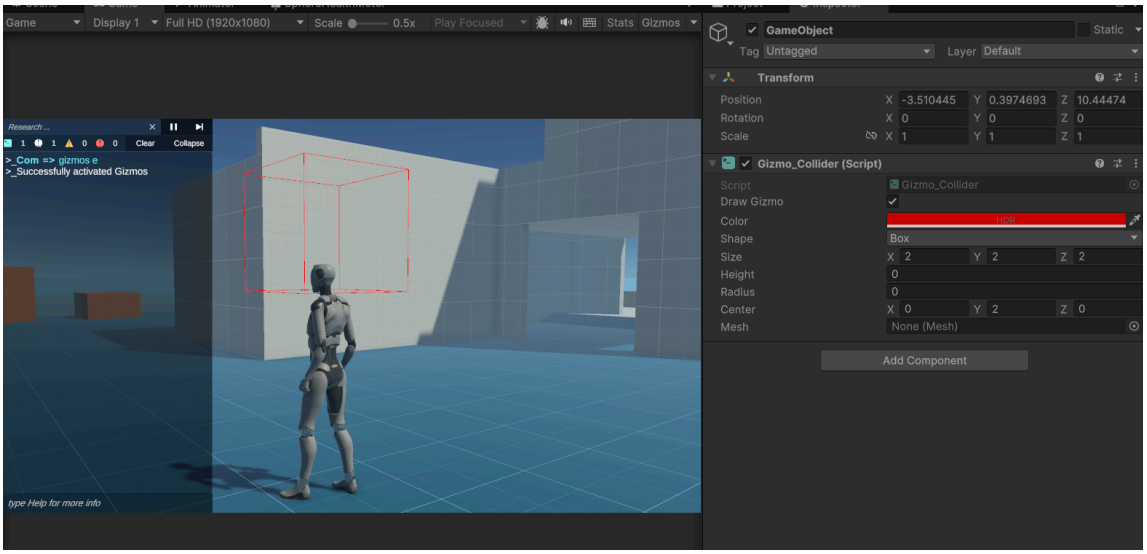
Gizmos that are manually setted up can are still managed by the ingame console. Simple type : *Gizmos/gizmos/-g/-G* followed by *enable/e* or *disable/d* to enable or disable the in game gizmos.

> *Note : this commands would also hide the collider gizmos. We are awaiting for you feedback to improve this feature.*

To set up a Gizmos as component simple add the choosen Gizmos to your gameobject.



Use the gizmos Collider to choose show a collider like shape.

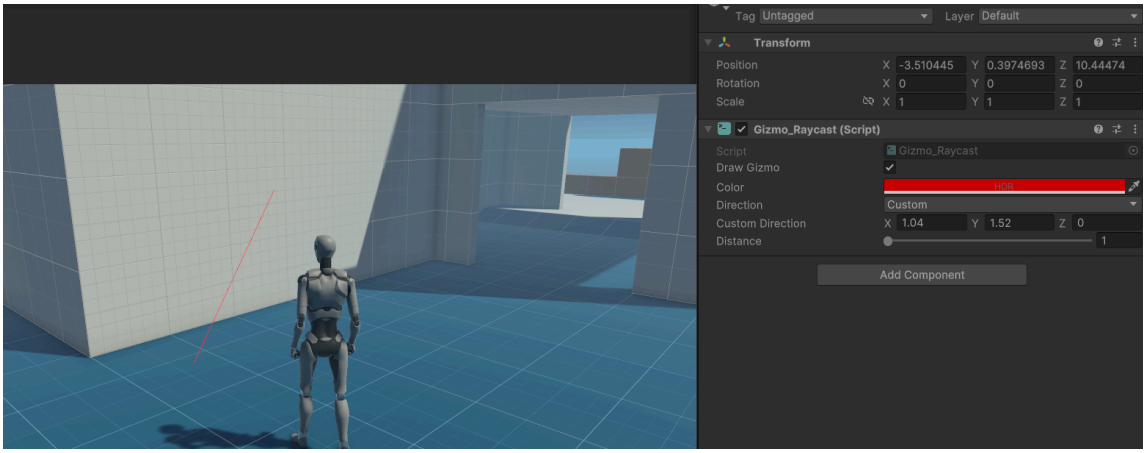


There are four options :

- The **Box**, that you can control using the **Size** and the **Center** parameters
- The **Sphere**, that you can control using the **Radius** and the **Center** parameters
- The **Capsule**, that you can use using the **Radius**, the **Center** and the **Height** parameters
- The **Mesh**, that you can use by drag and dropping a mesh in the mash container of the gameObject. [To enable this feature you need to enable Read/Write on your meshes in the import settings].

