Draw XXL

Getting Started Guide:
A step-by-step tutorial of how to use Draw XXL
2023-03-18

Thank you for purchasing Draw XXL. I hope it will be useful for you. This document explains the first steps on how to get started with the new asset. It contains these three sections:

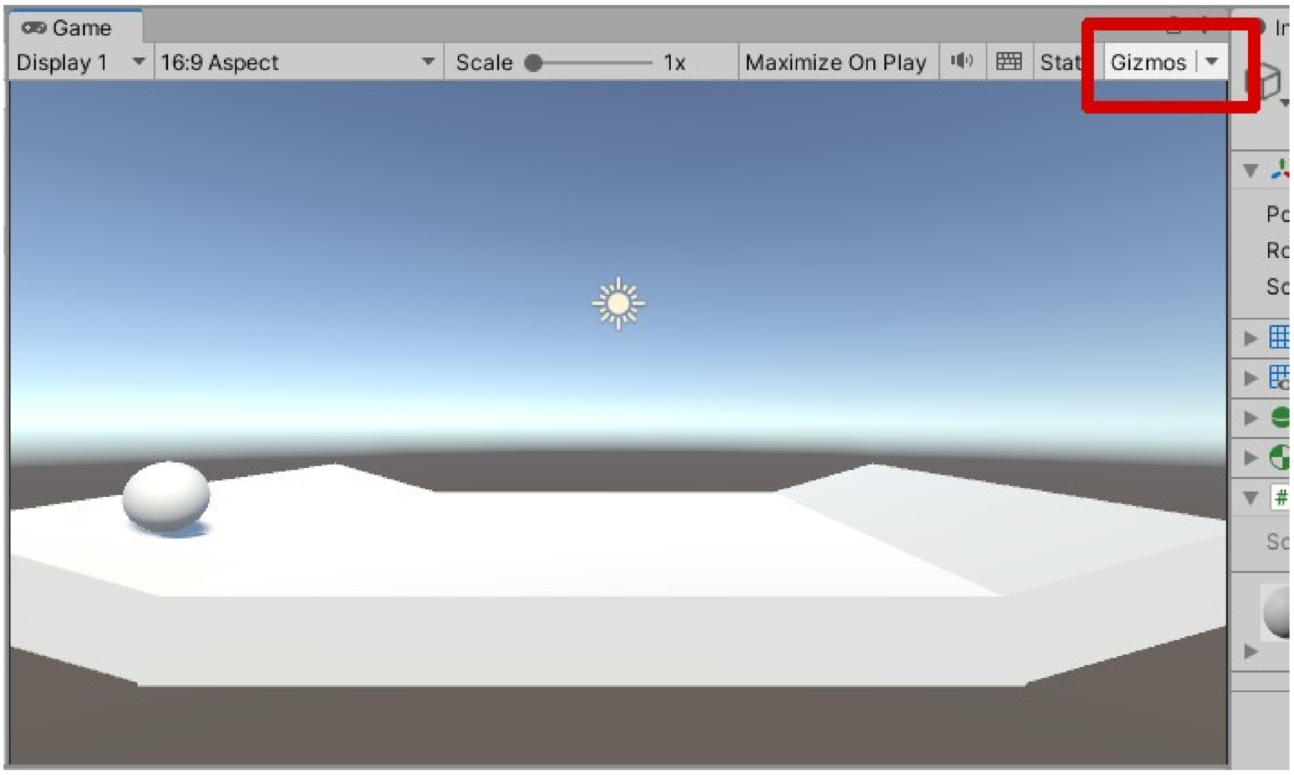
1. Getting Started

Before anything else:
If the asset doesn't work as expected, if you get stuck somewhere in the tutorial or if you have questions of any other kind:
Feel free to contact me. I'm glad to hear from you and are available via mail or Discord.

How to get started with the new asset? The first thing is (as with any other asset) to download and install it via the package manager to your project.

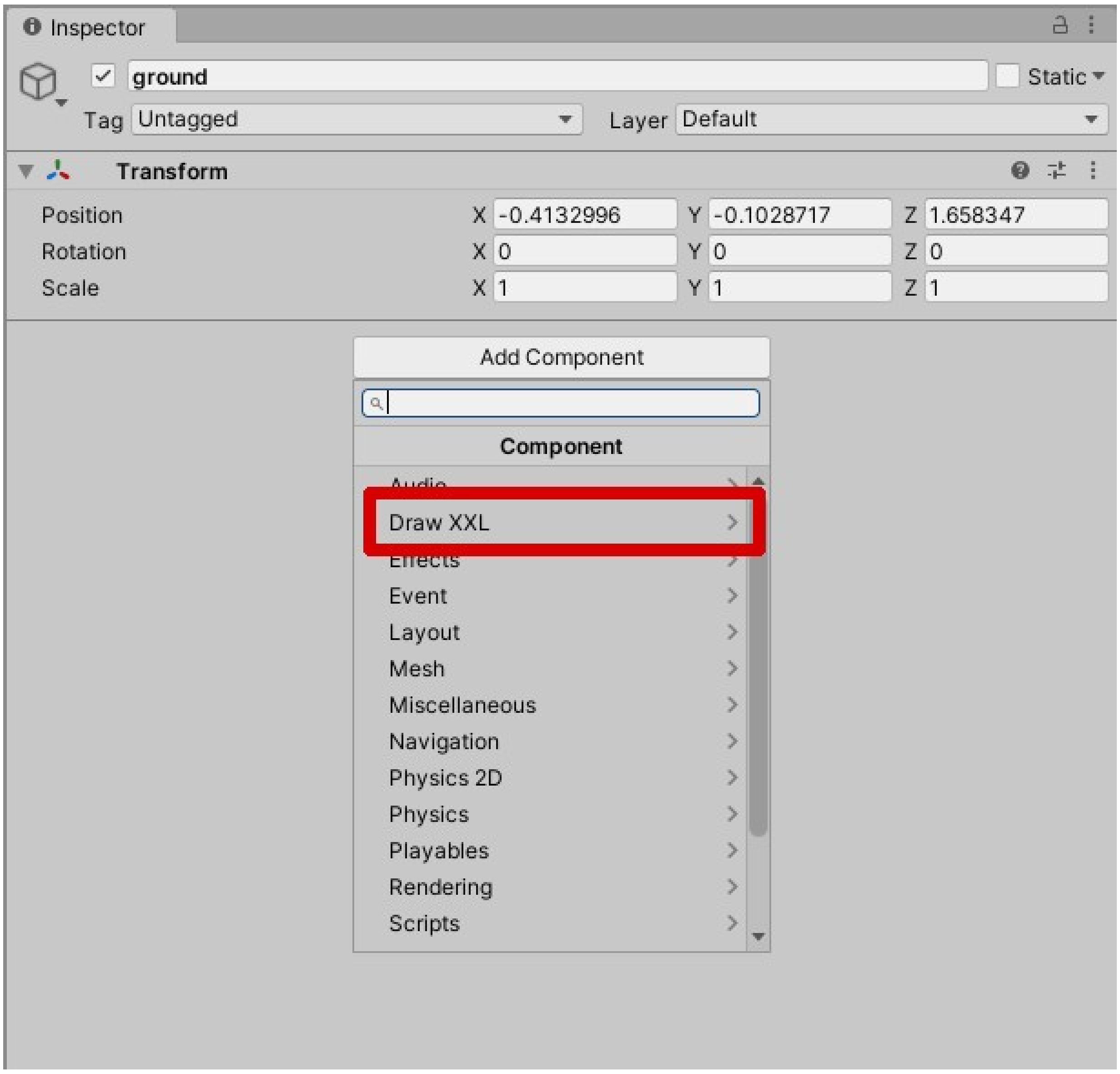
Now it is worth mentioning that Draw XXL can be used with two different approaches: Via *code* and via *components* (without any code). Both have its advantages and the next two tutorial chapters explain each of them.

