

# Immediate Mode Draw v1.4.4

## User Guide

### Overview

Immediate Mode Draw (IMDraw) for Unity is an API which enables drawing of a variety of primitives and text labels at run time. Debug visualisation is a common requirement of many projects. This asset is geared towards developers who want debug rendering that is convenient and efficient.

Once the simple installation and set-up has been completed, you are ready to start using IMDraw right away by simply issuing draw calls from your scripts. For example, to draw a wireframe box:

```
void Update ()
{
    IMDraw.WireBox3D(
        transform.position,
        transform.rotation,
        transform.scale,
        Color.green);
}
```

### Features

- Easy to use, trivial to set up and very flexible.
- Support for solid, wire frame, line and label primitives.
- Options for culling and fading based on distance from camera.
- Set limits on the maximum number of vertices that can be rendered.
- Gizmo extension API that provides extended functionality over the standard Unity gizmo class.
- Optimized to reduce its impact on your project.
- No garbage generation.
- Includes full source code, examples and documentation.
- Fully compatible with Unity 5.0.0f4 and higher.
- Support for scriptable render pipeline (both URP and HDRP).
- Intended for use on all platforms.
- Dedicated support via e-mail or forum.

## Setup

Once the package is installed, follow these easy steps:

1. Import the package via the asset store or via *Assets > Import Package > Custom Package* if you have the package file.
2. Add an *IMDrawManager* component to the scene. You may add this to a new game object or any other game object, so long as the component is active.
3. Add an *IMDrawCamera* component to the Camera that you wish to draw to.

If you wish to use *IMDraw* to draw to different cameras:

1. Add an *IMDrawCamera* component to each Camera game object that you wish to draw to.
2. Note that the priority of each *IMDrawCamera* affects the draw order of labels.
3. When drawing, use *IMDraw.SetTarget (IMDrawCamera camera)* before you issue a draw request to set which camera you are currently drawing to.

Important notes:

- You only require one active *IMDrawManager* component in the scene for drawing to work. If more than one *IMDrawManager* component exists, only one component will be used.
- *IMDrawCamera* can only be added to game objects with a Camera component on them.

## Documentation

Three pieces of documentation are included with *IMDraw*:

- This user guide.
- Reference guide which contains documentation for the API.
- Release notes.

## Examples

Included with the package is a folder called *Example* which contains an example scene. This sub-folder of the asset is purely optional and is not required for *IMDraw* to function. The example scene demonstrates the following:

- A demonstration of all available primitives.
- An example physics scene to demonstrate how *IMDraw* can be used to visualise colliders and contact points.
- A diagnostic test of API draw functions.

## IMDraw Usage

Immediate Mode Draw has been designed to catch draw requests from as many places as possible. The API is designed to be robust against situations where IMDrawManager or IMDrawCamera are either missing or disabled.

For a full list of all API functions please refer to the included IMDraw API Reference document.

Please note that primitives and labels are rendered in the following order:

- Mesh primitives (solid meshes and text meshes).
- Line primitives (including wireframe shapes).
- World space labels (also known as 3D labels).
- Screen space labels (also known as 2D labels) and 2D rectangles.

For line, mesh and text mesh primitives you can define the ZTest operation used to render them. This is useful in situations for example where you would want to certain things to render over the top of objects in the scene.