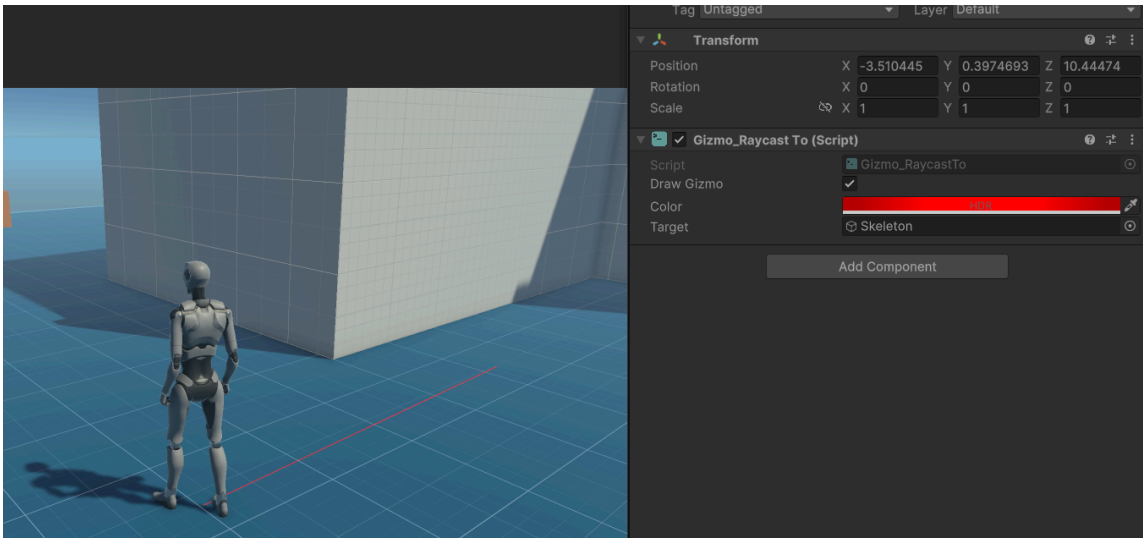
> *Note that those gizmos support bloom and they can be used as part of your project for other purposes than debugging.*

Use the Gizmos Raycast component to draw a Ray cast.



There are some direction preset for all the cartesian direction (up, left, etc...) but you can use custom to choose your own direction.

Use the Gizmos Raycast To component to draw a gizmos between two targets.



You just have to give it a ref to a gameobject and it'll draw a ray to it and update the ray at everyframes.

We are planning on adding custom editors in the future to simplify the usage of those features. Please take a look at the section Gizmos API to learn about the code API for the gizmos.

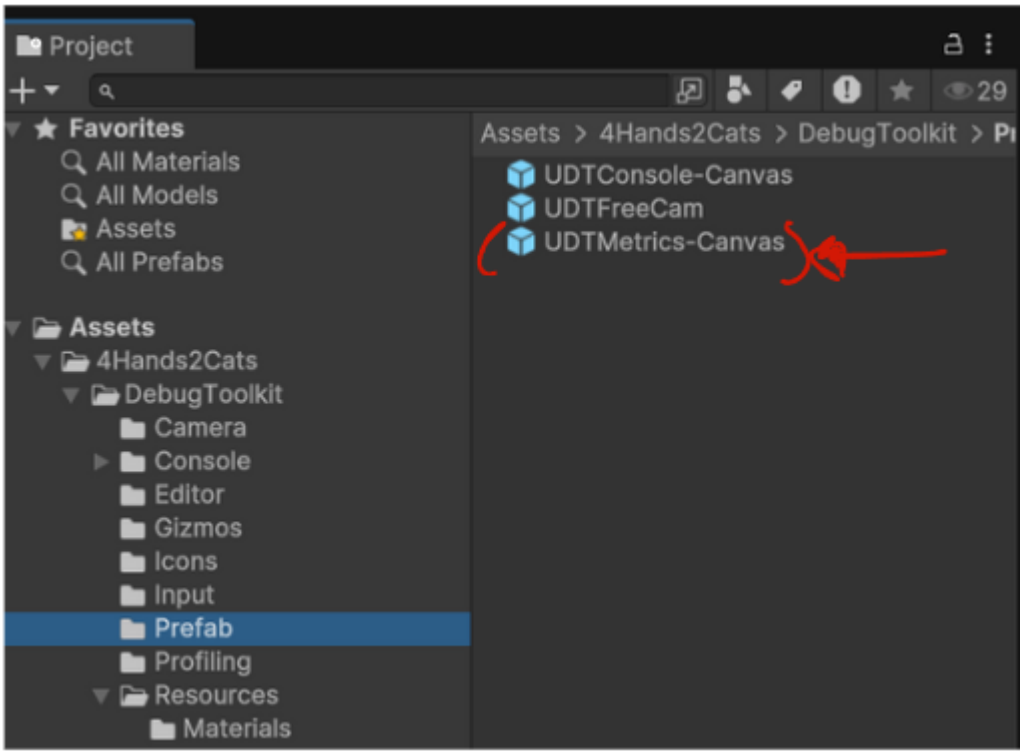
Metrics

The metrics are enabled using the interactive console with the command :

> *metrics/Metrics/-m/-M followed by enable/e or disable/d to enable or disable the metrics.*

For now it show the FPS, the number of batches, the number of tris and the number of vert. In future updates it'll help you to do some profiling.