Independent of which way you choose there is one thing to know before anything gets drawn to your screen: Draw XXL by default uses Unitys build in Debug.DrawLine() function under the hood. That means you have to enable gizmo drawing for the desired view panels (Scene View and/or Game View) in the Unity Editor. This is done by toggling the gizmo button in the top right corner of the Editor panel:

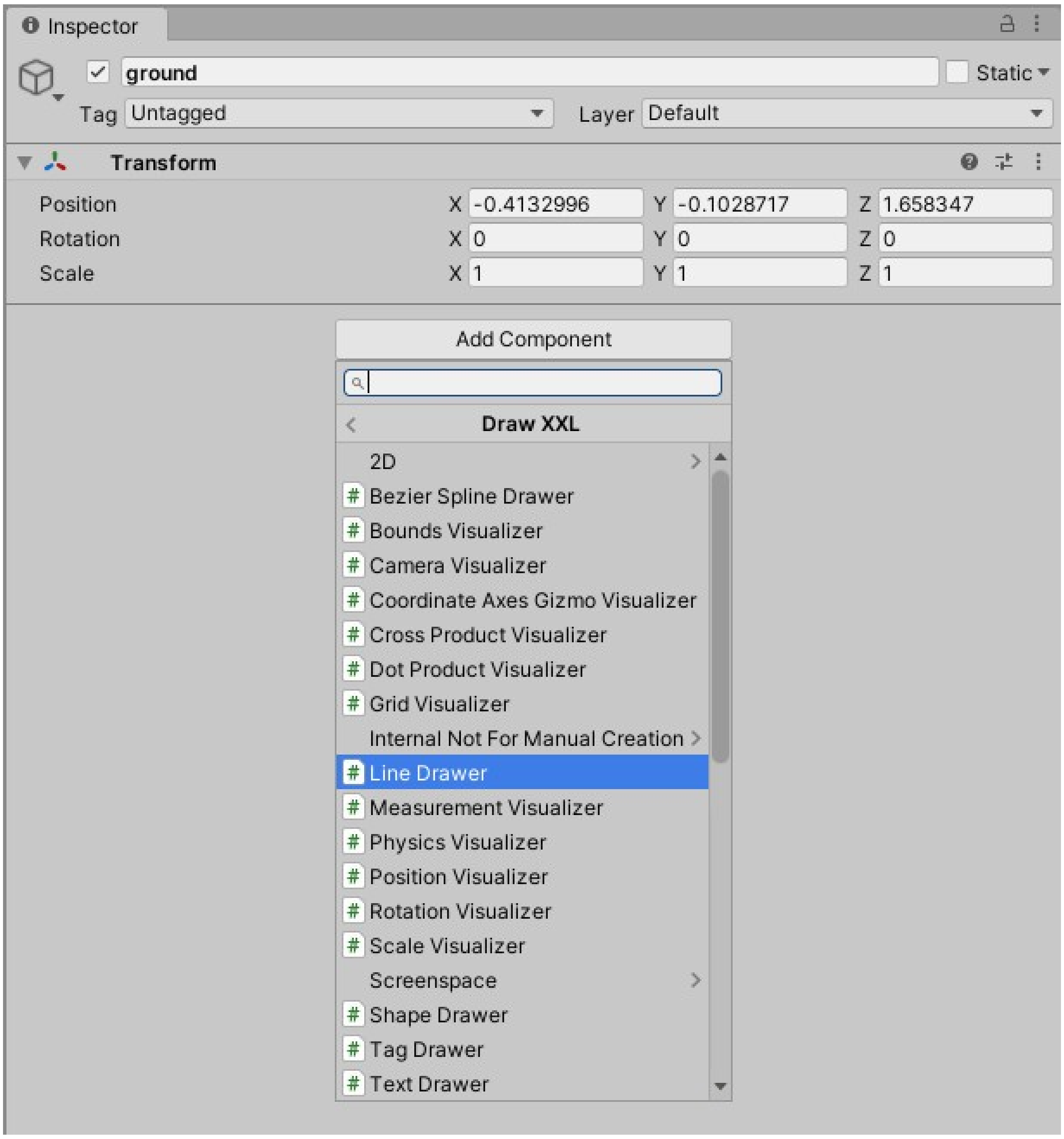


2. Drawing without code

An easy way of using Draw XXL is via components. Draw XXL offers a range of drawer components that can be added to Gameobjects just like you would add other basic components like a Rigidbody or a Collider. Provided you have the Draw XXL asset imported to your project you will find a new folder in the "Add component" menu named "Draw XXL".



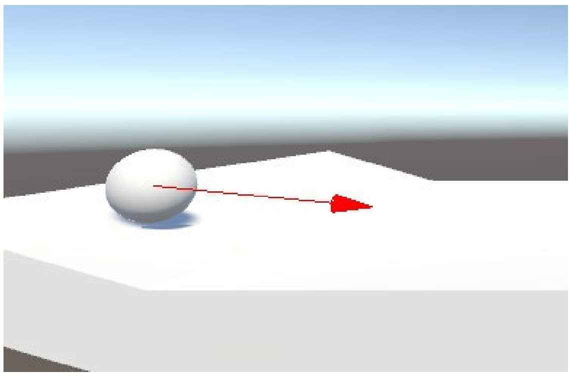
In this folder you will find the new available components for beeing added with only one further mouse click. The straightforward way of learning which options are available is to just browse through this "Draw XXL" subfolder and have a look at the names of the components or add some of them testwise to see what they do and which tweaking options they offer in the inspector.