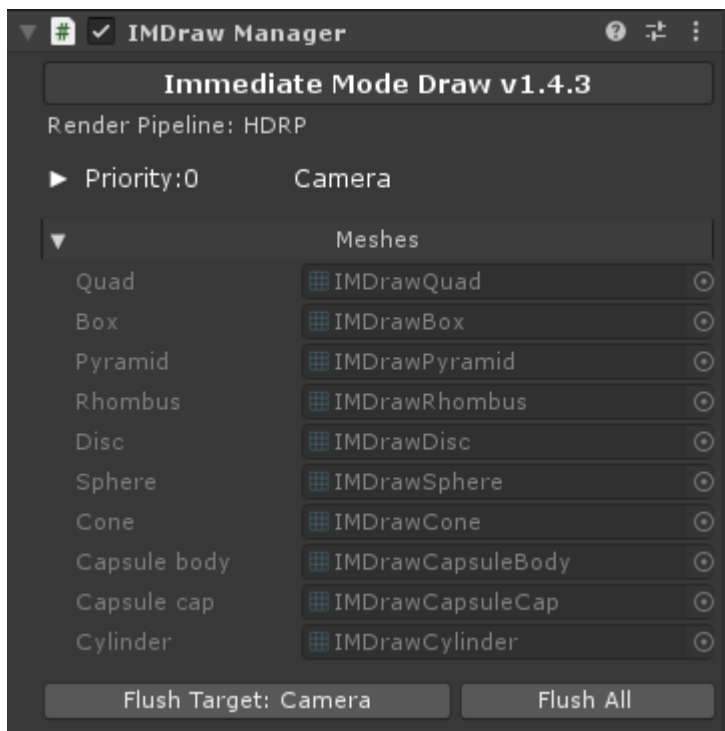
```
// Set subsequent draw operations to ZTest always
IMDraw.ZTest = IMDrawZTest.Always;

// This box will draw with over the top of everything in the scene
IMDraw.WireBox3D(Vector3.zero, Vector3.one, Color.white);

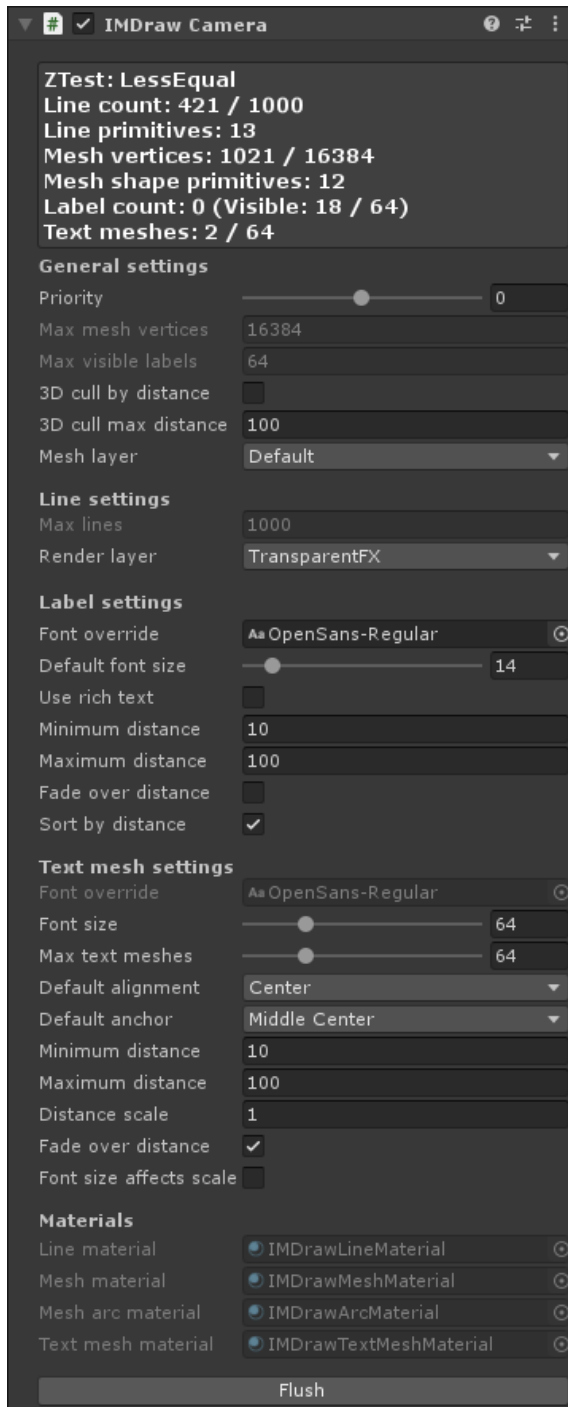
// Restores default to ZTest less or equal
IMDraw.SetDefaultZTest();
```

The following Z test operations are available:

Disabled, Never, Less, Equal, LessEqual, Greater, NotEqual, GreaterEqual, Always



- The *IMDrawManager* component must exist in the scene for the IMDraw to work.
- In **Unity 2017.1 and above**, the render pipeline can be selected. The appropriate render pipeline must be selected for IMDraw to function correctly. By default, it is set to auto detect. Should this fail, you can manually select the render pipeline that is used by your project. This cannot be altered during play in editor.
- At run-time, the inspector provides a list of all *IMDrawCamera*'s that are active. For each camera the following is shown:
  - The camera which is currently the target for draw calls (shown with the ► symbol).
  - The draw priority. If you have more than one *IMDrawCamera*, then the order of cameras affects the order in which labels are drawn.
  - The name of the game object which the *IMDrawCamera* component is attached to.
  - A "Set as target" button will appear for *IMDrawCamera* components which aren't active.
- The meshes list contains references to mesh assets that are using for solid primitive rendering. These are included and assigned automatically for you, however you can override them if you wish.
- Buttons for flushing the target camera or flushing all cameras. These buttons only appear them the project is running.



*Note: All properties have tooltips which explain what they do.*

- An *IMDrawCamera* must be added to the camera which you wish to draw to.
- When the project is running, stats will appear in the inspector to show what resources are currently being used by that component.
- General settings:
  - Priority – affects the draw order of labels if there is more than one *IMDrawCamera* in the scene. Lower priority will be drawn before higher priority.
  - Max mesh vertices – places a limit on the number of vertices used for mesh primitives.
  - 3D cull by distance - enable/disable culling of primitives based on their distance from the camera.
  - 3D cull max distance – the distance at which line, wireframe and solid primitives will be culled (if the above option is enabled).
  - Mesh layer – the target layer for rendered meshes.

- Line settings:
  - Max lines – the maximum number of lines that can be rendered. For line-based primitives such as wire spheres, drawing will be skipped if attempting to draw them puts the number of lines over this limit.
  - Render layer – the render layer used for line rendering (only applicable when using mesh-based rendering).
- Label settings:
  - Font override - Specify a font which is used by all labels for this camera. If no font is specified, the default Unity font is used.
  - Default font size – The default font size used by all labels for this camera.
  - Use rich text - Enable/disable HTML-style tags for text formatting mark-up.
    - Supported tags: <color> <size> <b> <i>
    - Note: size, bold and italic tags require the font to use dynamic font rendering.
  - Distance fade - Enable/disable fading of 3D positioned labels based on their distance from the camera.
  - Minimum/maximum distance – the distances between which 3D labels will be faded. Beyond the maximum distance, 3D labels will be at zero opacity and therefore their rendering will be skipped.
  - Sort by distance - enable/disable draw order sorting based on the distance of a 3D label from the camera. When enabled, the 3D labels which are closest to the camera will be drawn on top of other labels.
- Text mesh settings:
  - Font – font used by text mesh primitives.
  - Font size – font size. Note: increasing the font size and scaling down the text can improve the quality of the font rendering at the cost of font texture size.
  - Max text meshes – Maximum number of text meshes that can be drawn at the same time.
  - Minimum distance – Minimum fade distance (if fade over distance is enabled).
  - Maximum distance – Maximum draw distance for text meshes.
  - Distance scale – Scaling applied to text meshes that used fixed scale.
  - Fade over distance - Enable/disable fading of text mesh based on distance from the camera within the minimum and maximum distance.
  - Font size affects scale – Specifies if changing the font size affects scale. Useful in situations where you want to improve font texture resolution without making the text larger.
- Materials:
  - Materials for line, mesh and text mesh primitives can be assigned (when the project isn't running). These are required for respective primitive types to render.
- If the project is running, a "Flush" button will appear. Clicking this will flush all draw calls for that button.