You can use the metrics as a stand alone feature by drag and dropping it in your scene.



> *Note : If you use the metrics as a standalone feature, it'll note be managed by the console anymore.*

Gizmos Api

You can create and delete gizmos at runtime using the Gizmos API.

> *Note : this APIs might change depending on your feedback. Make sure to come back to this section if your logic breaks.*

For now there only is an API for the Collider Gizmos, but we are planing on completing this so you can use it for raycasts to.

```
public static Gizmo_Collider DrawBoxGizmos(GameObject targetObject, BoxCollider boxCollider, Color colliderColor)
public static Gizmo_Collider DrawSphereGizmos(GameObject gameObject, SphereCollider sphereCollider, Color gizmosColor)
public static Gizmo_Collider DrawCapsuleGizmos(GameObject gameObject, CapsuleCollider capsuleCollider, Color gizmosColor)
public static Gizmo_Collider DrawMeshGizmos(GameObject gameObject, MeshCollider meshCollider, Color gizmosColor)
```

You've got four static methods to work with. The first argument is always the game object you wish to add the Gizmos Collider to. The second argument is the corresponding Unity Engine collider component and the third one is the color you wish to use.

The Gizmos will spawn at the position of the collider and look exactly the same, but will work at runtime.

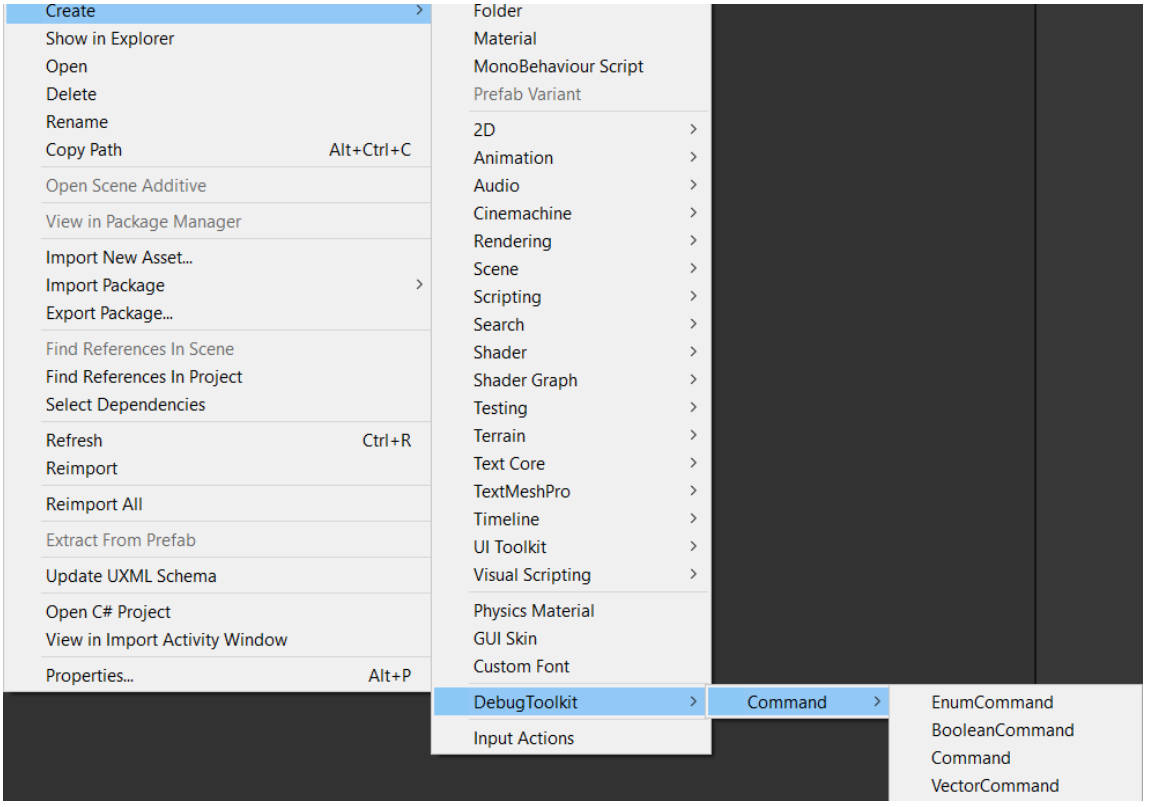
We plan on completing this API, and 2d collider support.