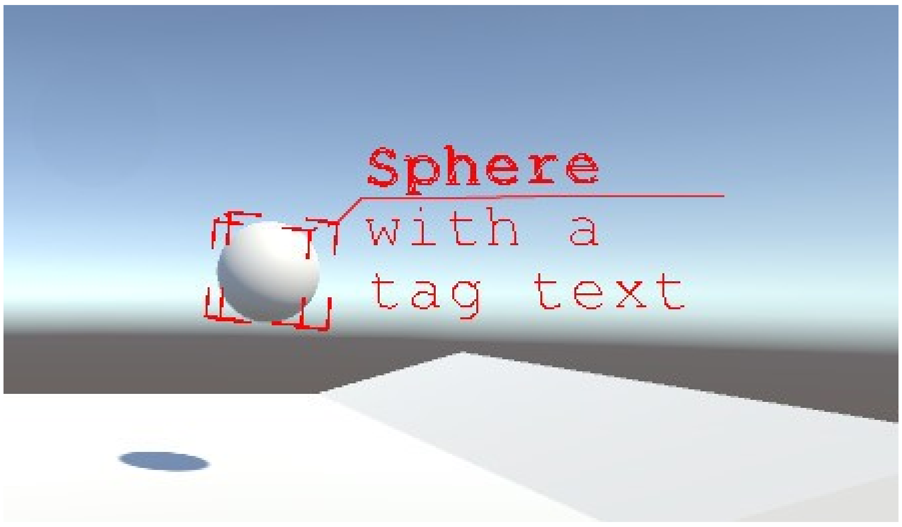
Position	X -0.4132996	Y -0.1028717	Z 1.658347
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Here are some usage examples:

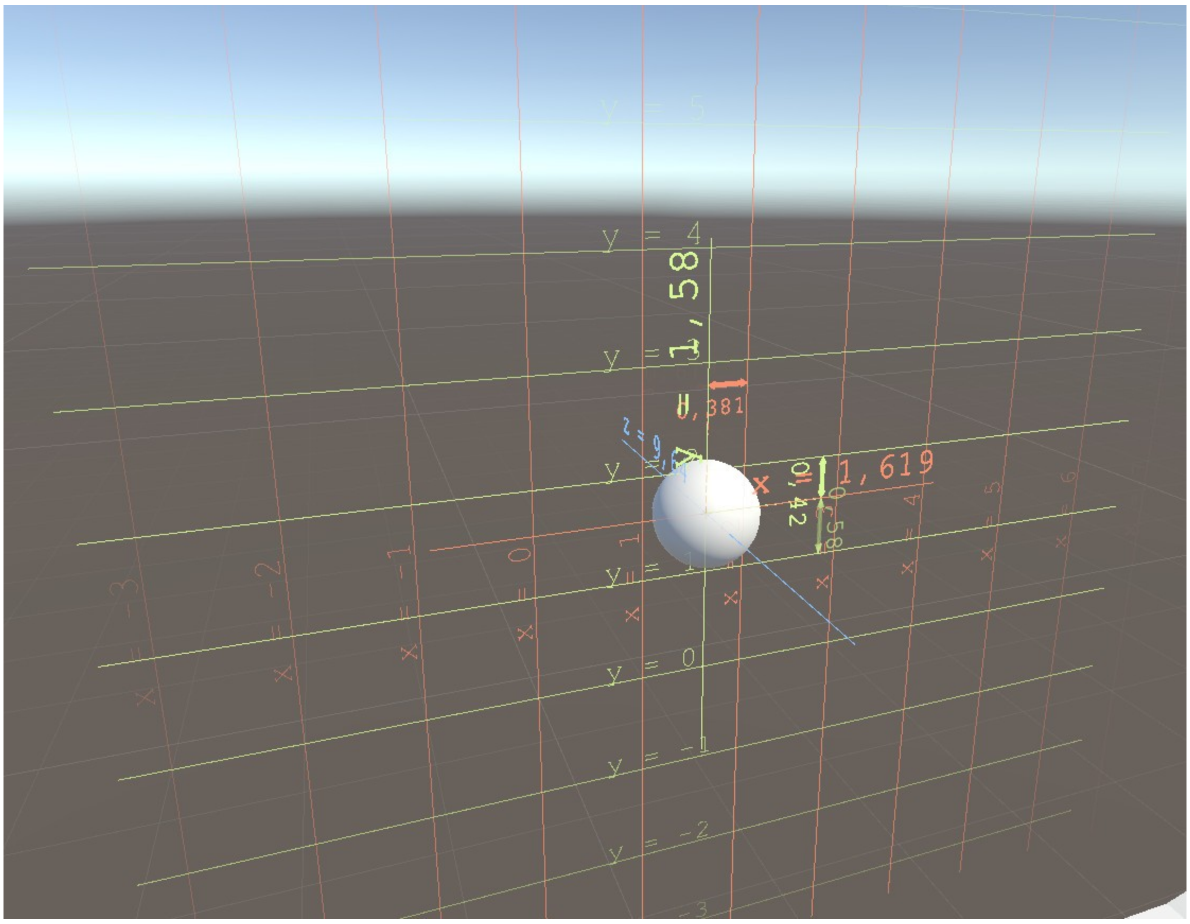
- Using a "Line Drawer" component to always see a GameObjects forward direction, also if it is not selected:



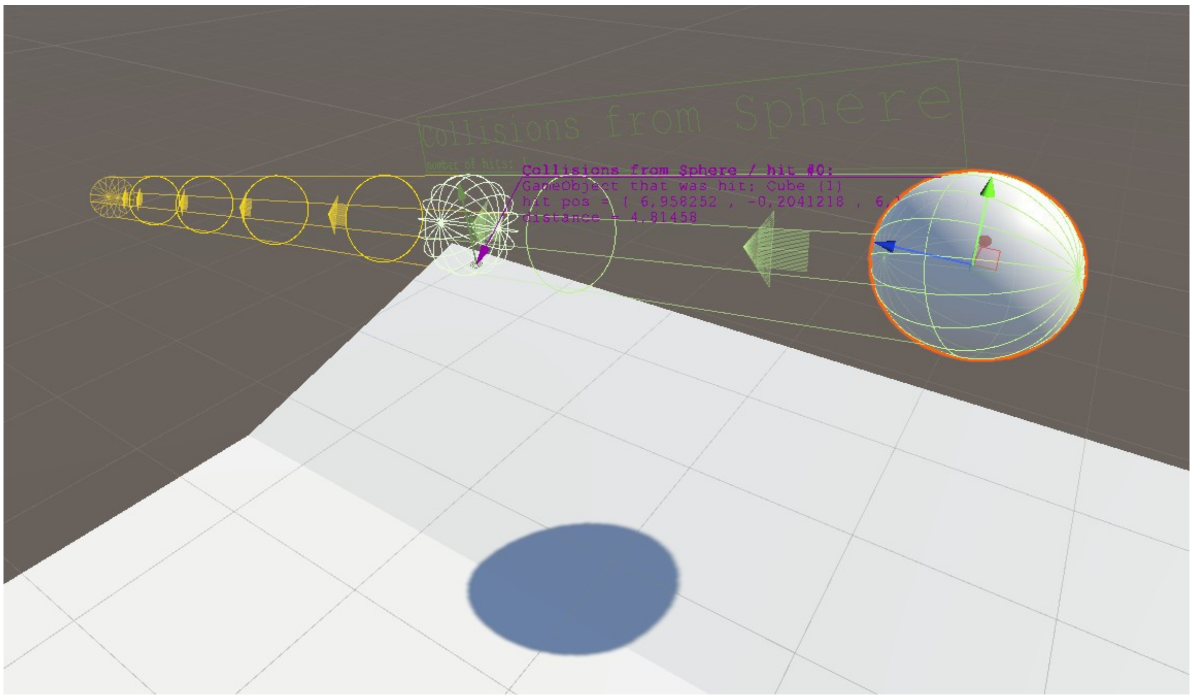
- Adding a tag text to a GameObject via a "Tag Drawer":



- Display the world grid around a GameObject via a "Grid Visualizer":



- Casting physics shapes through the scene with a "Physics Visualizer" and adjust the cast direction with the transform handle:



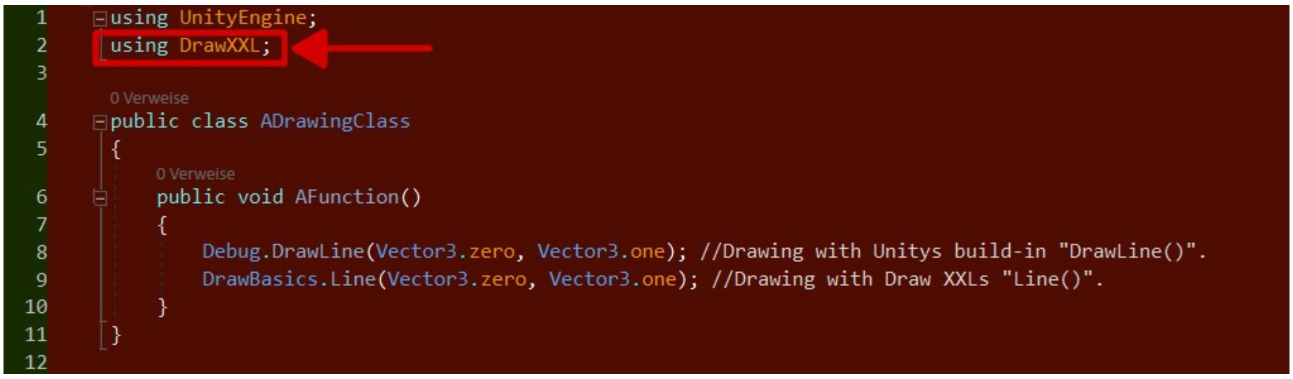
```
Collisions from Sphere / hit #0:
GameObject that was hit: Cube (1)
hit pos = ( 6,958252 , -0,2041218 , 6,
distance = 4,81458
```

The components have many options for tweaking and customization in the inspector. The options are mostly self-explanatory or are explained via tooltips. For example the "Shape Drawer" component hosts over 20 different shapes to choose from.

3. Drawing with code

The full potential and flexibility of Draw XXL is leveraged when drawing from code. Draw XXL offers a static drawing library where you can call drawing functions from anywhere in your code without initializing objects - as known from using Unitys Debug.DrawLine().

All you have to do to get started is declaring the usage of the Draw XXL namespace on top of your scripts, like so:



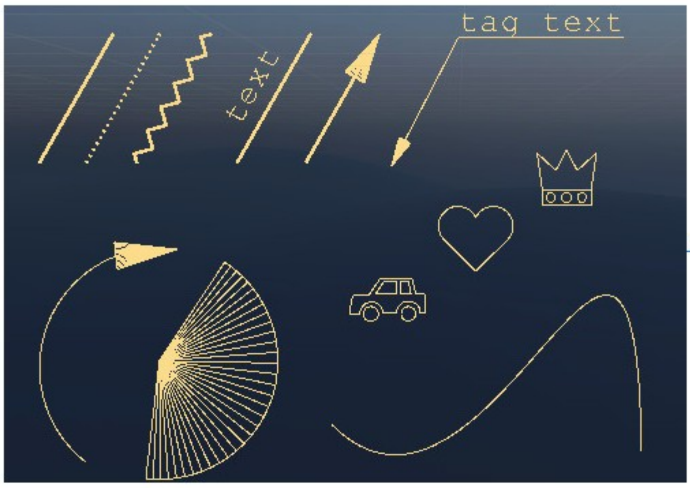
```
using UnityEngine;
using DrawXXL;

0 Verweise
public class ADrawingClass
{
    0 Verweise
    public void AFunction()
    {
        Debug.DrawLine(Vector3.zero, Vector3.one); //Drawing with Unitys build-in "DrawLine()".
        DrawBasics.Line(Vector3.zero, Vector3.one); //Drawing with Draw XXLs "Line()".
    }
}
```

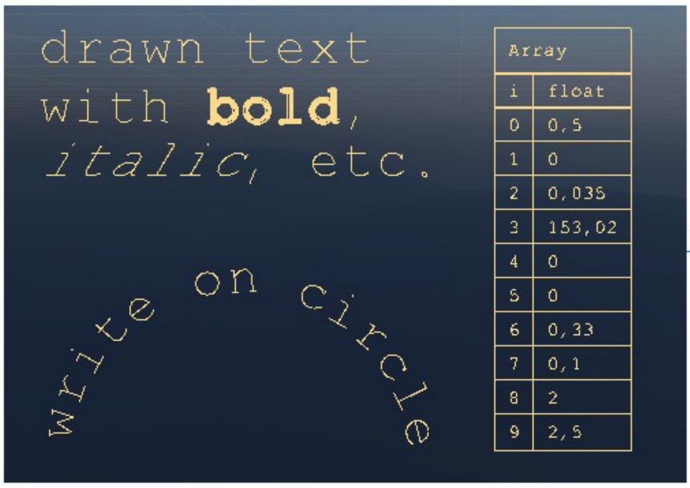
(continue on next page)

Now there is the question how to find out which draw functions Draw XXL offers and what their names and signatures are. This is where the API documentation comes into play. The API documentation is your starting point on the journey of learning the numerous available options. To make navigation easier every category and every drawn object is visualized with an image. So you can literally browse the API documentation "by image" instead of the normal approach of "by class and function names".