## Limitations

- IMDraw will work when used from *OnGUI* but be aware that *OnGUI* is called multiple times per frame which will result in excess draw calls.
- Drawing in *FixedUpdate* is possible however keep in mind that if the *Time.fixedDeltaTime* is lower than *Time.deltaTime* then it means that *FixedUpdate* is going to be called fewer times per second than *Update*. This means *FixedUpdate* will be skipped some frames, therefore attempting to draw in this situation will result in apparent flickering. When drawing elements for physics objects (such as colliders), the best solution is to draw in the *Update* function.
- Solid shape primitives are not batch rendered. This is due to a limitation with Unity where *Graphics.DrawMesh* calls are not batched. This function is used to ensure solid shape primitives that may be transparent are correctly sorted in the scene.
- GUI primitives (labels and 2D rectangles):
  - Labels are rendered using the Unity built-in *IMGUI*. Therefore they are subject to the limitations that come with that come with *IMGUI* such as no batching of draw calls and the overhead of Unity's *IMGUI* system.
  - Labels and 2D rectangles do not work in VR, since they use Unity GUI to render (which currently appears to be unsupported). For text rendering, text mesh can be used instead.
- Draw requests are currently not thread safe and must be issued on the main thread. This may change for future versions.
- IMDraw is designed to continue functioning in editor if your project is recompiled whilst it is playing. Keep in mind however that a recompile will cause any outstanding draw commands to be flushed due to scripts being reloaded.

## Scriptable render pipeline

- IMDraw supports scriptable render pipeline.
- On applicable versions of Unity, the *IMDrawManager* component has a render pipeline selector. By default, it is set to auto detect but can be manually set to legacy, URP or HDRP render pipelines.

## Working on mobile

IMDraw is designed to work on any platform but be aware of the following when using it on mobile:

- Transparency is generally quite expensive on mobile platforms. If this is an issue, try modifying the shaders used by IMDraw to render as opaque.
- Mobile platforms are sensitive to large numbers of draw calls. Be aware that each solid primitive or label counts as one or more draw call. Wire frame and line primitives are however batched into a single draw call.

## Using IMDraw with UnityScript (a.k.a. JavaScript)

In order for IMDraw to work with UnityScript, the IMDraw folder must be placed in a folder called **Plugins** so that the folder path looks like this: **Assets/Plugins/IMDraw**. The reason for this is because UnityScript is compiled before C#. For further information, see: <http://docs.unity3d.com/Manual/ScriptCompileOrderFolders.html>.

## IMGizmos Usage

IMGizmos is an API for drawing a gizmos. Gizmos are used to give visual debugging or setup aids in the scene view.

These functions may be used in `MonoBehaviour.OnDrawGizmos` and `MonoBehaviour.OnDrawGizmosSelected`.

An example scene is included which demonstrates the usage of this API.

## Troubleshooting

**Question:** `IMDraw` is not correctly being rendered at the correct depth.

**Solution:** Pay careful attention to your camera setup and what camera your *IMDrawCamera* component is attached to. Ensure that depth isn't being cleared between when your scene is rendered and when `IMDraw` is rendered.

**Question:** I have moved `IMDraw` from its default *Assets/IMDraw* path and now it no longer works.

**Solution:** In order for `IMDraw` to work effectively, some file paths need to be hard coded. You will need update the following strings to reflect the new path of `IMDraw`:

In *IMDrawManager.cs*, modify *IMDrawManagerEditor.MESH\_ASSET\_PATH*.

In *IMGizmos.cs*, modify *IMGizmos.MESH\_ASSET\_PATH* and *IMGizmos.IMGIZMOS\_MATERIAL\_PATH*.

**Question:** Labels and 2D rectangles do not work in VR.

**Answer:** Labels and 2D rectangles use Unity GUI to render. Unfortunately anything that is drawn by Unity GUI API does not show up in VR and is presumably unsupported. For text rendering, please use text mesh API functions instead.

**Question:** I have a problem that is not mentioned in this documentation.

**Solution:** Please do not hesitate to report your issue to either [imdraw@harveyc.net](mailto:imdraw@harveyc.net) or alternatively by posting in the [IMDraw forum thread](#). In your report, please include the following:

- What version of Unity you are using.
- What operating system you are using.
- What platforms/environments you are experiencing problems in. E.g. editor, release build, Android, iOS.
- What error messages you get in the debug log.
- What error messages you see in `IMDrawManager` and `IMDrawCamera` components.
- What parts of the `IMDraw` API are not working.
- What parts of the `IMGizmos` API are not working.