Make your own commands

The command system is based on scriptable objects. Scriptables object are really practical to use because they are Gameobject and scene agnostic.

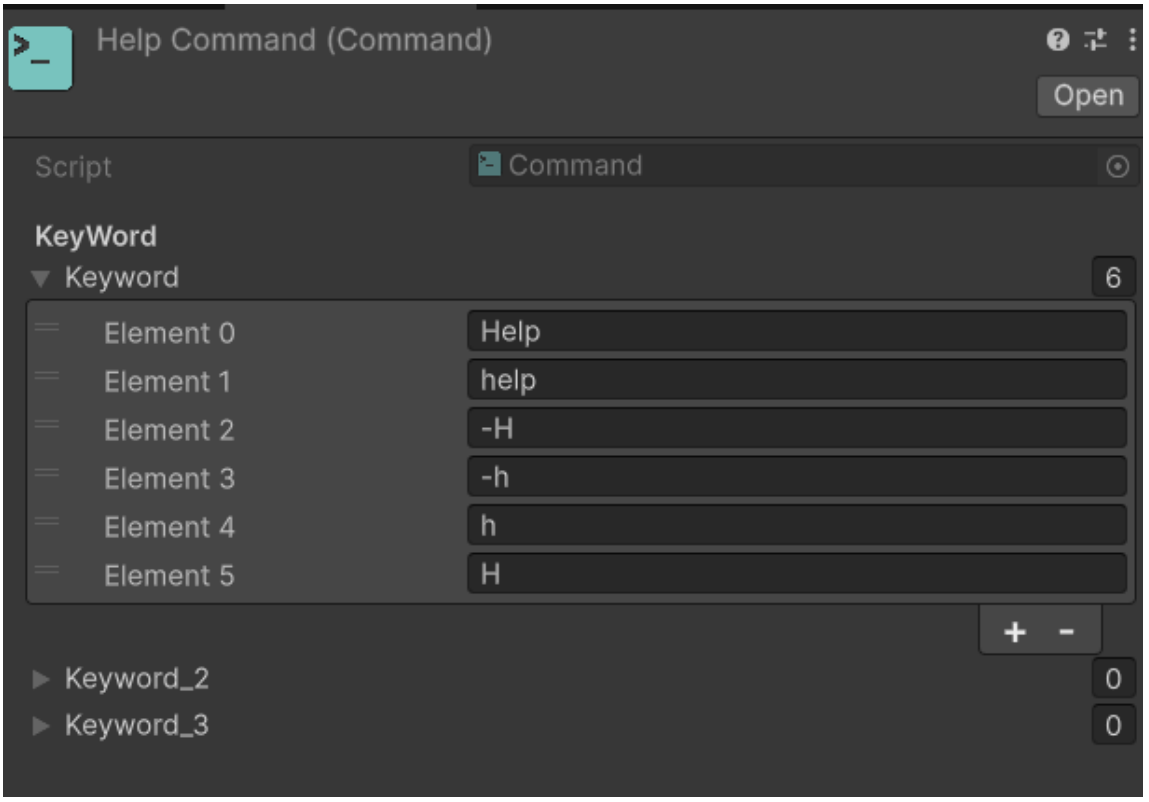
> *We are planning on making an API to create commands using C# attributs which'll give you two options to create your console commands.*