Let's go through an example:
On the API overview page you see different preview images that represent different shape categories, like so:

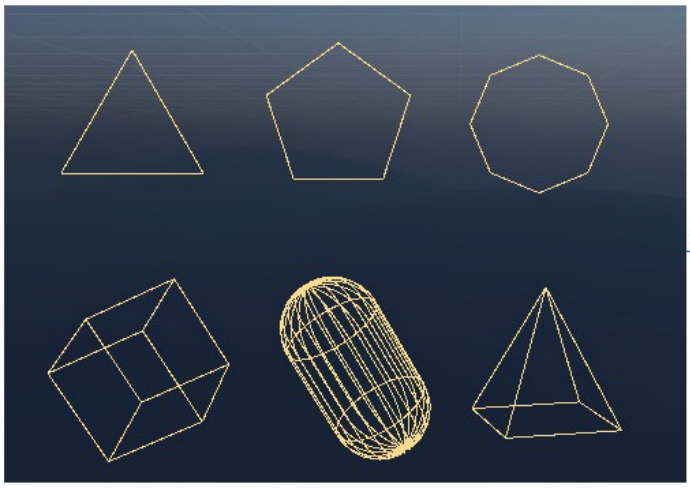


Draw Basics 2D



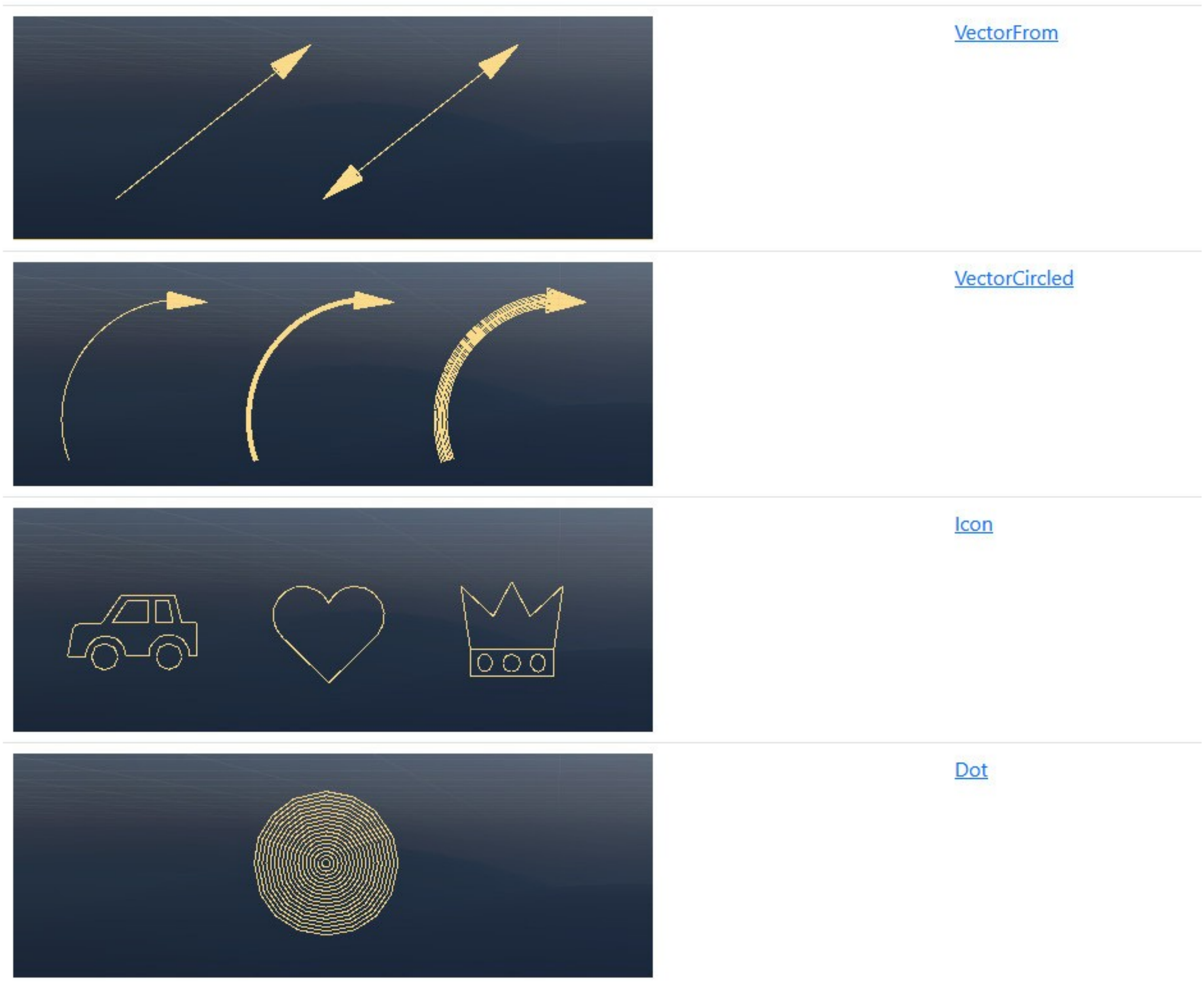
Array	
i	float
0	0,5
1	0
2	0,035
3	153,02
4	0
5	0
6	0,33
7	0,1
8	2
9	2,5