You can create like this

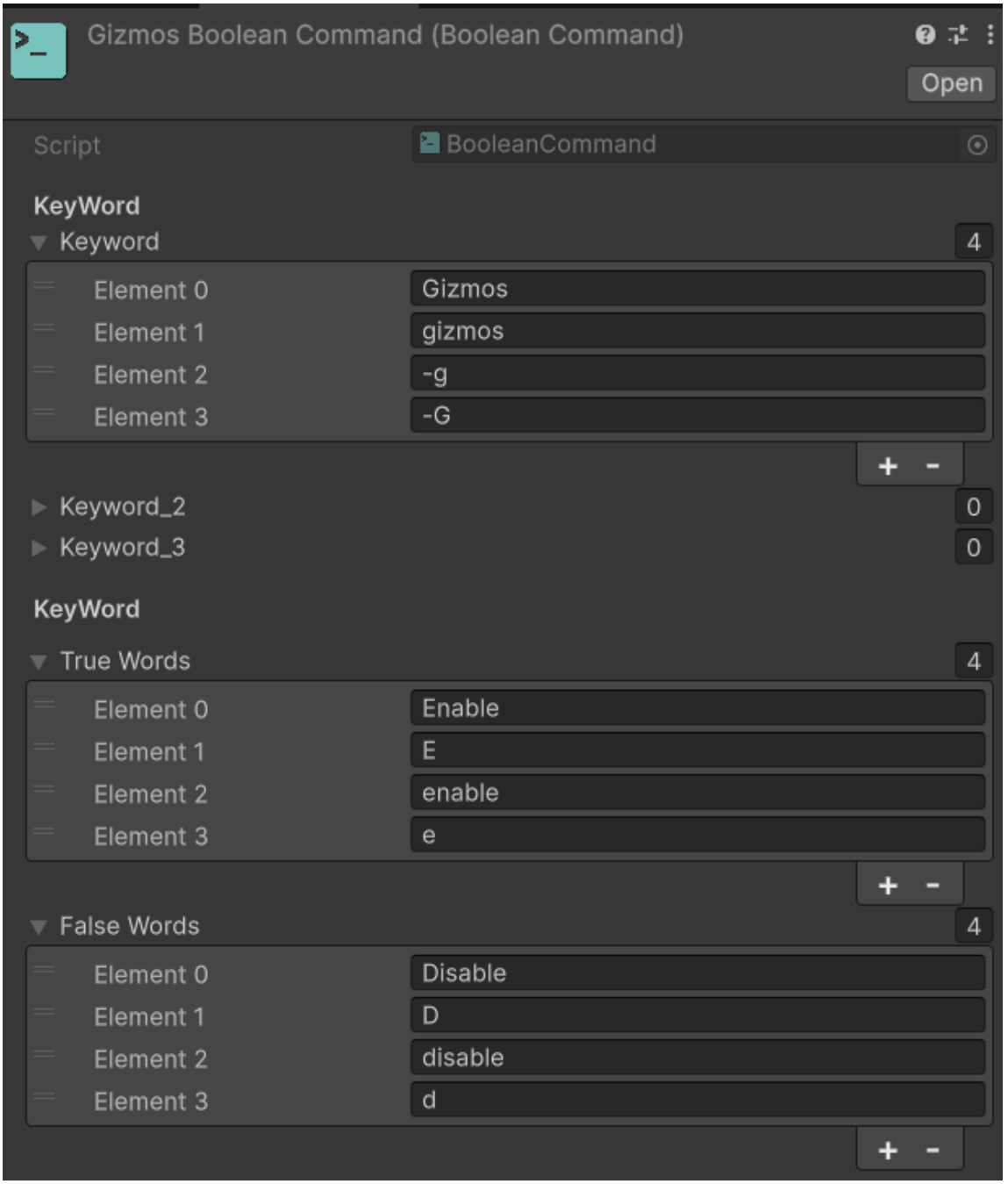


There are four type of commands :

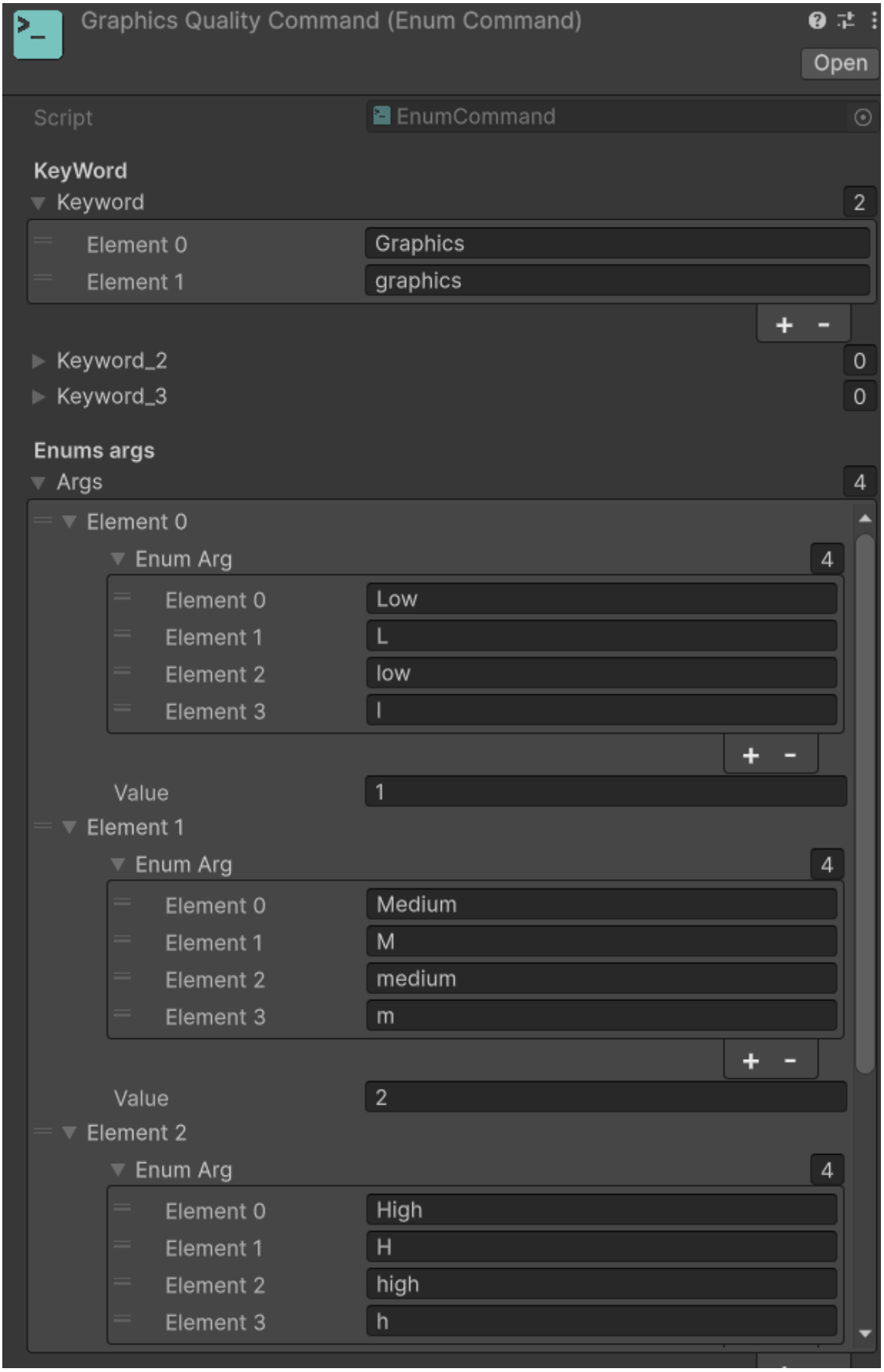
- The **Command** is the most basic one it is used to simply inputs a chain of up to three words that would send a signal of type => Command Got Activated. This is the one which is used for the help command per example.