Draw Text



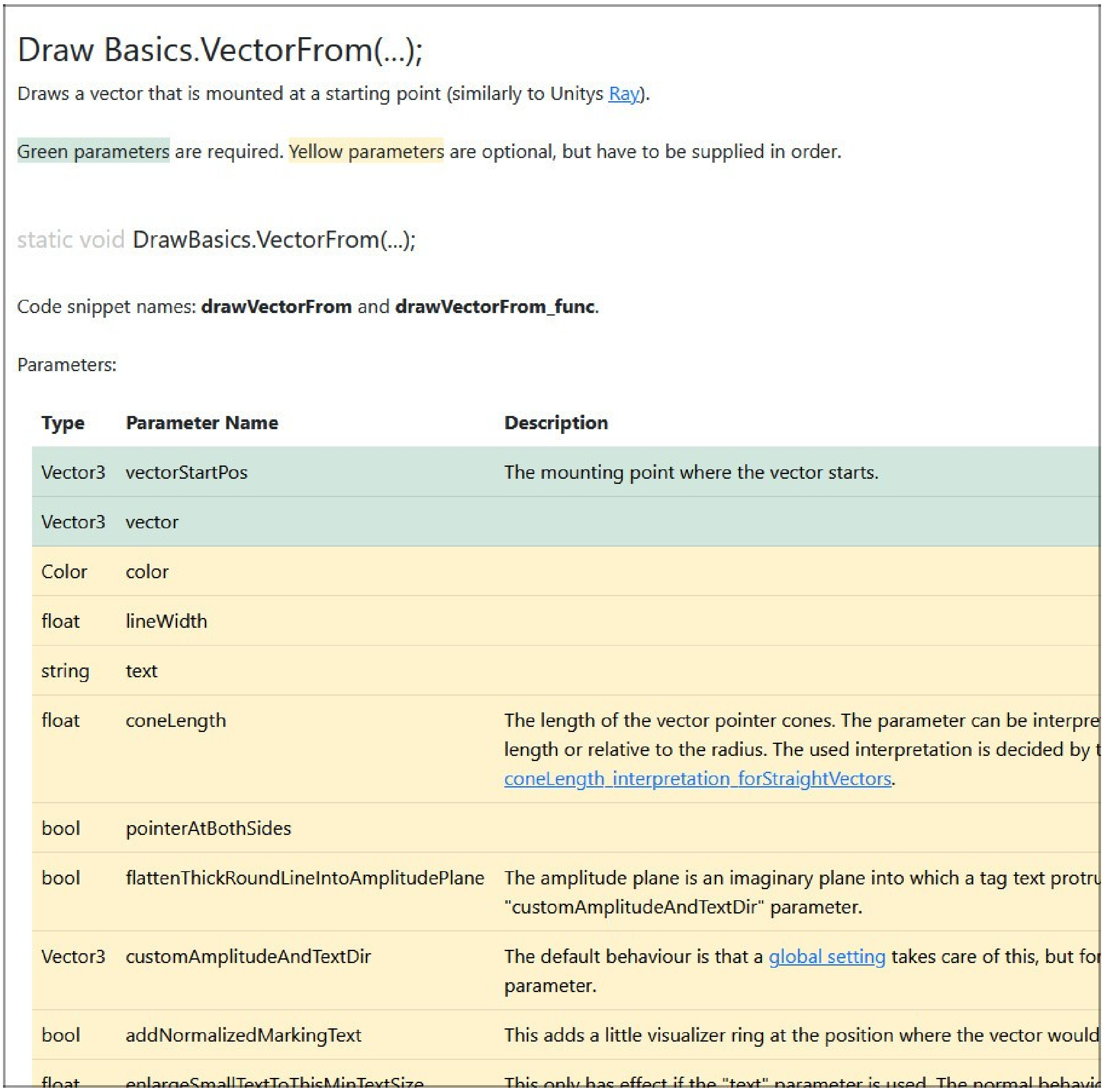
Draw Shapes

Now say you want to draw a vector to visualize the forward direction of a GameObject. On the API overview page you see that the category "Draw Basics" has a vector. After clicking on the Draw Basics category another page with many images opens. The images represent all shapes that you can draw inside the Draw Basics category. Again you can scroll up and down to see what drawing options are available. While scrolling you see the wanted vector:



	VectorFrom
	VectorCircled
	Icon
	Dot

Let's choose "VectorFrom". After clicking on VectorFrom the actual description of the vector (function name and signature) opens: (see next side)



Draw Basics.VectorFrom(...);

Draws a vector that is mounted at a starting point (similarly to Unitys Ray).

Green parameters are required. Yellow parameters are optional, but have to be supplied in order.

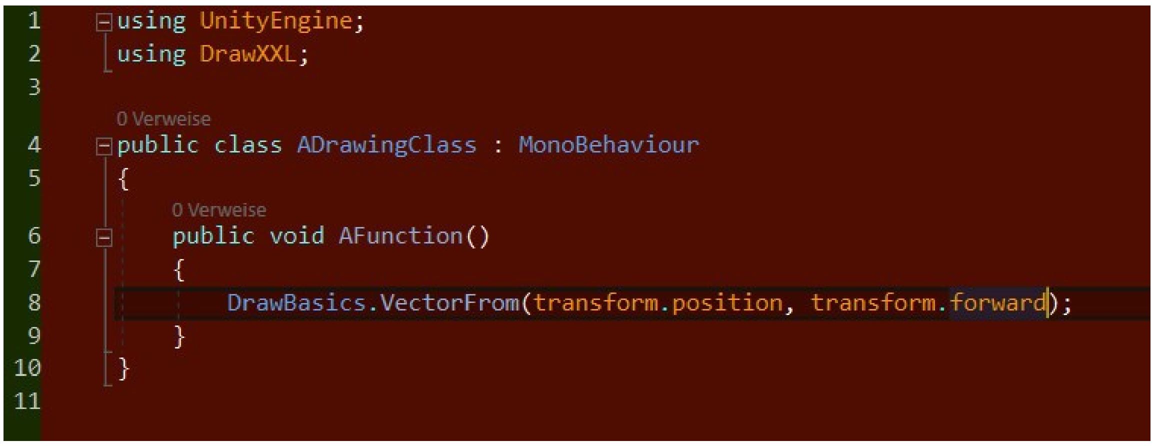
static void DrawBasics.VectorFrom(...);

Code snippet names: **drawVectorFrom** and **drawVectorFrom_func**.

Parameters:

Type	Parameter Name	Description
Vector3	vectorStartPos	The mounting point where the vector starts.
Vector3	vector	
Color	color	
float	lineWidth	
string	text	
float	coneLength	The length of the vector pointer cones. The parameter can be interpre length or relative to the radius. The used interpretation is decided by t coneLength_interpretation_forStraightVectors.
bool	pointerAtBothSides	
bool	flattenThickRoundLineIntoAmplitudePlane	The amplitude plane is an imaginary plane into which a tag text protru "customAmplitudeAndTextDir" parameter.
Vector3	customAmplitudeAndTextDir	The default behaviour is that a global setting takes care of this, but fo parameter.
bool	addNormalizedMarkingText	This adds a little visualizer ring at the position where the vector would
float	enlargeSmallTextToThisMinTextSize	This only has effect if the "text" parameter is used. The normal behavi