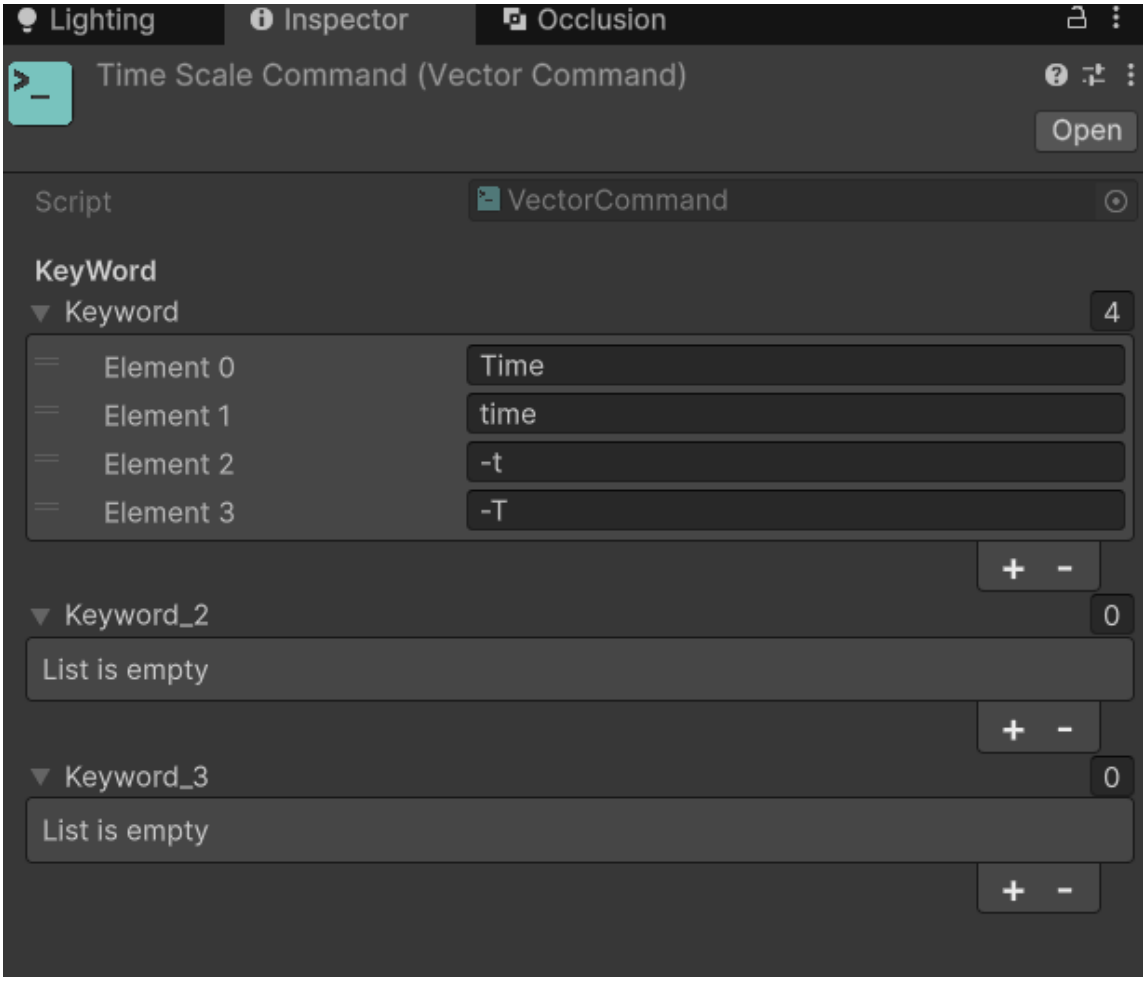
- The **Boolean Command** which takes two arguments to send either a signal activate or deactivated (true or false). This used for the Gizmos command per example. You've got the main keywords and then the keyword for true and the keyword for false.