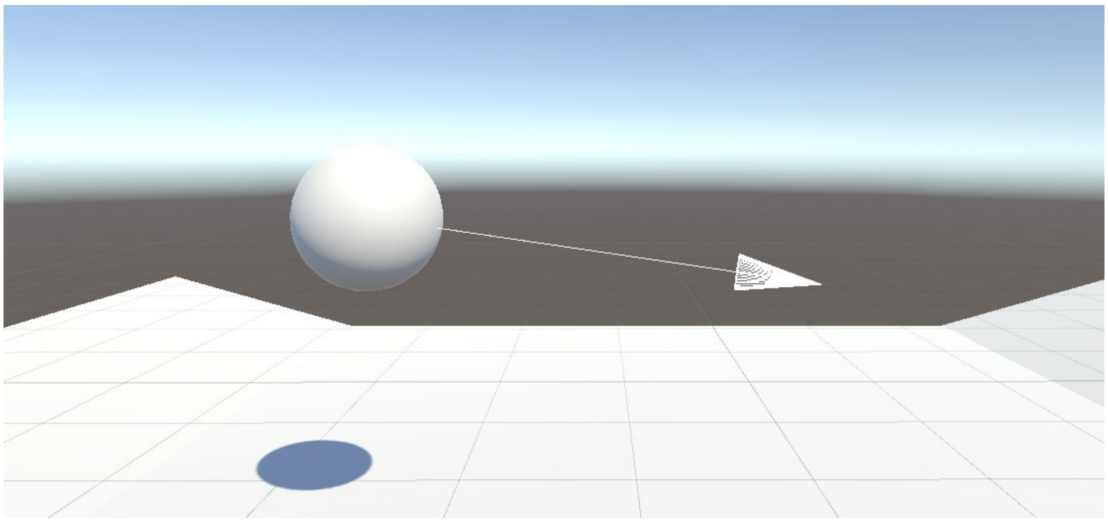
We can see from the function description the name of the vector draw function and the expected parameters. There are two mandatory parameters: "vectorStartPos" and "vector". These mandatory parameters are colored green in the API documentation. With that information we can craft our first drawing call to the Draw XXL library, like so:



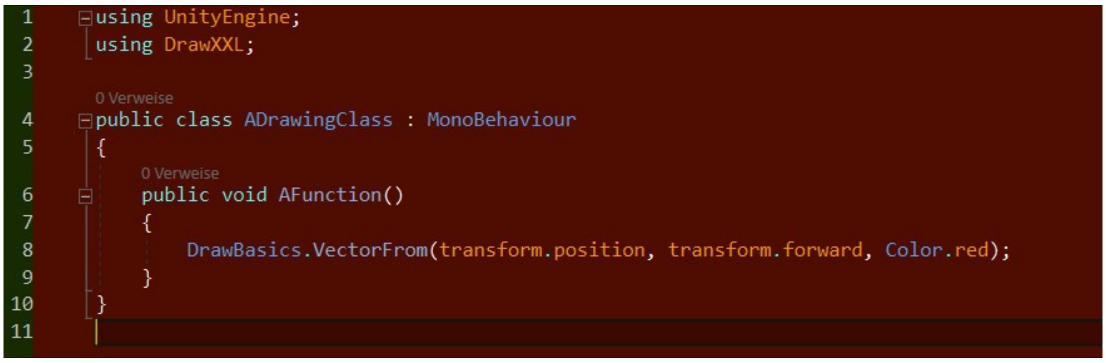
```
using UnityEngine;
using DrawXXL;

0 Verweise
public class ADrawingClass : MonoBehaviour
{
    0 Verweise
    public void AFunction()
    {
        DrawBasics.VectorFrom(transform.position, transform.forward);
    }
}
```

In the Unity Editor something like this should appear:



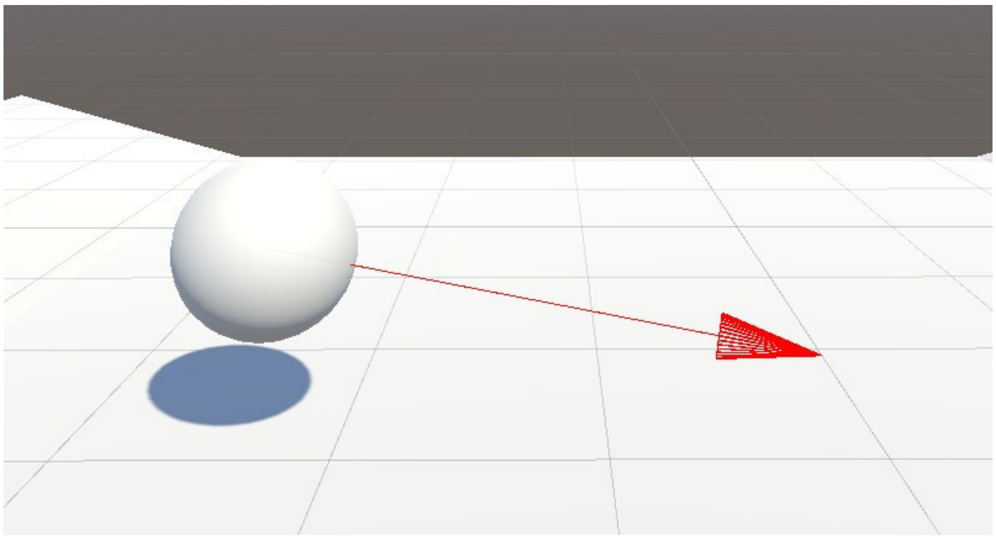
We already saw that green parameters from the function signature are mandatory. But what about the yellow parameters? Yellow parameters are optional, but have to be supplied in the correct order. They are used to further customize the drawn shape. Draw XXL makes heavy use of such optional parameters. In many cases they can just be ignored (since they are only optional), but in other cases it is good to have them and be able to tweak a drawn shape. Let's use some optional parameters to tweak our vector.
The first optional parameter is "color". We can extend our code line like this:



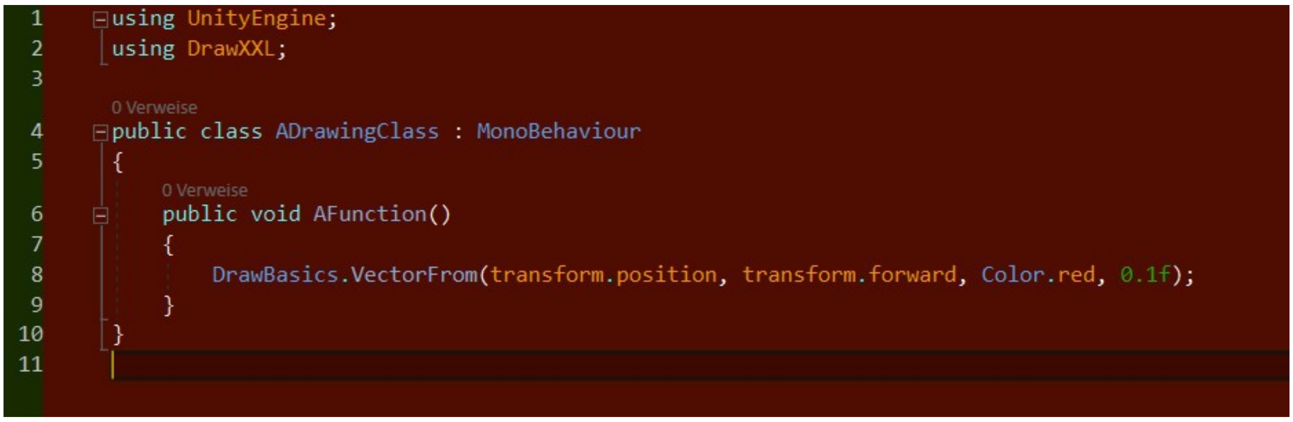
```
using UnityEngine;
using DrawXXL;

0 Verweise
public class ADrawingClass : MonoBehaviour
{
    0 Verweise
    public void AFunction()
    {
        DrawBasics.VectorFrom(transform.position, transform.forward, Color.red);
    }
}
```