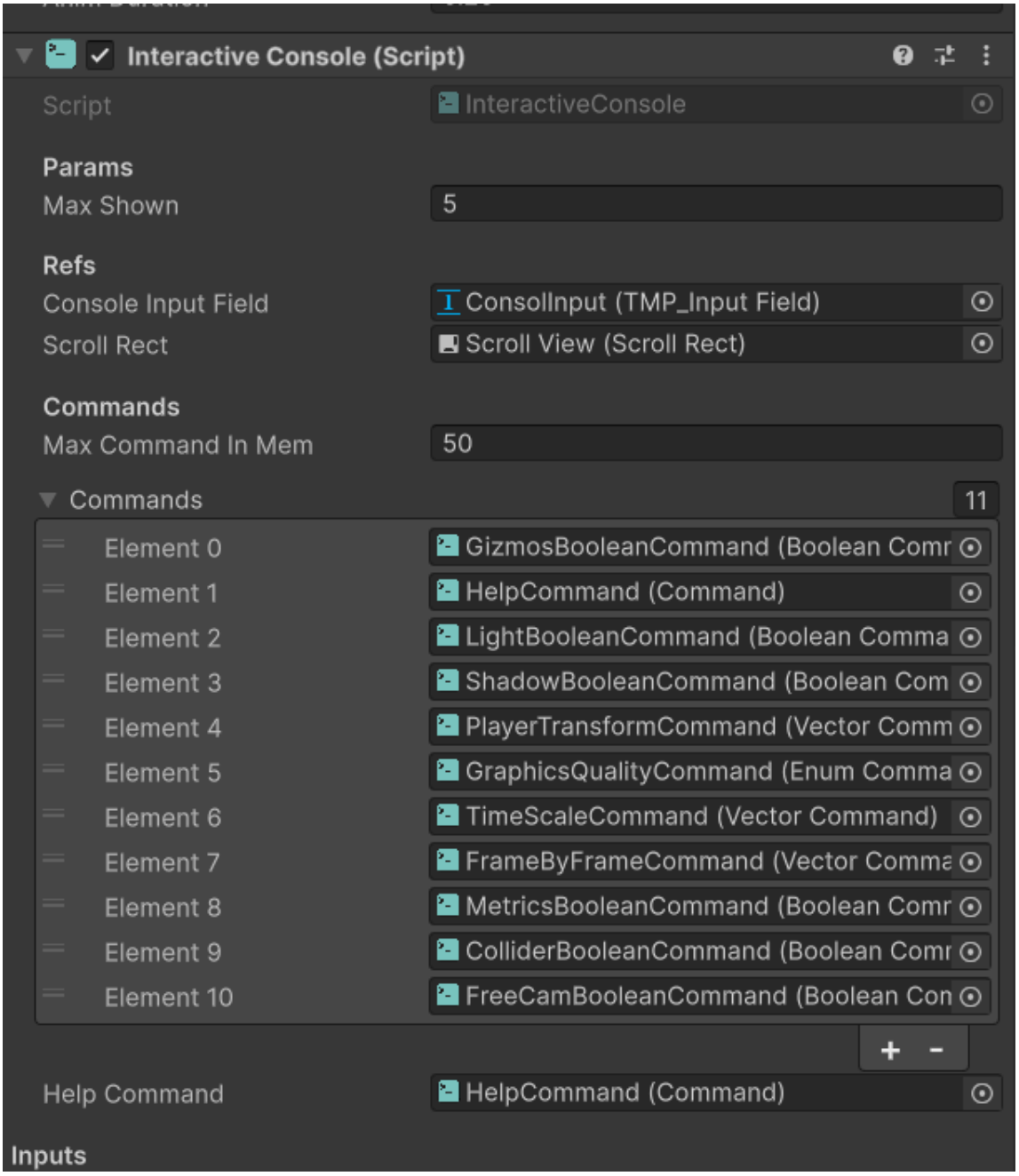
- The **Enum Command** which take n arguments to send a signal between 0 and n - 1. Its your job to handle what those index mean. You can bind those index with a keyword each so its easier to use in the in game console. This command is use in the Graphic Quality Command per example.


Graphics Quality Command (Enum Command)
Open
Script
EnumCommand
KeyWord
Keyword
2
Element 0
Graphics
Element 1
graphics
Keyword_2
0
Keyword_3
0
Enums args
Args
4
Element 0
Enum Arg
4
Element 0
Low
Element 1
L
Element 2
low
Element 3
l
Value
1
Element 1
Enum Arg
4
Element 0
Medium
Element 1
M
Element 2
medium
Element 3
m
Value
2
Element 2
Enum Arg
4
Element 0
High
Element 1
H
Element 2
high
Element 3
h

- The **Vector Command** which takes as an argument of size 1 to 4, so basically a float, a Vector2, a Vector3, or a quaternion. It has four signals, one four each vector size. Its used for the timescale command per example.



So now you now how to make a command but there is a need for a bit more set up. First you need to add this command to the interactive console component in the UDTConsole-Canvas prefab.



Now there is only one thing left for you to do : connect this command to a script of your choice. Once again this is pretty straight forward. To show you how to do it we are going to take a look at the **DebugTimeScale** script which is in the sample folder.

To be able to bind a command to one of your script you need to add the **DebugToolkit.Console** assembly to the assembly of your script.

Then in the script you need to :

- get a reference to the command you've create (you can use the resources folder or the addressable package to avoid some clicks in the inspector, here we used a serialized field).
- subscribe to the event of your choice (depending on the type of command).

- not forgetting to unsubscribe onDestroy to avoid memory leaks (pro tip : some times unity dosen't calls the OnDestroy correctly when you stop the play mod so don't forget to add a *if(this==null)return*)).


```
public class DebugTimeScale : MonoBehaviour
{
    [Header("Command")]
    [SerializeField] private VectorCommand timeScaleCommand;
    [SerializeField] private VectorCommand frameCommand;

    [Header("Control")]
    [SerializeField] private TimeControlButton pauseButton;
    [SerializeField] private TimeControlButton frameButton;


    private void Awake()
    {
        timeScaleCommand.OnVector1Inputed += HandleTimeScaleChanged;
        frameCommand.OnVector1Inputed += HandleFrameCommand;

        pauseButton.OnActivation.AddListener(delegate {
            pauseButton.Active = !pauseButton.Active;
            if(pauseButton.Active)
            {
                HandleTimeScaleChanged(val: 0);
            }
            else
            {
                HandleTimeScaleChanged(val: 1);
            }
        });
        frameButton.OnActivation.AddListener(delegate {
            frameButton.Active = !frameButton.Active;
            if(frameButton.Active)
            {
                HandleFrameCommand(amountOfFrames: 1);
                pauseButton.Active = true;
                pauseButton.UpdateColor();
                frameButton.UpdateColorAsync();
            }
        });
    }

    private void OnDestroy()
    {
        timeScaleCommand.OnVector1Inputed -= HandleTimeScaleChanged;
        frameCommand.OnVector1Inputed -= HandleFrameCommand;
    }
```

> *Note : You can find the existing commands used in the toolkit the 4Hands2Cats/DebugToolkit/Console/Interaction/CommandData folder.*

About us

At 4Hands2cats, we create debugging tools to streamline development. As two devs (and two cats), we focus on robust, user-friendly solutions for Unity. Our first asset, a complete debug toolkit, works in build and runtime for full control. We're committed to improving and expanding our tools.

FAQ

Where can I contact you to send you feedback or request some features ?

```
- Just come on the discord server and ask your question on the dedicated channel.
```

What's next ?

Usability

- Custom inspector for all components
- Completing the Gizmos API

Features

- A cinemachine freeCam
- Profiling features
- A way to create your own commands using a code API.
- An implementation of the gizmos tool to read the navmesh AI agent pathing.
- A way to print the logs and the metrics to send them to an adress or post process the result using a spreadshit.
- 2D support for colliders
- Mobile device support for the console.
- VR/XR support for the console.

Known issues

Awaiting for your feedbacks

Is there a feature you need? Feel free to ask on the dedicated discord server and we'll see what we can do. Our goal is to improve this asset as much as possible so it fullfills all your needs in term of debuging in your Unity Engine journey.

Patch Note

This section is the log of all the changes and additions to UDT since release.

V 1.0

UDT Release :

- In game Console
- In game gizmos
- In game FreeCam
- In game basic metrics