This should color our vector like so:



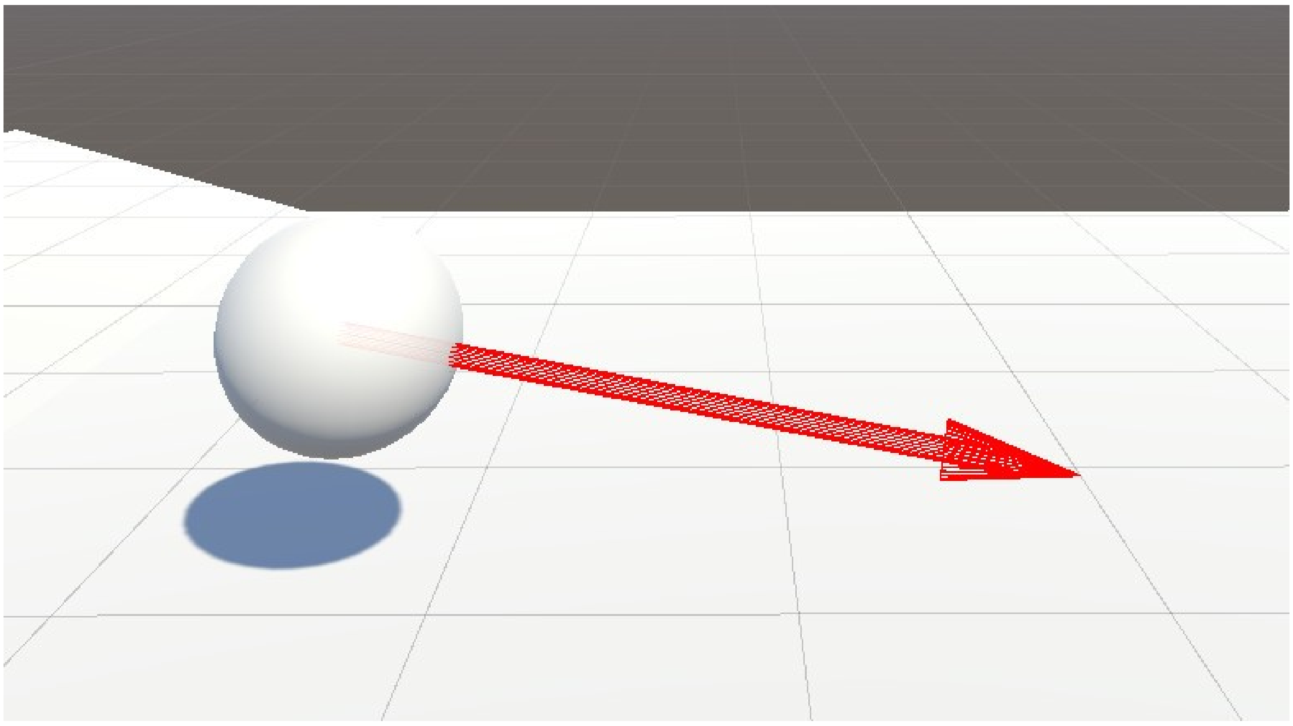
In busy environments drawn shapes are sometimes not well recognizable. An easy way to fix this is to raise the line width of the vector. "lineWidth" is the next optional parameter. Our code is then like so:



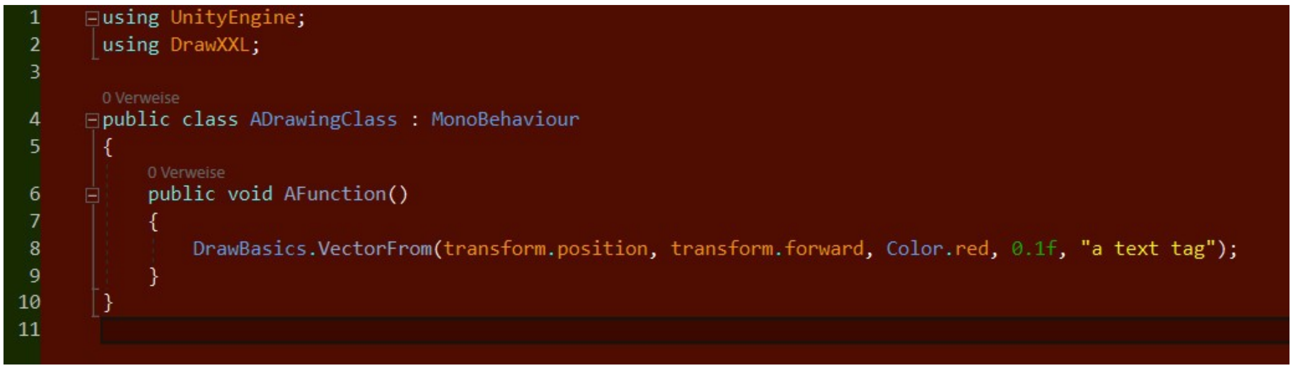
```
using UnityEngine;
using DrawXXL;

0 Verweise
public class ADrawingClass : MonoBehaviour
{
    0 Verweise
    public void AFunction()
    {
        DrawBasics.VectorFrom(transform.position, transform.forward, Color.red, 0.1f);
    }
}
```

And in the Editor it looks like this:



Finally we can add a text tag to the vector. "text" is the next parameter.



```
using UnityEngine;
using DrawXXL;

0 Verweise
public class ADrawingClass : MonoBehaviour
{
    0 Verweise
    public void AFunction()
    {
        DrawBasics.VectorFrom(transform.position, transform.forward, Color.red, 0.1f, "a text tag");
    }
}
```

The result: A vector with text tag: