

# **Unity uGUI Advanced Reference**

(English version)

By: Heppoko

# Introduction

## Who this book is for

This book is intended for readers who wish to gain a deeper understanding of the basic principles of Unity's uGUI. If you are expecting some quick tips and useful UI assets, you may be disappointed. But have you ever experienced something like this?

- Some useful UI assets are not supported on certain platforms.
- Some useful UI assets stopped working after upgrading Unity version.
- Some useful UI assets stopped working even though I didn't do anything to them!

By understanding the contents of this book, you will be able to respond calmly when you encounter such a situation.

The contents of this book should also be very useful for performance improvement. For example, there are several known techniques to improve uGUI performance, including the following.

- Split Canvases.
- Reduce the number of useless RaycastTargets.
- Avoid using Camera.main
- Try to avoid using LayoutGroup in Auto Layout.
- Don't animate the UI with Animation or Timeline

These are the most commonly known. After reading through this book, you will be able to understand the reasons why these methods are effective.

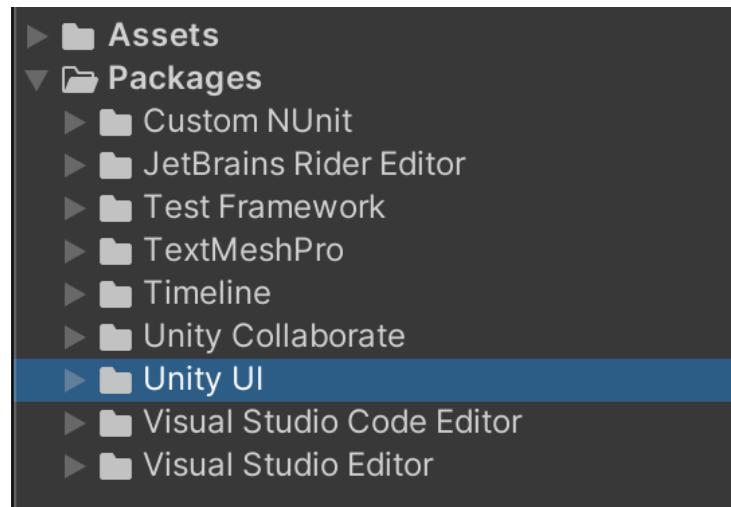
This book describes the properties and public methods of uGUI components. These explanations are not just a copy of the official reference, but the result of the author's verification by actually calling them after checking the source code. This should help you understand detailed behaviors that are not available in the official reference.

As a way to understand programming languages and middleware such as game engines in depth, it is very useful to read all the references and source codes. Reading through the whole book will give you a deep understanding of not only uGUI but also Unity in general.

## Accessing the uGUI source code

The source code of the C# part of uGUI is publicly available.

If you are using Unity 2019.2 or later, you can open *Unity UI* from *Packages* in the **Project** window to see the inside of the uGUI package, including the source code. If you put `Debug.Log()` or modify the source code, you need to follow the steps below.



1. Right-click on the **Packages/Unity UI** folder in the **Project** window and open it with *Show in Explorer* on Windows( or *Reveal in Finder* on MacOS).
2. This will open the folder where the Package is located, so copy the **com.unity.ugui@1.0.0** folder.
3. Paste **com.unity.ugui@1.0.0** into the **Packages** folder of your project (e.g. "`C:\MyProject\ Packages`").
4. Rename the pasted **com.unity.ugui@1.0.0** folder to **com.unity.ugui**.

This will allow you to change the uGUI package yourself. Not only will you be able to change them, but you will also be able to search for references to them in Visual Studio.

### Note

If you want to see the source code for Unity 2019.1 or earlier, you can look at the repositories available on GitHub.

<https://github.com/Unity- Technologies/uGUI>

Of course, the source code for components written in C++, such as [Canvas](#), is not available to the public because components such as [Canvas](#) belong to the [UnityEngine](#) namespace.

Components of [UnityEngine.UI](#) or [UnityEngine.EventSystems](#) namespaces (such as [Image](#) and [Text](#)) are written in C#. But components in the [UnityEngine](#) namespace, such as [Canvas](#), are written in C++.

## Current state of the UI Toolkit (UIElements)

Unity is developing a new UI framework, the UI Toolkit, formerly called UI Elements. The UI Toolkit is now runtime compatible with Unity 2021.2. However, as of 2021.2, there are many features that are not yet implemented. Note that the UI in the world space is not yet supported.

At least as of August 2021, there will be no reason for new projects to use the UI Toolkit as a basic GUI feature. (In fact, there are still many applications made with NGUI in 2021).

The UI Toolkit can coexist with uGUI. It may be a good idea to use uGUI for the time being and gradually migrate to UI Toolkit. In any case, we still need to keep in touch with uGUI for a while, which is also the reason for writing this book.

## Target Unity Version

This document is basically written based on Unity 2020.2. Some new uGUI features that were added before Unity 2020.2 will also be discussed. For example, some readers may not be familiar with the soft mask of the [Mask](#) component, premultiplied transparency of UI shaders, and padding of raycast regions, which were introduced in Unity 2020.1. You can learn about these new features in this book.

## License

All sample code in this document, whether derived from Unity uGUI or original, is distributed under the license of Unity uGUI.

<https://docs.unity3d.com/Packages/com.unity.ugui@1.0/license/LICENSE.html>

In most cases, the license of Unity uGUI is also indicated in the product, so you don't need to do anything special.

## About the English version

This document was originally written in Japanese and translated by DeepL. If you find any translation errors, incorrect source code, or broken layouts, please report them on GitHub.

## Table of Contents

<b>Introduction</b>	<b>2</b>
Who this book is for	2
Accessing the uGUI source code	3
Current state of the UI Toolkit (UIElements)	4
Target Unity Version	5
License	5
About the English version	5
Table of Contents	6
<b>Chapter 1 Canvas and Related Components</b>	<b>14</b>
Overview of Canvas	14
Canvas component	16
Properties of Canvas	17
Static methods of Canvas	30
Events of Canvas	31
CanvasScaler component	32
ConstantPixelSize mode	33
ScaleWithScreenSize mode	34
ConstantPhysicalSize mode	35
CanvasScaler Properties	36
GraphicRaycaster Components	44
Ignore Reversed Graphics	44
Blocked Objects	45
Blocking Mask	45
CanvasGroup component	46
Properties of CanvasGroup	47
Public methods of CanvasGroup	49
<b>Chapter 2 Rebuilding UI Elements and Canvas</b>	<b>50</b>
The UIBehaviour class	51
Methods of UIBehaviour	53
RectTransform component	56
Properties of RectTransform	58
Public method of RectTransform	65
RectTransform events	68
Raw Edit mode for RectTransform	69

Blue Print mode in RectTransform	70
Enabling Variable Resolution on Smart Devices	71
CanvasRenderer component	77
Properties of CanvasRenderer	78
Public methods of CanvasRenderer	83
Static methods of CanvasRenderer	87
Events of CanvasRenderer	91
Graphic related components	92
Canvas Rebuild	93
Rebuild by CanvasUpdateRegistry	94
Canvas batch build	106
Canvas Rebuild Summary	107
Reduce the number of UI elements	108
Splitting into sub-Canvas	108
Using the SpriteRenderer instead of the CanvasRenderer	109
<b>Chapter 3 Rendering</b>	<b>120</b>
Drawing Order in Unity	120
Camera Depth	120
Sorting Layer	121
Order in Layer (sortingOrder)	124
Render Queue	125
Improved rendering performance	127
Fill rate	127
Overdraw	128
Show/Hide UI elements	129
Graphic scaling	131
Using shaders to change colors in the UI	131
Don't create GameObjects instead of folders	131
Omit rendering of the 3D world space behind the UI.	131
Integrate images	132
Adjust the order in the hierarchy.	133
Shader	134
Default UI shaders	134
Custom UI shaders	142
HSV color space shader	145
Texture format	154
Uncompressed 32 bit	155
Uncompressed 16bit	156
PVRTC	157
ETC1	166

ETC2	173
ASTC	174
DXT / BC	176
Texture import settings	177
RenderTexture	179
Screen resolution	182
Gets/Sets screen resolution	182
Getting the screen resolution in the Editor	183
<b>Chapter 4 Graphic</b>	<b>184</b>
Graphic Components	184
Static property of Graphic	186
Properties of the Graphic	187
Public methods of Graphic's	191
Protected methods of the Graphic	198
The VertexHelper class	200
Properties of VertexHelper	201
Public methods of VertexHelper	203
MaskableGraphic component	212
MaskableGraphic properties	213
MaskableGraphic's public method	215
Create a transparent layer to absorb touch events.	217
<b>Chapter 5 Images and Effects</b>	<b>218</b>
Using RawImage vs. Image	218
RawImage component	219
Properties of RawImage	220
Public methods of RawImage	221
Image component	222
Static variables of Image	223
Properties of Image	224
Public methods of Image	235
Shadow component	238
Properties of Shadow	239
Public Methods of Shadow	240
Protected method of Shadow	241
Outline Components	242
Public methods of Outline	246
Draw the outline with a shader instead of an Outline	247

PositionAsUV1component	257
Mask component	266
Stencil test	267
Whether or not to use the stencil test	276
Properties of Mask	277
Public methods of Mask's	278
RectMask2D component	279
Properties of RectMask2D	281
Public methods of RectMask2D	283
Make the transparent part of the Image unresponsive to touch.	285
<b>Chapter 6 Text and Fonts</b>	<b>290</b>
Text component	291
Properties of Text	293
Static methods of Text	304
Public methods of Text	305
The TextGenerator class	307
Properties of TextGenerator	310
Public methods of TextGenerator's	314
Use TextGenerator to abbreviate overflowed characters with ellipsis (...)	318
Control the spacing of characters	320
Font assets	330
Font class	330
Properties of the Font class	331
public methods of Font	335
Static methods of Font	336
Events of Font	338
Font asset file format	339
Dynamic font	340
Font-atlas texture	341
Font fallback	344
TrueTypeFontImporter class.	346
Properties of TrueTypeFontImporter	347
Custom fonts	353
TextMesh Pro	358
Install TextMesh Pro	359
TMP Settings	360
Setting up Font Asset Creator	361
Generation Settings in TextMesh Pro	365

TextMeshProUGUI component	368
Properties of TextMeshProUGUI	370
Public methods of TextMeshProUGUI	386
Events of TextMeshProUGUI	392
Comparison of Text and TextMesh Pro	394
<b>Chapter 7 Selectable</b>	<b>395</b>
Selectable	395
Selectable selection state	396
Static variables for Selectable	397
Properties of Selectable	398
Static methods of Selectable's	405
Public methods of Selectable's	406
Animation settings	411
Button component	415
Properties of Button	416
Public methods of Button	417
Toggle component	419
Public variables of Toggle	421
Toggle Properties	423
Toggle's public methods	424
ToggleGroupcomponent	426
Properties of ToggleGroup	427
Public methods of ToggleGroup	428
Slider component	431
Properties of Slider	433
Public methods of Slider	437
Dropdown Components	441
Properties of Dropdown	444
Public methods of Dropdown	449
InputField component	454
caret	456
InputField and Japanese input	457
Properties of InputField	458
Public methods of InputField	479
Scrollbar component	485
Properties of Scrollbar	487
Public methods of Scrollbar	491
<b>Chapter 8 Scroll View</b>	<b>496</b>

ScrollRect component	498
Properties of ScrollRect	499
Public methods of ScrollRect	509
How to configure content for Scroll View	514
Notes on content updates	515
ScrollView Sample	516
<b>Chapter 9 Auto Layout</b>	<b>524</b>
Auto Layout Overview	524
Rules for determining the size of the Layout Element	526
Manual sizing with the Layout Element component	526
Layout Controller	527
ContentSizeFitter component to adjust its own size	527
AspectRatioFitter component	527
Layout Group	528
Driven properties of RectTransform	529
Create your own Auto Layout components	531
Calculating the layout	532
Layout rebuild triggers	533
ILayoutElement interface	534
Properties of ILayoutElement	535
Public methods of ILayoutElement	537
ILayoutController interface	538
Public method of ILayoutController	539
ILayoutGroup interface	540
ILayoutSelfController interface	541
ILayoutIgnorer interface	542
Properties of ILayoutIgnorer	543
LayoutElement component	544
LayoutGroup component	545
Properties of LayoutGroup	546
Public methods of LayoutGroup	550
HorizontalOrVerticalLayoutGroup component	551
Properties of HorizontalOrVerticalLayoutGroup	552
HorizontalLayoutGroup component	555
Public methods of HorizontalLayoutGroup	557

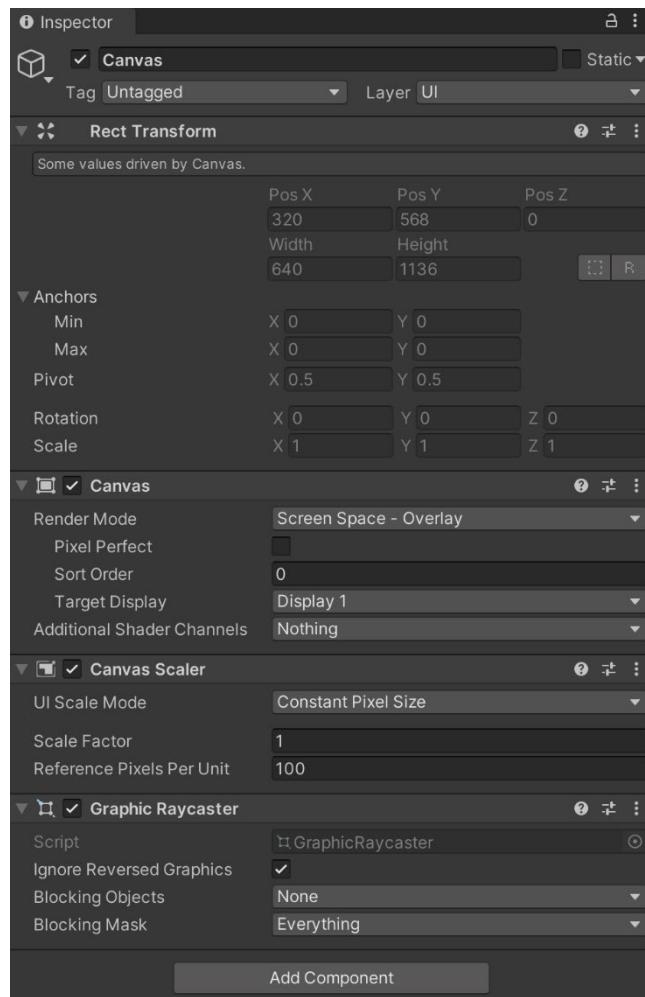
VerticalLayoutGroup component	558
VerticalLayoutGroup public method	560
GridLayoutGroup component	561
Combination of flexibleWidth and fixed height	562
Combination of fixed width and flexibleHeight	562
Combination of flexibleWidth and flexibleHeight	563
ContentSizeFitter component	564
Properties of ContentSizeFitter	565
Public methods of ContentSizeFitter	567
AspectRatioFitter component	568
Properties of AspectRatioFitter	569
Public methods of AspectRatioFitter	571
Use Anchor instead of Auto Layout	572
<b>Chapter 10 EventSystem</b>	<b>573</b>
EventSystem Components	573
Update() of EventSystem	574
Note on disabling EventSystem.	575
Static properties of EventSystem	576
Properties of EventSystem	577
Public methods of EventSystem	579
Events supported by EventSystem	581
Messaging System	583
Send your own messages using the Messaging System	584
Advantages and disadvantages of Messaging System	587
InputModule	588
StandaloneInputModule component	589
Items that can be set in StandaloneInputModule	590
Process flow of StandaloneInputModule	591
Static variables of StandaloneInputModule	592
Properties of StandaloneInputModule	593
Public methods of StandaloneInputModule	599
BaseInput	602
Properties of BaseInput	603
BaseInput public methods	606
Event Trigger component	608
Types of events that can be specified in EventTrigger	611
Raycaster	614
GraphicRaycaster	617

GraphicRaycaster Properties	618
Public methods of GraphicRaycaster	623
PhysicsRaycaster component	624
Properties of PhysicsRaycaster	625
Public methods of PhysicsRaycaster	627
Physics2DRaycaster component	628
Public methods of Physics2DRaycaster	629
Extending StandaloneInputModule	630
Prevent continuous touch/click within a certain period of time.	631
Prepare a flag to disable touch/click at once.	631
Disable taps in a SafeArea	631
Improving the Click judgment in ScrollView	631
Create the CustomStandaloneInputModule	633
<b>Chapter 11 Profiling</b>	<b>656</b>
Profiling tools in uGUI	656
Unity Profiler	656
CPU Usage area	657
Memory Area	658
UI Area	659
UI Details area	662
Frame Debugger	663
Analysis of CPU Usage area and patterns of causes	665

# Chapter 1 Canvas and Related Components

## Overview of Canvas

[Canvas](#) is a component for rendering UI elements of uGUI, and a good understanding of [Canvas](#) leads to a good understanding of uGUI.



[Canvas](#) cooperates with other components attached to the same [GameObject](#), such as [CanvasScaler](#) and [GraphicRaycaster](#) component.

[Canvas](#) is a Unity component written in native code (C++). uGUI's many components are written in C# and the source code is available, but [Canvas](#) is not.

Note

The C# bindings code for [Canvas](#) is available on GitHub.

<https://github.com/Unity-Technologies/UnityCsReference/blob/master/Modules/UI/ScriptBindings/UICanvas.bindings.cs>

In short, [Canvas](#) does the following three things.

1. Determine the drawing method and size of the screen that displays the UI.
2. Just before rendering, call the callback registered in [Canvas.willRenderCanvases](#) to have other classes prepare for rendering (Canvas rebuild).
3. Render all of the [CanvasRenderer](#) meshes under the [Canvas](#) (Batch build).

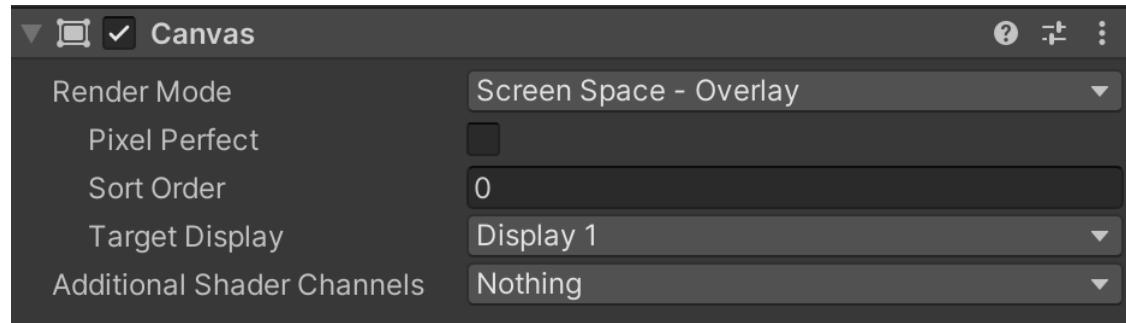
[Canvas](#) itself does not know whether the mesh it is trying to draw is an image or text. Also, [Canvas](#) is not directly involved in the processing of touch events; [EventSystem](#) and [StandAloneInputModule](#) handle that.

The first point is that the drawing method and screen size can be controlled by accessing properties such as [renderMode](#) from [Inspector](#) or script.

The second point is that the Canvas rebuild is not handled by [Canvas](#) itself, but by a class called [CanvasUpdateRegistry](#), and [Canvas](#) simply lets [CanvasUpdateRegistry](#) handle the rebuild at the appropriate time.

Regarding the third point, batch build, [Canvas](#) gets the geometry of each UI element under it from the [CanvasRenderer](#), combines it into a batch, generates the appropriate render commands, and sends them to the Unity graphics system. This operation is called **Batch Build**. This batch build process is done in native code; when a batch build of Canvas is required, we sometimes refer to it as "Canvas is dirty". The details of batch build are explained in the *Canvas Batch Build* section of *Chapter 2 UI Elements and Canvas Rebuild*.

## Canvas component



The definition of the [Canvas](#) class is shown below.

```
[RequireComponent(typeof(RectTransform)),  
 NativeClass("UI::Canvas"),  
 NativeHeader("Modules/UI/Canvas.h"),  
 NativeHeader("Modules/UI/UIStructs.h")]  
public sealed class Canvas : Behaviour
```

This is an excerpt from the C# bindings code for [Canvas](#).

[Behaviour](#) component, parent class of [Canvas](#), is the parent class of [MonoBehaviour](#). The inheritance relationship is as follows.

```
UnityEngine.Object  
UnityEngine.Component  
UnityEngine.Behaviour  
MonoBehaviour
```

Now, we will look at the properties, public methods, and events of the [Canvas](#) component, which will be explained in detail as the official Unity documentation is often too simplistic.

## Properties of Canvas

### renderMode

```
public extern RenderMode renderMode { get; set; }
```

Gets/Sets whether the UI should be drawn on the screen or exist as an object in 3D space, as shown as **Render Mode** in **Inspector**.

The definition of **RenderMode** is as follows.

```
/// <summary>
/// <para> RenderMode of Canvas</para>
/// </summary>
public enum RenderMode
{
    /// <summary>
    /// <para> Render at the end of the scene using 2D Canvas. </para>
    /// </summary>
    ScreenSpaceOverlay,

    /// <summary>
    /// <para> Render using the Camera set in the Canvas. </para>
    /// </summary>
    ScreenSpaceCamera,

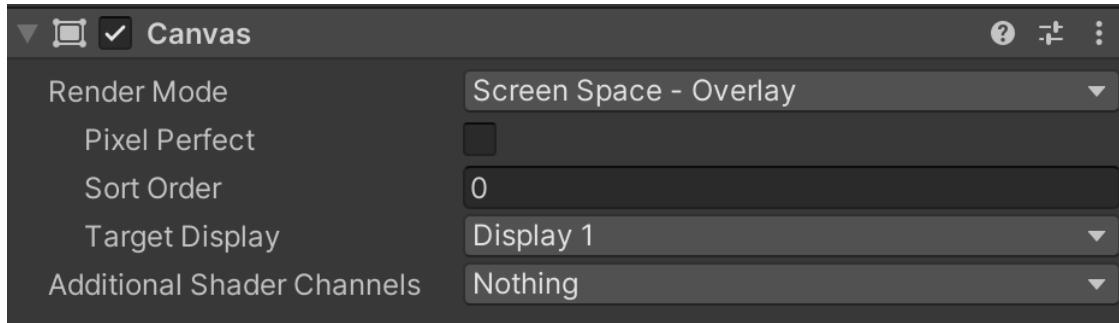
    /// <summary>
    /// <para> Render using the Camera in the scene. </para>
    /// </summary>
    WorldSpace
}
```

The default value is **ScreenSpaceOverlay**.

Each mode is explained below.

## ScreenSpaceOverlay mode

---



If `renderMode` is set to [ScreenSpaceOverlay](#), the UI will be drawn directly on the screen. In this case, there is no need to set up a camera in Canvas. Rendering will be overwritten and drawn last after all other cameras have finished rendering.

### Advantages of ScreenSpaceOverlay

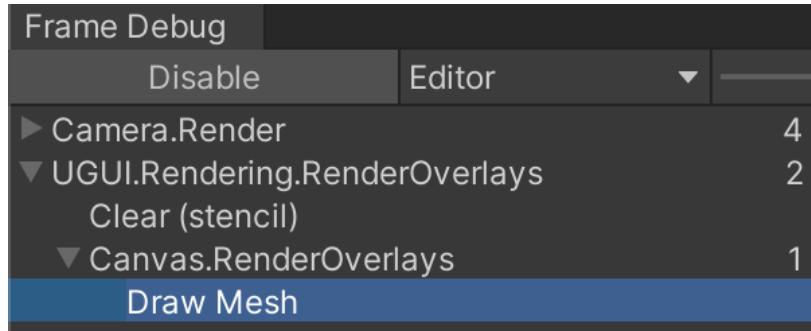
- There is no need to set the [Camera](#).
- When multiple Canvases exist, you can adjust the drawing order between the Canvases simply by increasing or decreasing the `renderOrder` value, making management easier.

### Disadvantages of ScreenSpaceOverlay

- It is difficult to display 3D models and particles on a [Canvas](#) using [MeshRenderer](#). If you try to do this, you will need to render them into a [RenderTexture](#) and then display the result as a [RawImage](#), which will degrade performance and increase the complexity of management.
- If you make it a child of a [Canvas](#) with another `renderMode` set, the UI will not be displayed at all.

You can use this [ScreenSpaceOverlay](#) mode for normal UIs that are not complicated. If you have problems with this mode, consider using the [ScreenSpaceCamera](#) mode.

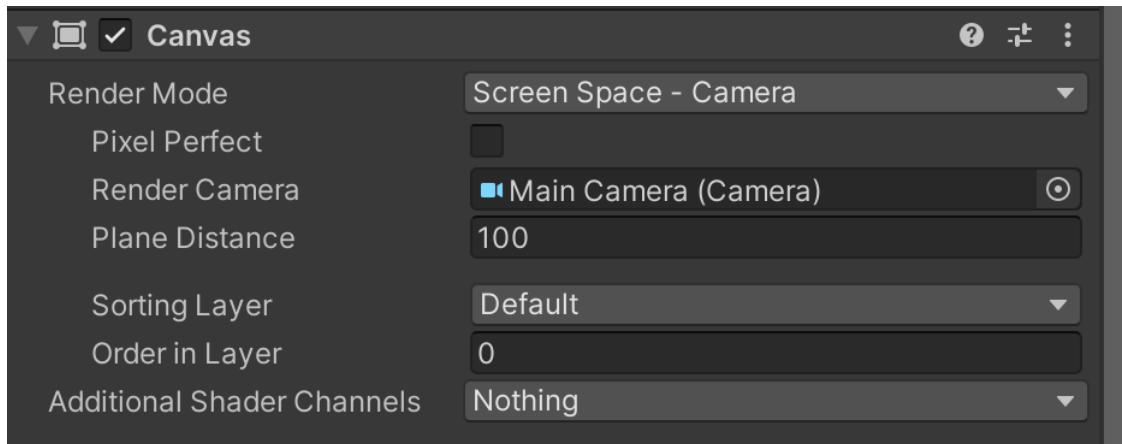
Also, when drawing in this mode, the **Frame Debugger** will show draw calls after `Canvas.RenderOverlays`.



## ScreenSpaceCamera mode

---

To solve the problem of mixing `ScreenSpaceOverlay` with 3D as explained above, you can use this `ScreenSpaceCamera` mode.



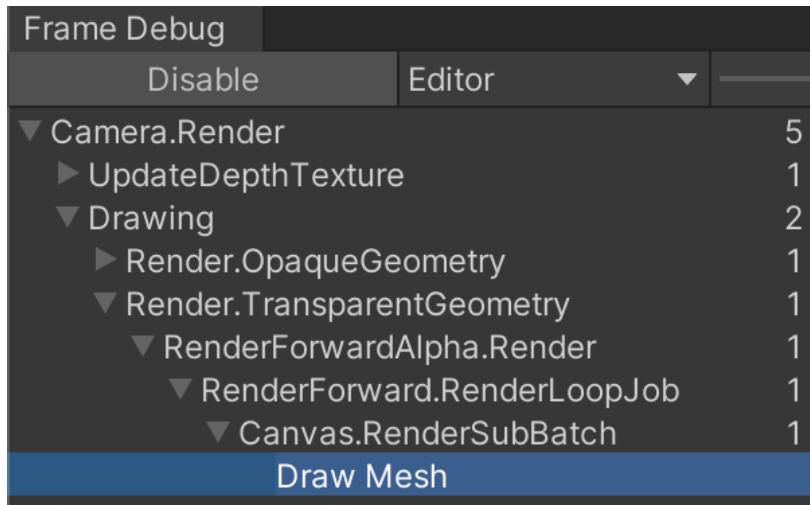
In this mode, you don't need to explicitly set the camera, but if you do, you will have to access `Camera.main` every time, which will consume CPU time. property (shown as Render Camera in Inspector).

In this mode, the drawing order between the `Canvas` is basically adjusted by `sortingOrder`. Since there are other parameters involved in determining the actual drawing order, we will explain the details in the `sortingOrder` property section.

**Note**

The access speed to `Camera.main` has been improved somewhat since 2019.4.9f1.

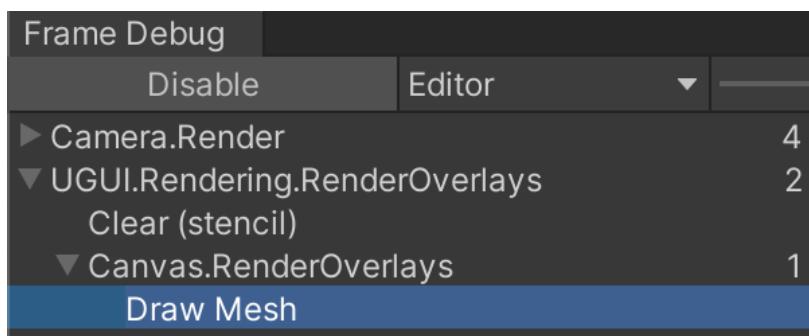
Also, when drawing in this mode, the **Frame Debugger** shows draw calls after `Render.TransparentGeometry`.



## WorldSpace mode

This is the mode to choose if you want to place the UI in 3D space, and the `worldCamera` property (labeled **Event Camera** in **Inspector**) needs to be set to `Camera`. Only in this mode, the `dynamicPixelsPerUnit` setting of the `CanvasScaler` component will be used to reduce the number of pixels used to draw distant `Text` and gain performance.

Also, when drawing in this mode, the **Frame Debugger** shows draw calls after `Render.TransparentGeometry`.



## pixelPerfect

---

```
public extern bool pixelPerfect { get; set; }
```

Gets/Sets whether the UI element should be displayed clearly in dot-by-dot mode or not.

The default value is `false`.

It is displayed as **Pixel Perfect** in **Inspector**. This property is only used when `renderMode` is `ScreenSpaceOverlay` or `ScreenSpaceCamera`.

The performance cost of enabling dot-by-dot display is very high. Also, just because dot-by-dot is enabled does not necessarily mean that the display will be clear. In the case of mobile devices, the resolution of the device and the rendering resolution may differ, so even if the setting is enabled in such a situation, the dot-by-dot display of the device may not be realized.

If you really want to use this option, you should only use it on a `Canvas` where UI elements do not move. Alternatively, you may be able to create the image you want by simply setting `Texture2D`'s `FilterMode` to `Point`, so you should consider that first.

## overridePixelPerfect

---

```
public extern bool overridePixelPerfect { get; set; }
```

Gets/Sets whether or not this `Canvas` overrides the `pixelPerfect` setting of the parent `Canvas`.

The default value is `false`, that is, it inherits the settings of the parent `Canvas`.

In **Inspector**, you can choose between **Inherit Pixel Perfect**, **On** (Pixel Perfect regardless of the parent's setting), and **Off** (Pixel Perfect regardless of the parent's setting).

## sortingOrder

---

```
public extern int sortingOrder { get; set; }
```

Gets/Sets the drawing priority in the same **Sorting Layer**.

The [Canvas](#) with the higher value will be drawn later.

The default value is [0](#).

The **Inspector** shows it as a **Sort Order**.

Note

**Sorting Layer** is different from **Layer**. If you want to add or remove **Sorting Layers**, go to *Project Settings* and edit *Tags and Layers*.

If `renderMode` is [ScreenSpaceOverlay](#), then [Camera](#) depth and **Sorting Layer** are not taken into account, and only this `sortingOrder` is used to control the drawing order.

If `renderMode` is [ScreenSpaceCamera](#) or [WorldSpace](#), the [Canvas](#) with the smallest [Camera](#) depth value is drawn first, followed by the [Canvas](#) with the smallest **Sorting Layer**, the [Canvas](#) with the smallest `SortingOrder`, and the Canvas farther from the camera. If the `renderMode` is [ScreenSpaceCamera](#) or [WorldSpace](#), the Canvas with the smaller Camera depth value is drawn first, followed by the Canvas with the smaller Sorting Layer, the Canvas with the smaller `SortingOrder`, and the [Canvas](#) farther from the camera.

In summary, the [Canvas](#) is selected in the following order, and is drawn in the order in which it was selected (at the back).

In the case of [ScreenSpaceOverlay](#), the [Canvas](#) with the smallest `sortingOrder` is drawn first.

In the case of [ScreenSpaceCamera](#) or [WorldSpace](#)

1. [Canvas](#) with a small [Camera](#) depth value
2. [Canvas](#) with small **Sorting Layer**
3. [Canvas](#) with small `sortingOrder`
4. For [ScreenSpaceCamera](#), `planeDistance` is large [Canvas](#)
5. In the case of [WorldSpace](#), [Canvas](#) is far from the [Camera](#).

In this order, the [Canvas](#) is selected and drawn.

The details of the drawing order are explained in the section on *Unity's drawing order* in *Chapter 3, Rendering*.

## overrideSorting

---

```
public extern bool overrideSorting { get; set; }
```

Gets/Sets whether or not this [Canvas](#) overrides the [sortingOrder](#) of the parent [Canvas](#).

The default value is [false](#), which means that the parent [Canvas](#)'s [sortingOrder](#) is used.

If the value of this property is [true](#), it will be possible to set the **Sort Order** from [Inspector](#).

## worldCamera

---

```
[NativeProperty("Camera", false, TargetType.Function)]  
public extern Camera worldCamera { get; set; }
```

If [renderMode](#) is [ScreenSpaceOverlay](#), the value of this property is not used.

If [renderMode](#) is [ScreenSpaceCamera](#), the value of this property is the [Camera](#) used to adjust the size of the [Canvas](#), and the [Inspector](#) will display **Render Camera**.

If [renderMode](#) is [WorldSpace](#), the value of this property is the [Camera](#) to which the event is sent, and the [Inspector](#) shows **Event Camera**.

## targetDisplay

---

```
public extern int targetDisplay { get; set; }
```

Gets/Sets the ID of the display to show the UI when [renderMode](#) is [ScreenSpaceOverlay](#).

If [renderMode](#) is set to anything other than [ScreenSpaceOverlay](#), the display to be displayed will depend on the [worldCamera](#) **Target Display** setting.

## additionalShaderChannels

```
public extern AdditionalCanvasShaderChannels additionalShaderChannels { get; set; }
```

The default vertex shader in uGUI only uses (receives) `position`, `color`, and `uv0`, but if you want to use other channels (`uv1`, `uv2`, `uv3`, `normal`, `tangent`), get/set this property.

The definition of `AdditionalCanvasShaderChannels` is as follows.

```
/// <summary>
/// <para>An additional parameter to include in existing channels when Canvas meshes are created</para>
/// </summary>
[Flags]
public enum AdditionalCanvasShaderChannels
{
    /// <summary>
    /// <para> No additional shader parameters are needed. </para>
    /// </summary>
    None = 0x0,

    /// <summary>
    /// <para> Include UV1 in the mesh vertices. </para>
    /// </summary>
    TexCoord1 = 0x1,

    /// <summary>
    /// <para> Include UV2 in the mesh vertices. </para>
    /// </summary>
    TexCoord2 = 0x2,

    /// <summary>
    /// <para> Include UV3 in the mesh vertices. </para>
    /// </summary>
    TexCoord3 = 0x4,

    /// <summary>
    /// <para> Include Normal in mesh vertices. </para>
    /// </summary>
```

```
Normal = 0x8,  
  
/// <summary>  
/// <para> Include Tangent in mesh vertices. </para>  
/// </summary>  
Tangent = 0x10  
}
```

The [AdditionalCanvasShaderChannels](#) enumeration type is a flag of bits.

The default value for this property is [None](#).

If you want to use the [PositionAsUV1](#) component, you need to set up a bit for [TexCoord1](#) in [additionalShaderChannels](#). If you want to use additional channels such as [UV1](#) to give some parameters to the UI shader, you need to set up the respective bits. For example, the [TextMeshProUGUI](#) component of [TextMeshPro](#) uses [TexCoord1](#), [Normal](#), and [Tangent](#), and sets the corresponding [AdditionalCanvasShaderChannels](#) flag during text mesh generation. The [TextMeshProUGUI](#) component uses [TexCoord1](#), [Normal](#), and [Tangent](#).

Since UI shaders cannot use Material Property Blocks ([\[PerRendererData\]](#)), with some exceptions such as textures, the technique of using vertex channels to specify the specific behavior of each UI element may be used. [AdditionalShaderChannels](#) can be used in such a case. If you are not sure, leave it at the default setting of [None](#) ([Nothing](#) in [Inspector](#)).

## planeDistance

```
public extern float planeDistance { get; set; }
```

Gets/Sets how far away this [Canvas](#) is from [worldCamera](#) when [renderMode](#) is [ScreenSpaceCamera](#).

The default value is [100](#).

The [Inspector](#) shows it as **Plane Distance**.

If this value is small, it means that the image is close to [worldCamera](#) and will be displayed in the foreground. On the other hand, if the value is large, it means that the image is far from [worldCamera](#) and will be displayed in the back.

## renderOrder

---

```
public extern int renderOrder { get; }
```

It is supposed to get the drawing order of each [Canvas](#) in the scene, but it doesn't actually return an appropriate value. Therefore, you should not rely on the value of this property. For information about the drawing order when multiple [Canvas](#) exist, see the explanation in the [sortingOrder](#) property section above.

## sortingLayerName

---

```
public extern string sortingLayerName { get; set; }
```

Gets/Sets the name of the **Sorting Layer** of the [Canvas](#).

The default value is "Default".

## cachedSortingLayerValue

---

```
public extern int cachedSortingLayerValue { get; }
```

Get the ID of the **Sorting Layer** of the [Canvas](#).

It is essentially the same as calling [GetLayerValueFromName\(canvas.sortingLayerName\)](#).

## sortingLayerID

---

```
public extern int sortingLayerID { get; set; }
```

Gets/Sets the ID of the **Sorting Layer** of the [Canvas](#).

"Default" has an ID of [0](#), so the default value for this property is [0](#).

The ID of a **Sorting Layer** is a unique ID, and if you add a **Sorting Layer** yourself, the ID of the added **Sorting Layer** will be a random value such as [1317894267](#) or [702774661](#). If you want to get the sorting layer value (i.e. priority) instead of the ID, refer to [cachedSortingLayerValue](#).

### isRootCanvas

```
public extern bool isRootCanvas { get; }
```

Gets whether or not this [Canvas](#) is the root Canvas (i.e., not a sub-Canvas).

### rootCanvas

```
public extern Canvas rootCanvas { get; }
```

Get the root Canvas.

If it is itself the root Canvas, it returns itself.

### pixelRect

```
public extern Rect pixelRect { get; }
```

Get the rectangle that represents the area of this [Canvas](#).

[x](#) and [y](#) are fixed to [0](#) regardless of the [renderMode](#) setting or the [worldCamera](#) position. [width](#) and [height](#) are the width and height of the [Canvas](#).

### scaleFactor

```
public extern float scaleFactor { get; set; }
```

Gets/Sets the factor to scale up/down the size of the [Canvas](#).

If [renderMode](#) is [WorldSpace](#), the value of this property will not be used.

In fact, the value is copied from the `scaleFactor` of the `CanvasScaler` attached to the same `GameObject` and set to this property. Conversely, copying the value from the `scaleFactor` of `Canvas` to the `scaleFactor` of `CanvasScaler` will not occur. To prevent value mismatch, values should not be set via this property. Note that the copying of the value from the `scaleFactor` of `CanvasScaler` to this property is done at the timing of `Update()` of `CanvasScaler`.

### referencePixelsPerUnit

```
public extern float referencePixelsPerUnit { get; set; }
```

Gets/Sets the value used to calculate the number of pixels per unit.

#### Note

When drawing an `Image`, the `pixelsPerUnit` of the `Sprite` (default value is `100`) divided by this property will be the number of pixels per unit (if `CanvasScaler's uiScaleMode` is other than `ConstantPhysicalSize`, the value will be `1` pixel per unit). If the `CanvasScaler's uiScaleMode` is not `ConstantPhysicalSize`, it will be `1` pixel per unit).

The value is calculated based on the `referencePixelsPerUnit` value of the `CanvasScaler` attached to the same `GameObject`, and set to this property. As with `scaleFactor`, the value should not be set via this property. As with `scaleFactor`, setting values via this property should not be done, and setting values from `CanvasScaler` to this property is done at the timing of `CanvasScaler's Update()`.

### normalizedSortingGridSize

```
[NativeProperty("SortingBucketNormalizedSize", false, TargetType.Function)]  
public extern float normalizedSortingGridSize { get; set; }
```

Gets the size at which the `Canvas` will divide the rendering area.

The minimum value is `0`, the maximum value is `1`, and the default value is `0.1f`. Even if the value is set to `0`, the default value of `0.1f` will be set.

When the `Canvas` renders, the rendering area is divided into grids and processed as a Job. For example, if the rendering area is `100` units and the `normalizedSortingGridSize` is `0.1f`, then each grid will be `10` units.

This value affects the performance of the Canvas rebuild. If the number of splits is too large, the number of Jobs may be too large and performance may be degraded (the processing time of [Semaphore.WaitForSignal](#) for rendering Jobs will become longer). If you have a lot of canvas rebuilds, increasing this property to [0.5f](#) or so (after performing basic optimizations such as splitting into sub-Canvas) may improve performance, so please consider this as a last resort.

## Static methods of Canvas

### ForceUpdateCanvases

```
public static void ForceUpdateCanvases();
```

Force a refresh of the [Canvas](#).

Calling this method will force [Canvas.SendWillRenderCanvases\(\)](#), which is called every frame, to be called, and then [Canvas](#) rebuild with [CanvasUpdateRegistry.PerformUpdate\(\)](#) will be performed. will be explained in detail later.

### GetDefaultCanvasMaterial

```
[FreeFunction("UI::GetDefaultUIMaterial")]
public static extern Material GetDefaultCanvasMaterial();
```

Gets the default material to be used if no material is specified.

### GetETC1SupportedCanvasMaterial

```
[FreeFunction("UI::GetETC1SupportedCanvasMaterial")]
public static extern Material GetETC1SupportedCanvasMaterial();
```

Gets the default material for ETC1, which will be used if no material is specified.

## Events of Canvas

### willRenderCanvases event

```
public static event WillRenderCanvases willRenderCanvases;
```

WillRenderCanvases will be explained many times in this document.

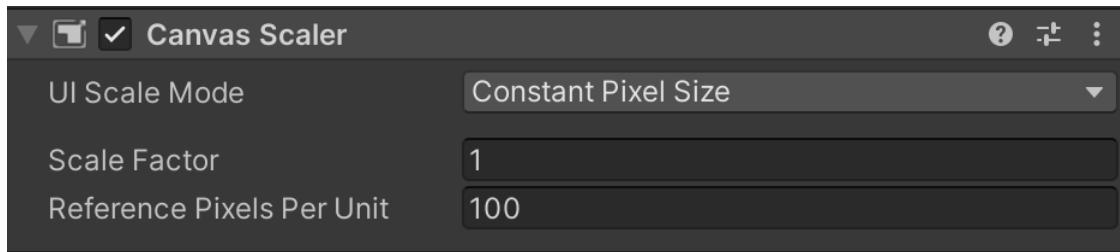
You can add your own events here.

```
class MyImage : MonoBehaviour
{
    void Start()
    {
        Canvas.willRenderCanvases += MyEvent;
    }

    void OnDestroy()
    {
        Canvas.willRenderCanvases -= MyEvent;
    }

    void MyEvent()
    {
        Debug.Log("Canvas.willRenderCanvases event occurred");
    }
}
```

## CanvasScaler component



```
[RequireComponent(typeof(Canvas))]
[ExecuteAlways]
[AddComponentMenu("Layout/Canvas Scaler", 101)]
[DisallowMultipleComponent]
public class CanvasScaler : UIBehaviour
```

The [CanvasScaler](#) component is used to control the overall scale and pixel density of UI elements in a [Canvas](#).

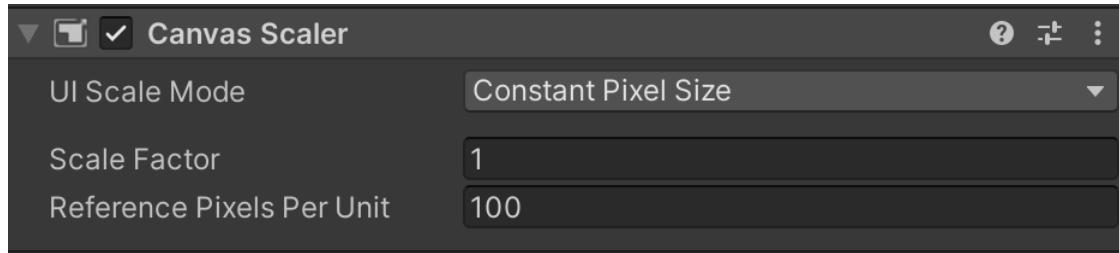
This component is automatically attached to the [Canvas](#) when the [Canvas](#) is created by right-clicking in **Hierarchy** or *GameObject -> UI -> Canvas*. By using CanvasScaler, it is possible to achieve resolution-independent UI placement.

Scaling with the [CanvasScaler](#) affects everything under the [Canvas](#). Font size, image borders, etc. are also affected by scaling.

If the [renderMode](#) of the [Canvas](#) is [ScreenSpaceOverlay](#) or [ScreenSpaceCamera](#), you can set the [uiScaleMode](#). [uiScaleMode](#) has three types: [ConstantPixelSize](#), [ScaleWithScreenSize](#), and [ConstantPhysicalSize](#). There are three types of [uiScaleMode](#): [ConstantPixelSize](#), [ScaleWithScreenSize](#), and [ConstantPhysicalSize](#).

### ConstantPixelSize mode

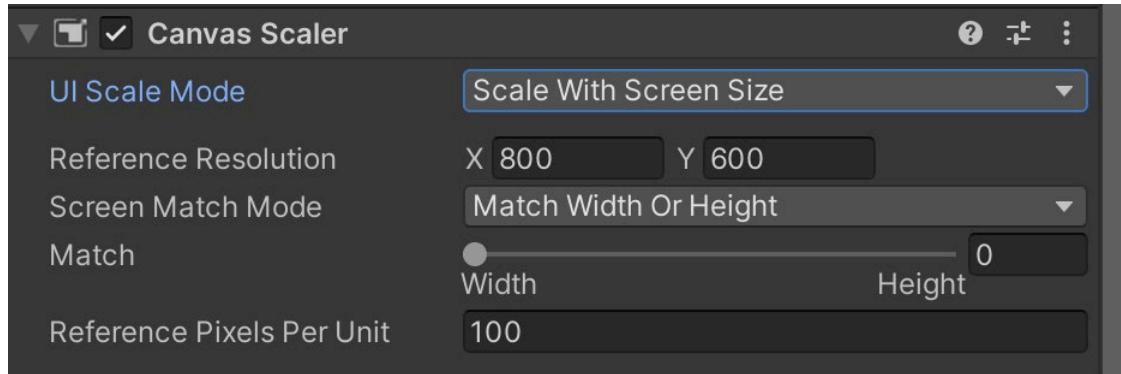
When `uiScaleMode` is `ConstantPixelSize`, the position and size of the UI element is specified in terms of pixels on the screen.



This is the default behavior of `Canvas` when `CanvasScaler` is not attached. However, if you change the value of `scaleFactor`, all UI elements in the `Canvas` will be scaled.

## ScaleWithScreenSize mode

If `uiScaleMode` is `ScaleWithScreenSize`, the position and size of the UI element shall be specified according to the number of pixels in `referenceResolution`.

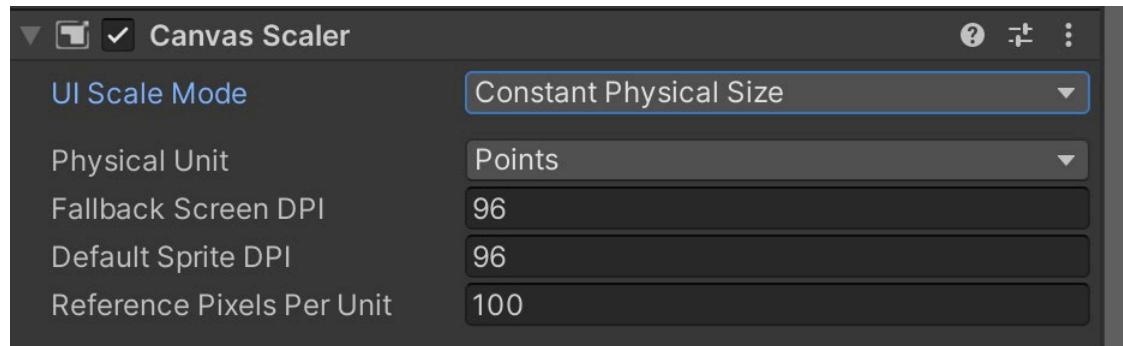


If the current screen resolution is greater than the `referenceResolution`, the `Canvas` will be scaled to fit the screen resolution while maintaining the `referenceResolution`. Conversely, if the current screen resolution is less than the `referenceResolution`, the `Canvas` will be scaled to fit the screen resolution.

Furthermore, if the aspect ratio of the current screen resolution is different from that of the `referenceResolution`, the `Canvas` will be distorted when it is enlarged or reduced. To avoid this, the resolution of the `Canvas` will be changed to something different from the `referenceResolution`. How this is done depends on the `screenMatchMode` setting.

## ConstantPhysicalSize mode

If `uiScaleMode` is `ConstantPhysicalSize`, the position and size of UI elements are specified in physical units such as millimeters or points.



This mode works on the assumption that the device reports the screen DPI properly. A fallback DPI can be specified in case the device does not report the DPI properly.

When the `renderMode` of the `Canvas` is `WorldSpace`, the pixel density of the UI element can be set.

## CanvasScaler Properties

### uiScaleMode

```
public CanvasScaler.ScaleMode uiScaleMode { get; set; }
```

Gets/Sets how the UI elements in Canvas are scaled according to each mode.

The definition of `CanvasScaler.ScaleMode` is as follows.

```
public enum ScaleMode
{
    /// <summary>
    /// In Constant Pixel Size mode, the position and size of the UI element is specified in pixels of
    /// the screen.
    /// </summary>
    ConstantPixelSize,

    /// <summary>
    /// In Scale With Screen Size mode, the position and size of the UI element can be specified in
    /// terms of the number of pixels specified by referenceResolution. If the current screen resolution is
    /// greater than the referenceResolution, the Canvas will be scaled to fit the screen while maintaining
    /// the referenceResolution. If the current screen resolution is smaller than the referenceResolution,
    /// the canvas will be shrunk.
    /// </summary>
    ScaleWithScreenSize,

    /// <summary>
    /// In Constant Physical Size mode, the position and size of the UI element is specified in physi
    /// cal units such as millimeters or points.
    /// </summary>
    ConstantPhysicalSize
}
```

The description of each mode is as follows.

## Constant Pixel Size mode

---

This is the mode in which the size of UI elements does not depend on the screen size. This is the default mode. Also, [Canvas](#) without [CanvasScaler](#) attached behaves the same as this mode.

Although we wrote that "the size of UI elements does not depend on the screen size" in this mode, the size of UI elements can be increased or decreased by changing the value of [scaleFactor](#). The size of the UI element can be adjusted by setting the [scaleFactor](#) to a factor that compares the assumed resolution with the screen size on the actual device, [Screen.width](#) or [Screen.height](#). However, the actual size of the [RectTransform](#) of the [Canvas](#) will increase or decrease according to the [scaleFactor](#).

If the aspect ratio of the screen is the same between the Unity Editor and the actual device, this is sufficient. However, since smartphones and tablets have various resolutions and aspect ratios, it is recommended to use the **Scale With Screen Size** mode in this case.

## Scale With Screen Size mode

---

If the current screen resolution is larger than the [referenceResolution](#), the [Canvas](#) will be enlarged; if it is smaller, the [Canvas](#) will be shrunk.

If the screen resolution matches the aspect ratio of [referenceResolution](#), the appearance of the UI is preserved. The behavior when the aspect ratio does not match depends on the [screenMatchMode](#) setting.

## Constant Physical Size mode

---

This is a mode in which UI elements are independent of screen resolution and maintain their physical size.

The position and size of UI elements are specified in physical units such as millimeters, points, and picas.

### referenceResolution

---

```
public Vector2 referenceResolution { get; set; }
```

Gets/Sets the reference resolution of the [Canvas](#) when the [uiScaleMode](#) is [ScaleWithScreenSize](#).

The default value is (800, 600).

### screenMatchMode

```
public CanvasScaler.screenMatchMode screenMatchMode { get; set; }
```

Gets/Sets how to scale the [Canvas](#) when the aspect ratio of the current screen resolution is different from the aspect ratio of [referenceResolution](#) when the [uiScaleMode](#) is [ScaleWithScreenSize](#).

The definition of [CanvasScaler.ScreenMatchMode](#) is as follows.

```
/// <summary>
/// Specifies how the Canvas is scaled.
/// </summary>
public enum ScreenMatchMode
{
    /// <summary>
    /// Set the Canvas scale reference to width, height, or somewhere in between
    /// </summary>
    MatchWidthOrHeight = 0,

    /// <summary>
    /// Scale the Canvas horizontally or vertically so that its size is no smaller than the referenceResolution.
    /// </summary>
    Expand = 1,

    /// <summary>
    /// Truncate horizontally or vertically so that the size of the Canvas is no larger than the referenceResolution.
    /// </summary>
    Shrink = 2
}
```

The default value is [MatchWidthOrHeight](#).

The description of each mode is as follows.

## MatchWidthOrHeight mode

---

Matches either the width or the height of the [Canvas](#) to the [referenceResolution](#). Which one to match is determined by the value of the [matchWidthOrHeight](#) property.

If the value of [matchWidthOrHeight](#) is **0** (= **Width**), the [Canvas](#) will be drawn with the full width. In other words, if the screen resolution is longer than the reference resolution, there will be gaps at the top and bottom, and if the screen resolution is longer than the reference resolution, the top and bottom will be cut off.

If the value of [matchWidthOrHeight](#) is **1** (= **Height**), the [Canvas](#) will be drawn with the full height and width. In other words, if the screen resolution is taller than the reference resolution, the left and right sides will be cut off and displayed, and if the screen resolution is wider than the reference resolution, there will be gaps on the left and right sides.

If the value of [matchWidthOrHeight](#) is [between 0 and 1](#), it will be a blend of the above behaviors.

## Expand mode

---

Even if the aspect ratio is not the same, it will always be scaled so that the entire [Canvas](#) is displayed. In other words, the top, bottom, left, and right sides will never be cut off. This mode is useful when supporting both smartphones and tablet devices. For more details, see the section "*Realizing Variable Resolution on Smart Devices*" in *Chapter 2, "Rebuilding UI Elements and Canvas*.

## Shrink mode

---

Even if the aspect ratio does not match, it will be scaled so that gaps are not displayed. Therefore, the top, bottom, left, and right sides of the [Canvas](#) may be cut off.

### matchWidthOrHeight

---

```
public float matchWidthOrHeight { get; set; }
```

If the [uiScaleMode](#) is [ScaleWithScreenSize](#), specifies whether the width or height of the [Canvas](#) should be adjusted to the [referenceResolution](#).

The default value is **0**, i.e., match the width.

The details are described in the *MatchWidthOrHeight mode* section.

### scaleFactor

---

```
public float scaleFactor { get; set; }
```

Gets/Sets the percentage of scaling of all UI elements in Cavans.

The default value is 1.

When this property is changed, the `localScale` of the `GameObject` will be changed accordingly. For example, if you change the `scaleFactor` to 2, the `localScale` will be (2, 2, 2), which will scale the underlying UI element.

### physicalUnit

---

```
public CanvasScaler.Unit physicalUnit { get; set; }
```

Gets/Sets the physical unit for specifying the position and size.

This property is used when `uiScaleMode` is `ConstantPhysicalSize`.

The definition of the `CanvasScaler.Unit` enumeration type is as follows.

```
/// <summary>
/// Physical Unit Type.
/// </summary>
public enum Unit
{
    /// <summary>
    /// Centimeter
    /// </summary>
    Centimeters,

    /// <summary>
    /// Millimeters
    /// </summary>
```

```
Millimeters,  
  
/// <summary>  
/// Inches  
/// </summary>  
Inches,  
  
/// <summary>  
/// Points  
/// One point is 1/12 of a pica, or 1/72 of an inch.  
/// </summary>  
Points,  
  
/// <summary>  
/// Paika  
/// 1 pica is 1/6 inch.  
/// </summary>  
Picas  
}
```

The default value is [Unit.Points](#).

## fallbackScreenDPI

---

```
public float fallbackScreenDPI { get; set; }
```

If the device does not report the correct DPI, get/set the value to be used as the DPI instead.

The default value is [96](#).

## defaultSpriteDPI

---

```
public float defaultSpriteDPI { get; set; }
```

Gets/Sets the number of Sprite pixels per inch when the [Sprite's pixelsPerUnit](#) value matches the [CanvasScaler's referencePixelsPerUnit](#) value.

The default value is [96](#).

### dynamicPixelsPerUnit

```
public float dynamicPixelsPerUnit { get; set; }
```

Gets/Sets the amount of pixels per unit used to dynamically create bitmaps in the UI, such as [Text](#).

The default value is [1](#).

This property is valid only when the [renderMode](#) of the [Canvas](#) is set to [WorldSpace](#). Decreasing it from the default value of [1](#) will make the image blurry, while increasing it will make it clearer. For UI placed in world space, if the Canvas is far from the Camera, reducing the value will improve rendering performance. On the other hand, if the Canvas is close to the Camera, increasing the value will give a better look.

### referencePixelsPerUnit

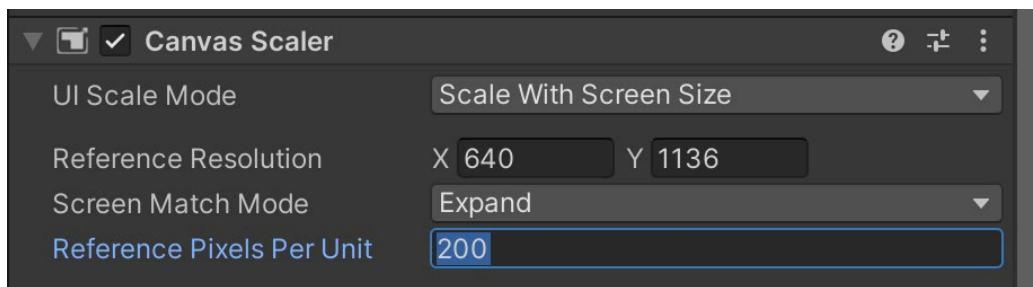
```
public float referencePixelsPerUnit { get; set; }
```

Gets/Sets the value used to calculate the number of pixels per unit.

The default value is [100](#).

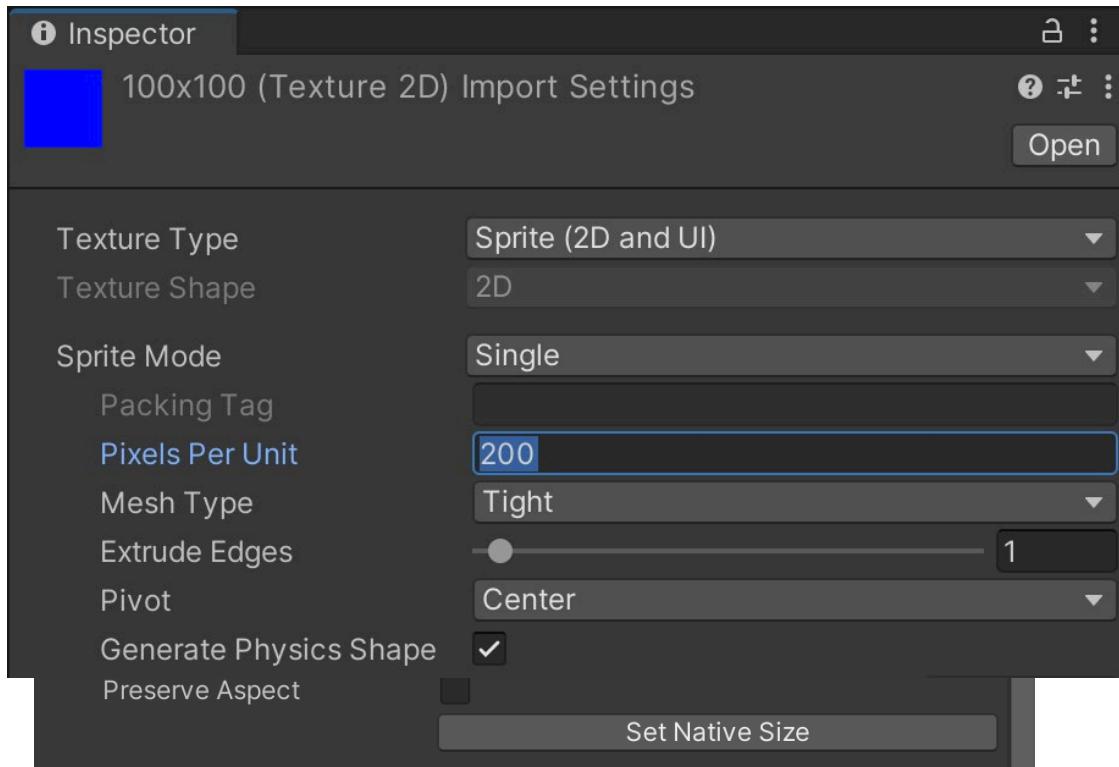
The default value for [pixelsPerUnit](#) in [Sprite](#) (**Pixels Per Unit** in **Inspector**) is [100](#), which means that one pixel in [Sprite](#) corresponds to one pixel in the UI. Let's actually change each value and see how it affects the results.

First, let's change the [referencePixelsPerUnit](#) of [Canvas Scaler](#) to [200](#).



If you click on the **Set Native Size** button of the [Image](#), the [width](#) and [height](#) of the [RectTransform](#) will change to [200](#), and the appearance will become larger.

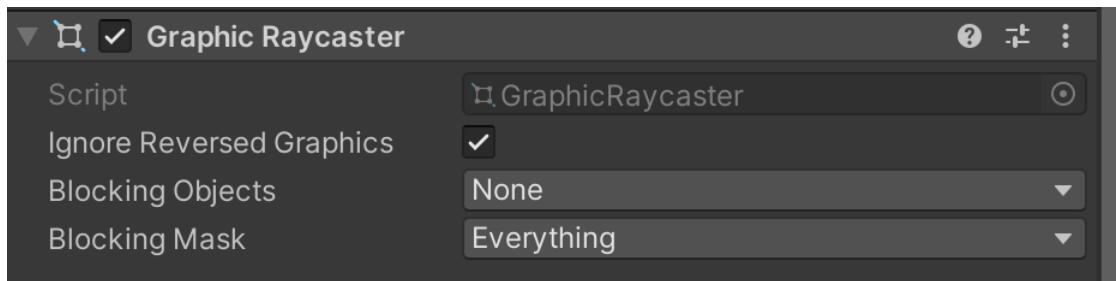
Now, set the **Pixels Per Unit** of the [Sprite](#) asset of [Image](#) to [200](#), and press the **Apply** button.



Pressing the **Set Native Size** button on the [Image](#) again will change the [width](#) and [height](#) of [RectTransform](#) to [100](#) and return it to its original size.

In addition, if you set the [CanvasScaler referencePixelsPerUnit](#) back to [100](#) and click the **Set Native Size** button on the [Image](#), the [width](#) and [height](#) of the [RectTransform](#) will be set to [50](#), halving its size.

## GraphicRaycaster Components



The [GraphicRaycaster](#) monitors all the [Graphics](#) on the canvas to see which one is hit by the raycast. To be more specific, [GraphicRaycaster](#) receives touch events from [EventSystem](#) and determines which of the UI elements in the canvas was touched.

The details will be explained in detail in the *GraphicRaycaster* section of *Chapter 10 EventSystem*, but in this section only the items displayed in the **Inspector** will be briefly explained.

### Ignore Reversed Graphics

Specifies whether to ignore hits on UI elements that face the back side.

The default value is [true](#).

In **World Space Canvas**, let's assume that the canvas is flipped over. By default, the UI element is ignored when touched, but if this property is set to [false](#), it will respond to touches from the back side.

## Blocked Objects

Specifies the type of object to block raycasting.

The default value is [None](#), so all raycast hits will be judged (i.e., not blocked).

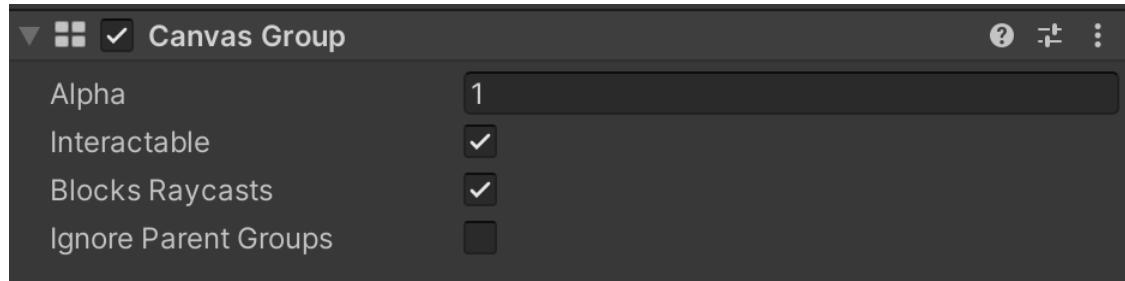
[TwoD](#) will block 2D Physics, [ThreeD](#) will block 3D Physics, and [All](#) will block both 2D and 3D Physics. It will probably be used in screens where there are 2D or 3D objects under the canvas and you want to get their touch detection.

## Blocking Mask

[LayerMask](#) is used to specify which [Layer](#) objects are to be raycasted.

By default, all [Layers](#) are targeted.

## CanvasGroup component



[CanvasGroup](#) is an auxiliary component that can be used to have a specific effect on all underlying UI elements. Specifically, it can change the alpha value, raycast, and [interactable](#) of all UI elements under it. In addition, [CanvasGroups](#) can be nested.

## Properties of CanvasGroup

### alpha

```
public float alpha { get; set; }
```

Gets/Sets the alpha value (opacity) to be applied to the entire group.

The default value is 1.

In practice, this value is the alpha of the vertex color of each UI element. Since alpha values are multiplied, the alpha value used for drawing is the result of multiplying the alpha value obtained from the [CanvasRenderer's GetAlpha\(\)](#), the alpha value of the [Image's color](#), the alpha value of the material's [Tint](#) (if any), and the [alpha](#) of this [CanvasGroup](#). The alpha value will be the result of multiplying the four together.

One point worth noting is that draw calls occur even if the alpha value of the [Image's color](#) or the material's [Tint](#) is 0, but draw calls do not occur if the [CanvasGroup's alpha](#) is 0. This draw call reduction is a function of the [CanvasGroup](#) (presumably). On the other hand, if a UI element is created (by extending the [Graphic](#) class) where the alpha of the vertex colors are all zero, a draw call will occur.

### interactable

```
public bool interactable { get; set; }
```

Gets/Sets whether the UI elements (such as buttons) in this group accept input or not.

The default value is [true](#), that is, it responds to input.

### blocksRaycasts

```
public bool blocksRaycasts { get; set; }
```

Gets/Sets whether or not the entire group is to be judged for raycasting.

The default value is `true`, which is the target of the raycast judgment.

The behavior of `interactable` and `blocksRaycasts` is similar but different: if `interactable` is `false`, the raycast hit judgment itself is still performed, so even if the touched button is unresponsive, no further raycast judgment is performed. On the other hand, if `blocksRaycasts` is `false`, the raycast hit judgment is not performed, so the button underneath it will respond.

### ignoreParentGroups

---

```
public bool ignoreParentGroups { get; set; }
```

Gets/Sets whether or not to ignore the effect of the parent `CanvasGroup` when the `CanvasGroup` is nested.

The default value is `false`, which means it is affected by the `alpha`, `interactable`, and `blockRaycasts` settings of the parent `CanvasGroup`.

## Public methods of CanvasGroup

### IsRaycastLocationValid

```
public bool IsRaycastLocationValid (Vector2 sp, Camera eventCamera);
```

Make a raycast decision for the group.

Returns **true** if the raycast hit at the position of the **screenPoint** given in the argument.

## Chapter 2 Rebuilding UI Elements and Canvas

In this chapter, we will discuss UI elements placed under the [Canvas](#) and the process of rebuilding the layout and geometry of the [Canvas](#), called rebuilding.

The [GameObject](#) of the UI element under the [Canvas](#) has the following characteristics.

1. It has components that inherit from [UIBehaviour](#).
2. It has [RectTransform](#), which is an extension of [Transform](#).
3. If it is a drawing target, it has a [CanvasRenderer](#).

First, let's take a look at [UIBehaviour](#).

## The UIBehaviour class

```
public abstract class UIBehaviour : MonoBehaviour
```

UIBehaviour is a component that defines the common behavior of UI-related components, and the methods defined in [UIBehaviour](#) are as follows.

1. Basic MonoBehaviour lifecycle related methods
  - Awake()
  - OnEnable()
  - Start()
  - OnDisable()
  - OnDestroy()
2. Methods for determining object survival
  - IsActive()
  - IsDestroyed()
3. Methods for the Editor
  - OnValidate()
  - Reset()
4. Methods called during UI element-specific state changes.
  - OnRectTransformDimensionsChange()
  - OnBeforeTransformParentChanged()
  - OnTransformParentChanged()
  - OnDidApplyAnimationProperties()
  - OnCanvasGroupChanged()
  - OnCanvasHierarchyChanged()

The implementations other than "2. Method for judging object survival" are empty, and will be overridden and implemented as necessary by each inherited component. Therefore, in reality, [UIBehaviour](#) is a class similar to an interface.

Next, let's look at the inheritance relationship of UI-related components that inherit from [UIBehaviour](#).

```
MonoBehaviour
← UIBehaviour
← EventSystem
← BaseInput
← BaseInputModule
← PointerInputModule
    ← StandaloneInputModule
← BaseRaycaster
    ← GraphicRaycaster
    ← PhysicsRaycaster
        ← Physics2DRaycaster
← Graphic
    ← MaskableGraphic
        <- Image
        <- RawImage
        <- Text
← Selectable
    ← Button
    ← Dropdown
    ← InputField
    ← Scrollbar
    ← Slider
    ← Toggle
← ScrollRect
← ToggleGroup
← BaseMeshEffect
    ← Shadow
        ← Outline
← PositionAsUV1
```

As you can see, not only UI elements such as [Image](#), [Text](#), and [Button](#), but also components such as [EventSystem](#) inherit [UIBehaviour](#).

## Methods of UIBehaviour

### OnRectTransformDimensionsChange

```
protected virtual void OnRectTransformDimensionsChange()
```

Called when the size of [RectTransform](#) changes.

May be called when the [sizeDelta](#), [offsetMax](#), [offsetMin](#), [anchorMax](#), [anchorMin](#), and [pivot](#) of [RectTransform](#) are changed.

This may be followed by a layout rebuild, which will be discussed later.

### OnBeforeTransformParentChanged

```
protected virtual void OnBeforeTransformParentChanged()
```

It is called before the parent is changed by [SetParent\(\)](#), etc.

See also [OnTransformParentChanged\(\)](#).

### OnTransformParentChanged

```
protected virtual void OnTransformParentChanged()
```

Called when the parent is changed by [SetParent\(\)](#), etc.

It is called not only by the direct parent, but also when the parent's parent or other higher level changes.

In the following sample, [OnBeforeTransformParentChanged\(\)](#) and [OnTransformParentChanged\(\)](#) are called when [SetParent\(\)](#) is called, and when the former is called, the parent has not changed, and when the latter is called, the parent has changed. When the latter is called, we can confirm that the parent has changed.

```

using UnityEngine;
using UnityEngine.Events;

public class OnTransformParentChangedSample : UIBehaviour
{
    private Transform oldParent;

    // Set from Inspector
    public Transform newParent;

    protected override void Start()
    {
        base.Start();

        if (transform.parent != null)
        {
            oldParent = transform.parent as RectTransform;
        }

        transform.SetParent(newParent);
    }

    protected override void OnBeforeTransformParentChanged()
    {
        Debug.Log("OnBeforeTransformParentChanged IsParentOld? " + (transform.parent == oldParent) +
        ", IsParentNew? " + (transform.parent == newParent));
    }

    protected override void OnTransformParentChanged()
    {
        Debug.Log("OnTransformParentChanged IsParentOld? " + (transform.parent == oldParent) +
        ", IsParentNew? " + (transform.parent == newParent));
    }
}

```

*Console output results*

```
OnBeforeTransformParentChanged IsParentOld? True, IsParentNew? False  
OnTransformParentChanged IsParentOld? False, IsParentNew?
```

### OnDidApplyAnimationProperties

```
protected virtual void OnDidApplyAnimationProperties();
```

Called when some property is changed by an animation such as [Animation](#) or [Timeline](#).

We don't know which properties are changed when this method is called. This is why we do both Layout rebuild and Graphic rebuild for [Graphic](#) components, just in case. This is why using [Animations](#) and [Timelines](#) is not recommended in uGUI.

### OnCanvasGroupChanged

```
protected virtual void OnCanvasGroupChanged();
```

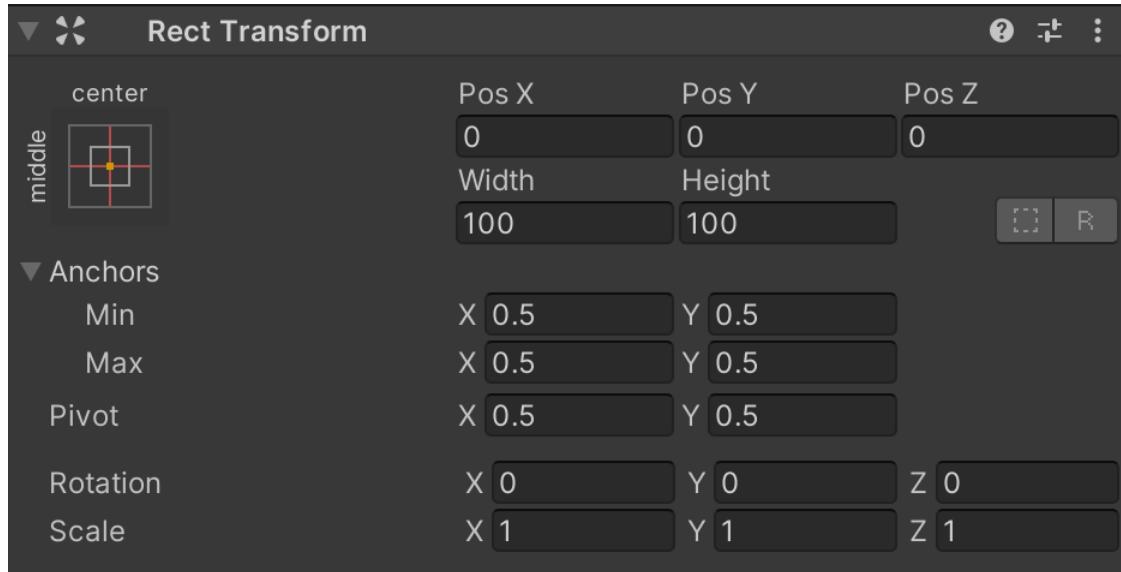
Called when the [alpha](#), [interactable](#), [blocksRaycasts](#), or [ignoreParentGroups](#) of the [CanvasGroup](#) to which it belongs are changed.

### OnCanvasHierarchyChanged

```
protected virtual void OnCanvasHierarchyChanged();
```

Called when a [GameObject](#) on the [Canvas](#) to which it belongs or a parent [Canvas](#) above it is switched between active and inactive. Or, if the [Canvas](#) to which it belongs is a sub-Canvas, it is called when the [overrideSorting](#) of that [Canvas](#) is changed.

## RectTransform component



The [RectTransform](#) component is a subclass of [Transform](#), and is used for UI layout.

If you create an empty object in the [Canvas](#) in the Editor, the [RectTransform](#) will be attached automatically. On the other hand, if you create an object from a script with `new GameObject()` and parent it to a [Canvas](#) or UI element, [RectTransform](#) will not be attached. In this case, you need to explicitly attach [RectTransform](#) by calling `AddComponent<RectTransform>()`, or attach a component such as [Image](#).

**Note**

[Graphic](#), the parent class (or more precisely, the parent of the parent) of [Image](#), has the `[RequireComponent(typeof(RectTransform))]` attribute, so attaching it will automatically attach [RectTransform](#).

`RectTransform` can be obtained from the script via `GetComponent`.

```
RectTransform rectTransform = GetComponent<RectTransform>();
```

However, `RectTransform` is a subclass of `Transform`. However, since `RectTransform` is a subclass of `Transform`, you can actually use

```
RectTransform rectTransform = transform as RectTransform;
```

It is also possible to get it by casting it as in the following example, which has slightly better performance.

In any case, getting `RectTransform` consumes a lot of CPU time, so you can avoid wasting CPU time by caching it in member variables.

## Properties of RectTransform

### anchoredPosition / anchoredPosition3D

```
public Vector2 anchoredPosition { get; set; }
```

Gets/Sets the position relative to the parent.

According to the script reference, it is "the pivot position of RectTransform relative to the anchored reference point. The `anchoredPosition3D` converts the `anchoredPosition` to `Vector3` and returns `localPosition.z` as Z.

When the anchor or pivot is changed, the `anchoredPosition` will be recalculated and the value will be changed.

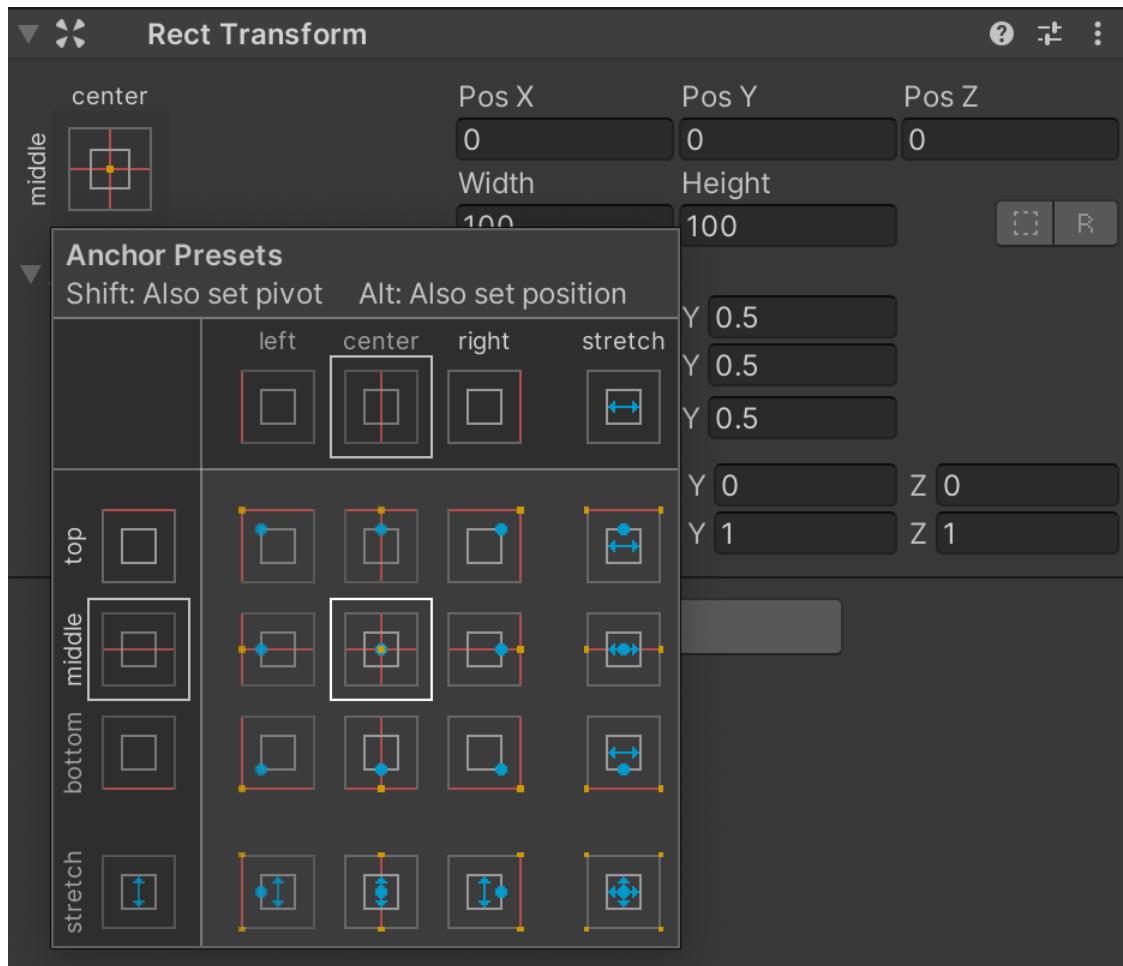
### anchorMax / anchorMin

```
public Vector2 anchorMax { get; set; }
public Vector2 anchorMin { get; set; }
```

The **anchor** is a number that indicates the location of the four corners of the region as seen by the parent.

The upper right is represented by `(anchorMax.x, anchorMax.y)`, and the lower left by `(anchorMin.x, anchorMin.y)`.

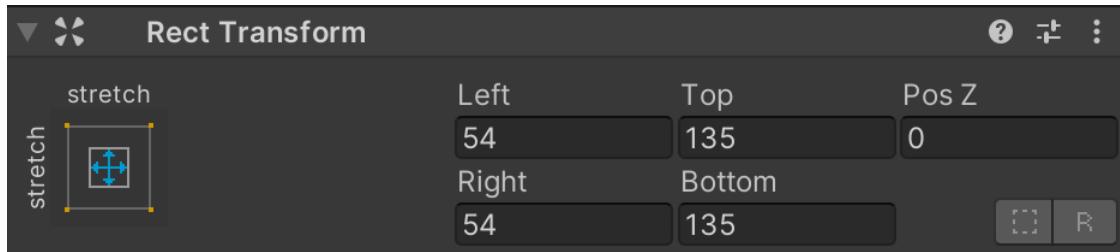
By default, the top-left corner is `(0.5f, 0.5f)` and the top-right corner is also `(0.5f, 0.5f)`, which means that the position relative to the parent is centered.



The anchor is a number used to specify the position relative to the parent. If it is a UI you want to display at the top of the screen, such as a header, you will probably want to specify the distance from the top of the parent, and if it is a UI you want to display at the bottom of the screen, such as a footer, you will probably want to specify the distance from the bottom of the parent. By setting these anchors appropriately, the position of the UI will not be disturbed even if the aspect ratio of the screen changes.

For the vertical direction of the anchor, **stretch** can be specified in addition to **top** for **top** alignment, **middle** for center alignment, and **bottom** for bottom alignment. For the left-right direction, you can specify the left justification (**left**), center justification (**center**), and right justification (**right**), as well as stretch. When you select one of the presets, the values corresponding to **anchorMax** and **anchorMin** will be changed in the **Inspector**.

In the case of **stretch**, the display in **Inspector** will change.



When the left and right sides are **stretched**, **Left** is displayed instead of **PosX** to indicate the distance from the left edge, and **Right** is displayed instead of **Width** to indicate the distance from the right edge. When the top and bottom are **stretched**, the distance from the top edge is displayed as **Top** instead of **PosY**, and the distance from the bottom edge is displayed as **Bottom** instead of **Height**. For example, if **Left** and **Right** are set to **0**, the area will expand to the full left and right, and if **Bottom** and **Top** are set to **0**, it will expand to the full top and bottom.

## pivot

```
public Vector2 pivot { get; set; }
```

Gets/Sets the center position of the rotation.

The default value is **(0.5f, 0.5f)**, which represents the center.

If this value is set to **(0, 0)**, the center of rotation will be the lower left, and if it is set to **(1, 1)**, the center of rotation will be the upper right.

For more information on the relationship between anchors and pivots, please refer to the diagram on Tsuchiya Tsukasa's blog (written in Japanese).

<https://someiyoshino.info/entry/2021/02/07/211440>

## rect

```
public Rect rect { get; }
```

Get the relative position of the lower left corner of the rectangle from the pivot, as well as its width and height.

If the pivot is at the center (0.5f, 0.5f), then `x` will be `width * -0.5f` and `y` will be `rect.height`. `rect` values cannot be edited directly. You cannot edit the value of `rect` directly. If you want to change the width or height from the script, you have to change the `sizeDelta`.

## sizeDelta

```
public Vector2 sizeDelta { get; set; }
```

Gets/Sets the difference between the size of the rectangle and the distance between the anchors.

The size of a rectangle is the size indicated by (`rect.width`, `rect.height`).

The distance between `x` anchors is `anchorMax.x - anchorMin.x`, and the distance between `y` anchors is `anchorMax.y - anchorMin.y`. In other words, if the left and right sides of the anchor are **center** (i.e., `anchorMax.x = 0.5f` and `anchorMin.x = 0.5f`), the distance between the `x` anchors is **0**, and if the top and bottom are **middle** (`anchorMax.y = 0.5f` and `anchorMin.y = 0.5f`), the distance between the `y` anchors is **0**.

If the distance between the `x` anchors is **0**, the value of `sizeDelta.x` will be equal to `rect.width`, and if the distance between the `y` anchors is **0**, the value of `sizeDelta.y` will be equal to `rect.height`.

Now, let's consider an example where the distance between the anchors is not zero.

Let's assume that the **Canvas** width is **1280** pixels and the height is **760** pixels, and you have set up a **RectTransform** with a **stretch** anchor with a gap of **100** pixels on the top and bottom and **60** pixels on the left and right.



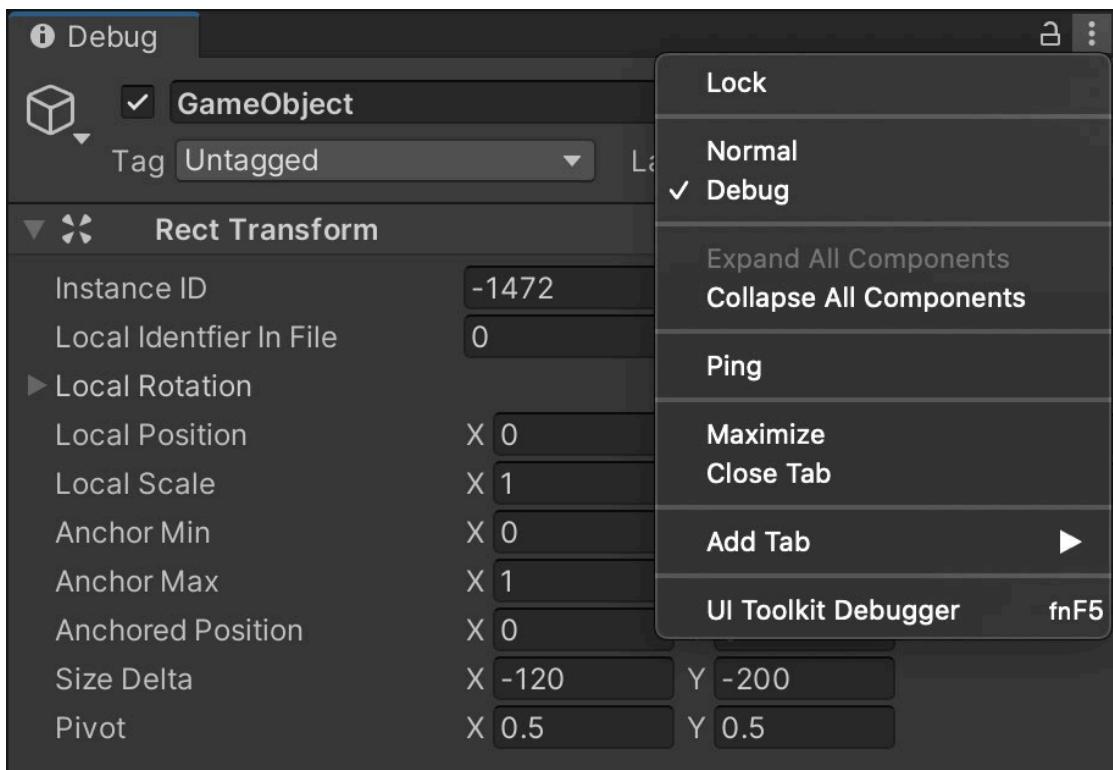
In this case, the size of the rectangle would be  $(1280 - 60 - 60, 760 - 100 - 100) = (1160, 560)$ .

The distance between the anchors is the size of the parent rectangle, and if there are no **stretch** anchors between this **RectTransform** and the **Canvas**, the distance between the anchors is equal to the size of the **Canvas**, which is **(1160, 560)**. - If there are no stretch anchors between the **Canvas**,

the distance between the anchors is equal to the size of the Canvas, which is (1160, 560). `sizeDelta` is the size of the rectangle, so  $(1160, 560) - (1280, 760) = (-120, -200)$ .

Note

If you want to see the `sizeDelta` from **Inspector**, you can do so by setting the display mode to **Debug**.



If the parent contains a `RectTransform` with a **stretch** anchor, the distance between the anchors will be the size of the `RectTransform`'s rectangle.

$\langle\text{Distance Between Anchors}\rangle = (\text{Left of Inspector's RectTransform}) + \text{rect.width} + (\text{Right of Inspector's RectTransform})$

If you want to change the width or height of the rectangle, you should change the `sizeDelta` instead of `rect`, but if the anchor is **stretch**, it is complicated to handle as explained above.

- If the anchor is not **stretch**, `sizeDelta.x` will match `rect.width` and `sizeDelta.y` will match `rect.height`.
- If the anchor is **stretch**, `sizeDelta.x` will be  $-1 * ((\text{Left on Inspector}) + (\text{Right on Inspector}))$ , and `sizeDelta.y` will be  $-1 * ((\text{Top on Inspector}) + (\text{Bottom on Inspector}))$ .

`offsetMax / offsetMin`

```
public Vector2 offsetMax { get; set; }
```

`offsetMax` is the offset of the upper right corner of the rectangle with respect to the upper right anchor, and `offsetMin` is the offset of the lower left corner of the rectangle with respect to the lower left anchor.

These values are not directly displayed in the **Inspector**.

`offsetMax` is the value of

```
anchoredPosition + Vector2.Scale(sizeDelta, Vector2.one - pivot);
```

and `offsetMin` is

```
anchoredPosition - Vector2.Scale(sizeDelta, pivot);
```

This is calculated as

`Scale` returns the result of multiplying the `x` and `y` components of each.

```
Vector2.Scale(a, b) = new Vector(a. x * b. x, a. y * b. y);
```

If you change the value of `offsetMax` or `offsetMin`, the `anchoredPosition` and `sizeDelta` will be changed.

If `offsetMax` is changed, `sizeDelta` and `anchoredPosition` will be as follows.

```
Vector2 vector = offsetMax - (anchoredPosition + Vector2.Scale(sizeDelta, Vector2.one - pivot))
;
sizeDelta += vector;
anchoredPosition += Vector.Scale(vector, pivot);
```

If `offsetMin` is changed, `sizeDelta` and `anchoredPosition` will be as follows.

```
Vector2 vector = value - (anchoredPosition - Vector2.Scale(sizeDelta, pivot));
sizeDelta -= vector;
anchoredPosition += Vector.Scale(vector, Vector2.one - pivot);
```

## Public method of RectTransform

### ForceUpdateRectTransforms

```
public extern void ForceUpdateRectTransforms();
```

Forces a recalculation of [RectTransforms](#) internal data.

Calling this method will cause a Canvas rebuild.

### GetLocalCorners

```
public void GetLocalCorners(Vector3[] fourCornersArray);
```

Get the coordinates of the four corners in local coordinates.

For example, let's assume that [rect.size](#) is [\(80, 100\)](#).

In this case, if pivot is [\(0.5f, 0.5f\)](#), the following array will be returned.

```
(-40.0, -50.0, 0.0)  
(-40.0, 50.0, 0.0)  
(40.0, 50.0, 0.0)  
(40.0, -50.0, 0.0)
```

On the other hand, if pivot is [\(0.0f, 0.0f\)](#), the following array will be returned.

```
(0.0, 0.0, 0.0)  
(0.0, 100.0, 0.0)  
(80.0, 100.0, 0.0)  
(80.0, 0.0, 0.0)
```

Since this property is a local coordinate, it is not affected by the parent.

## GetWorldCorners

```
public void GetWorldCorners(Vector3[] fourCornersArray);
```

Get the coordinates of the four corners in world coordinates.

The result of [GetLocalCorners\(\)](#) plus the world coordinates will be returned.

## SetInsetAndSizeFromParentEdge

```
public void SetInsetAndSizeFromParentEdge(Edge edge, float inset, float size);
```

By selecting one of the edges of the rectangle and specifying the starting point and size, the anchor, position, and size of the rectangle are changed. In order to apply the changes, we need to make one call each to the [edge](#), specifying the vertical and horizontal axes, respectively.

It's complicated, so it will be easier to understand if you see a real example.

When the following code is executed,

```
var rectTransform = transform as RectTransform;
rectTransform.SetInsetAndSizeFromParentEdge(RectTransform.Edge.Top, 10, 100);
rectTransform.SetInsetAndSizeFromParentEdge(RectTransform.Edge.Left, 20, 200);
```

the rectangle will be modified as follows.

- The anchor is top/left.
- From 20 pixels below the top edge of the Canvas, downward to a height of 100 pixels.
- From 10 pixels right of the left edge of the Canvas, 200 pixels wide to the right.

Now, if you run the following code

```
var rectTransform = transform as RectTransform;
rectTransform.SetInsetAndSizeFromParentEdge(RectTransform.Edge.Bottom, 10, 100);
rectTransform.SetInsetAndSizeFromParentEdge(RectTransform.Edge.Right, 20, 200);
```

the rectangle will be modified as follows.

- The anchor is bottom/right.
- From 20 pixels above the bottom edge of the Canvas, upward to a height of 100 pixels.
- From 10 pixels left of the right edge of the Canvas, 200 pixels wide to the left

## SetSizeWithCurrentAnchors

---

```
public void SetSizeWithCurrentAnchors(Axis axis, float size);
```

Change the width or height while keeping the current anchor.

For example, the following code will change the width to 200 pixels without changing the anchor.

```
var rectTransform = transform as RectTransform;  
rectTransform.SetSizeWithCurrentAnchors(RectTransform.Axis.Horizontal, 200);
```

## RectTransform events

### reapplyDrivenProperties

```
public static event ReapplyDrivenProperties reapplyDrivenProperties;
```

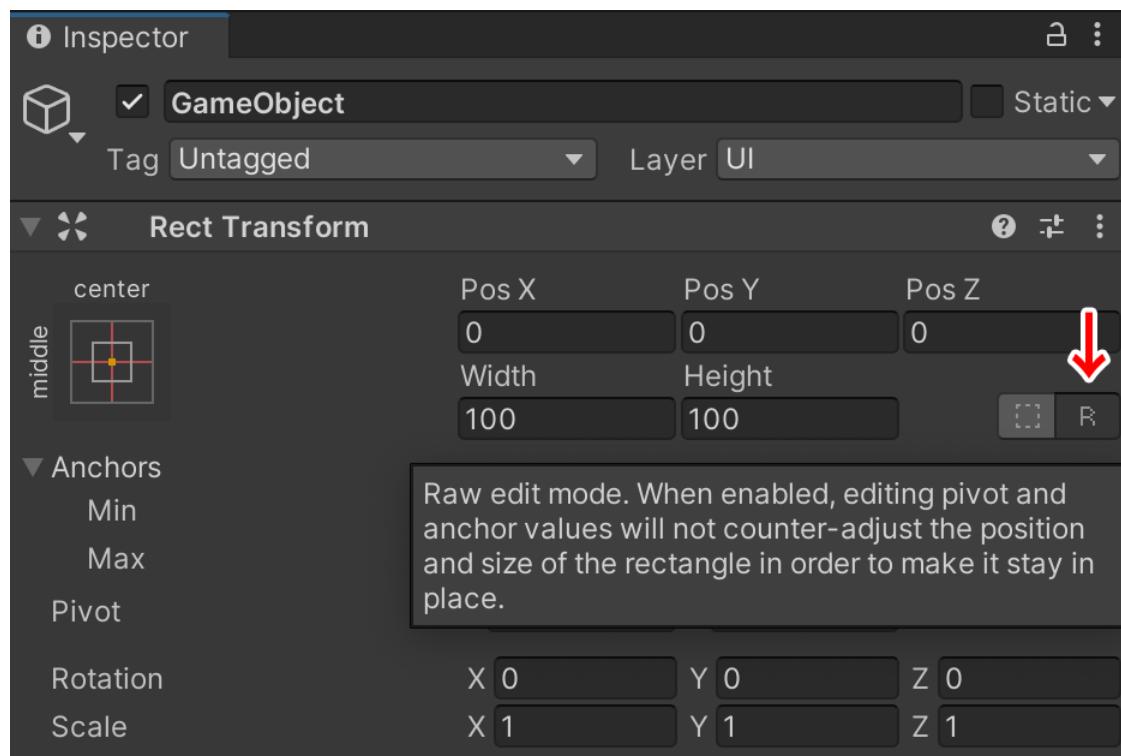
This callback is called when the driven property of [RectTransform](#) is reapplied. In fact, it is called when the [GameObject](#) is switched to active or inactive.

A driven property is a property that cannot be edited manually due to Auto Layout. The details of driven properties are explained in *Chapter 9, "RectTransform's Driven Properties in Auto Layout.*

## Raw Edit mode for RectTransform

In normal mode, changing the anchor or pivot will change the `anchoredPosition` and size (but not the apparent position). However, when **Raw Edit mode** is enabled, changing the pivot or anchor will not change the position or size.

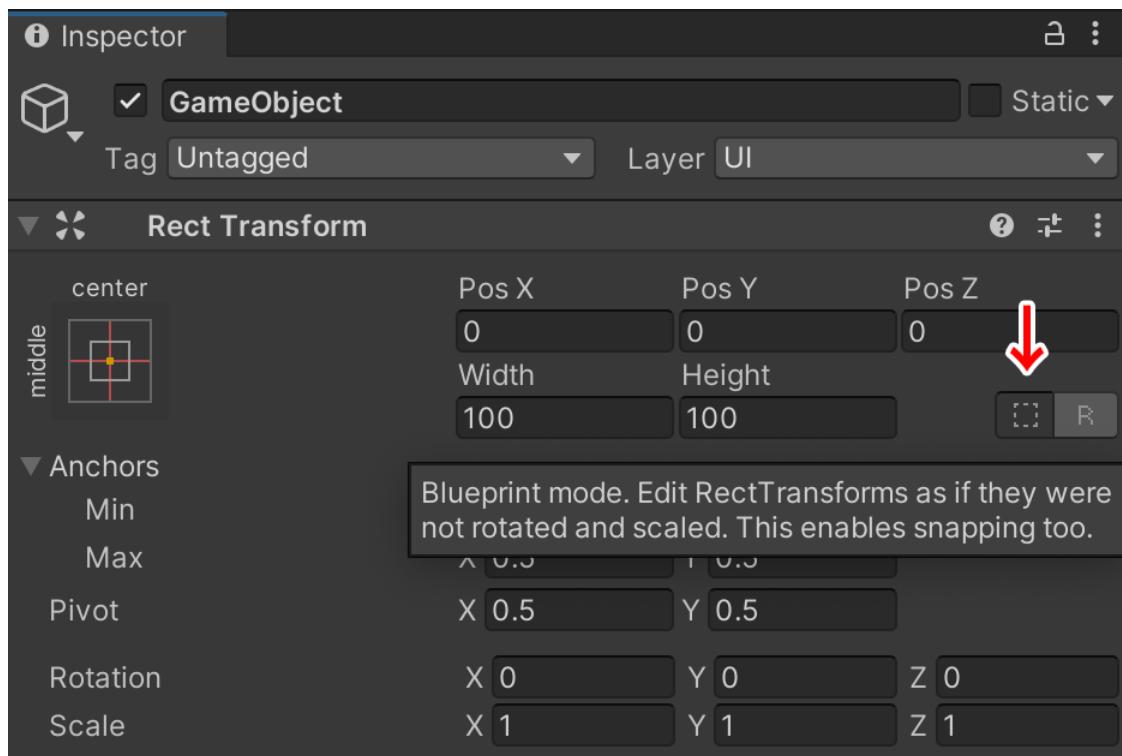
To enable **Raw Edit mode**, press the button labeled "R" to the right of **Height**.



## Blue Print mode in RectTransform

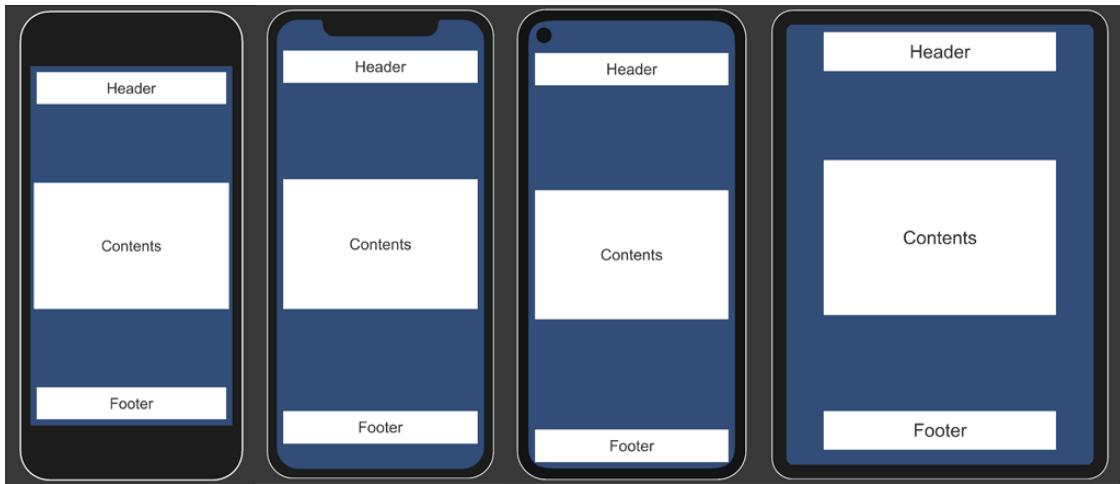
Even if the **Rotation** or **Scale** is changed, the original rectangle will be visible in the **SceneView** with the **Rect Tool** selected. Also, the snap operation using the original state will be effective.

The **Rect Tool** can be activated by pressing the square button in the upper right corner of the Editor window; to activate **Blue Print mode**, press the dotted square button to the right of **Height**.



## Enabling Variable Resolution on Smart Devices

We assume a UI that displays player information in the header section at the top of the screen, menu buttons in the footer section at the bottom of the screen, and the main information in the center of the screen, just like a typical game for smart devices.

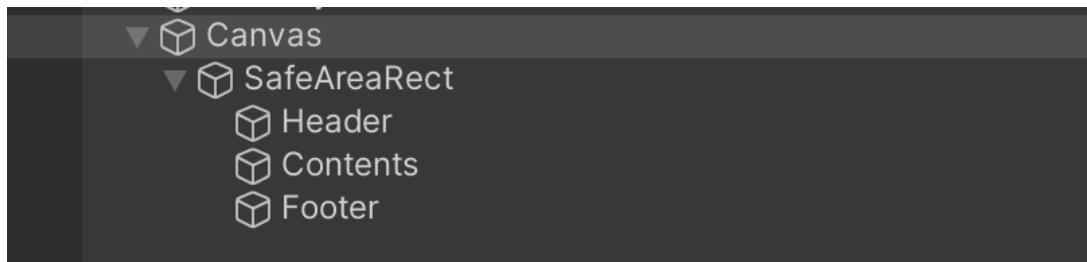


If you want to avoid black frames on the top and bottom on vertical smartphones or on the left and right on tablets, you will have to get a little creative with your layout. We will also have to deal with the notch (and its associated safe area) on iPhoneX and later.

Here we assume a vertical game, but the basic idea is the same for a horizontal game.

1. Decide on the device to be used as the UI placement standard. In this case, we'll use the iPhone5 (W 640 x H 1136).
2. Set the `referenceResolution` of `CanvasScaler` to `(640, 1136)`.
3. Set the `scaleMode` of `CanvasScaler` to `ScaleWithScreenSize`.
4. Set the `CanvasScaler`'s `screenMatchMode` to `Expand`. This will create a gap on the left and right sides if the image resolution is wider than the `referenceResolution`.
5. Place an empty game object directly under the `Canvas`, and attach the `SafeAreaRect` shown below.
6. Place the UI under the `SafeAreaRect`.
  - Header anchor is top/center
  - Anchors for Contents are middle/center.
  - Footer anchor is bottom/center (bottom-aligned)

The hierarchical structure is as follows.



The source code for the `SafeAreaRect` component is shown below.

```
using UnityEngine;

[ExecuteAlways]
[RequireComponent(typeof(RectTransform))]
public class SafeAreaRect : MonoBehaviour
{
    private RectTransform rectTransform;
    private Canvas canvas;
    private Rect safeArea = new Rect(0, 0, 1, 1);

#if UNITY_EDITOR
    private void OnEnable()
    {
        UnityEditor.EditorApplication.update += UpdateSafeArea;
    }

    private void OnDisable()
    {
        UnityEditor.EditorApplication.update -= UpdateSafeArea;
    }
}

// Returns false if the resolution is wrong in the Editor
private bool IsScreenResolutionCorrect()
{
    if (canvas == null)
    {
        canvas = GetComponentInParent<Canvas>();
    }
}
```

```

if (canvas.pixelRect.width == 0 || canvas.pixelRect.height == 0)
{
    return false;
}

string[] editorScreenRes = UnityEditor.UnityStats.screenRes.Split('x');
if (editorScreenRes.Length >= 2)
{
    if (int.TryParse(editorScreenRes[0], out int editorScreenWidth))
    {
        if (int.TryParse(editorScreenRes[1], out int editorScreenResHeight))
        {
            if (Screen.width == editorScreenWidth && Screen.height == editorScreenResHe
ight)
            {
                return true;
            }
        }
    }
    return false;
}
#endif

private void Update()
{
    // Since Screen.safeArea may not return the correct value even during Start() on some Andro
id devices, we check if Screen.safeArea has not changed during Update().
    UpdateSafeArea();
}

private void UpdateSafeArea()
{
#if UNITY_EDITOR
    if (!IsScreenResolutionCorrect())
    {
        return;
    }
#else
    if (safeArea == Screen.safeArea)
    {

```

```

        return;
    }
#endif
if (canvas == null)
{
    canvas = GetComponentInParent<Canvas>();
}

if (rectTransform == null)
{
    rectTransform = transform as RectTransform;
}

float anchorMinX = Screen.safeArea.position.x / canvas.pixelRect.width;
float anchorMinY = Screen.safeArea.position.y / canvas.pixelRect.height;

float anchorMaxX = (Screen.safeArea.position.x + Screen.safeArea.size.x) / canvas.pixelRect.width;
float anchorMaxY = (Screen.safeArea.position.y + Screen.safeArea.size.y) / canvas.pixelRect.height;

rectTransform.anchorMin = new Vector2(anchorMinX, anchorMinY);
rectTransform.anchorMax = new Vector2(anchorMaxX, anchorMaxY);

safeArea = Screen.safeArea;
}
}

```

In this example, I set `CanvasScaler's screenMatchMode` to `Expand`, but if you set it to `MatchWidthOrHeight` and change the `matchWidthOrHeight` dynamically, you can achieve behavior similar to `Expand`.

```

using UnityEngine;
using UnityEngine.UI;

[ExecuteAlways]
[RequireComponent(typeof(Canvas))]
public class SmartDeviceCanvasScaleHelper : MonoBehaviour
{
    // Based on the number of pixels in iPhone 5 as an example

```

```

[SerializeField] private float standardWidth = 640.0f;
[SerializeField] private float standardHeight = 1136.0f;

private CanvasScaler scaler;
private Canvas canvas;

private void Start()
{
    UpdateScaler();
}

#if UNITY_EDITOR
private void OnEnable()
{
    UnityEditor.EditorApplication.update += UpdateScaler;
}

private void OnDisable()
{
    UnityEditor.EditorApplication. update -= UpdateScaler;
}

private void Update()
{
    UpdateScaler();
}
#endif

private void UpdateScaler()
{
    if (scaler == null)
    {
        scaler = GetComponent<CanvasScaler>();
        scaler.uiScaleMode = CanvasScaler.ScaleMode.ScaleWithScreenSize;
        scaler.referenceResolution = new Vector2(standardWidth, standardHeight);
        scaler.screenMatchMode = CanvasScaler.ScreenMatchMode.MatchWidthOrHeight;
    }

    if (canvas == null)
    {
        canvas = GetComponent<Canvas>();
    }
}

```

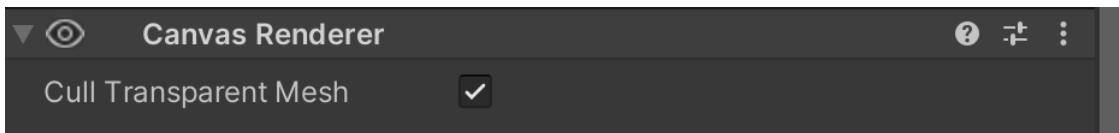
```
if (canvas.pixelRect.width == 0 || canvas.pixelRect.height == 0)
{
    return;
}

float standardAspectRatio = standardWidth / standardHeight;
float currentAspectRatio = canvas.pixelRect.width / canvas.pixelRect.height;

// a device with a short height is considered a tablet device
if (currentAspectRatio > standardAspectRatio)
{
    // match the height (there will be a horizontal gap)
    scaler.matchWidthOrHeight = 1;
}
else
{
    // Match the width (it will grow vertically)
    scaler.matchWidthOrHeight = 0;
}
}
```

By attaching this [CustomExpandCanvasScalerHelper](#) to the [Canvas](#) and customizing it, it is possible to open a gap on the left and right for a specific device, or conversely, to stretch it vertically.

## CanvasRenderer component



```
[NativeClass ("UI::CanvasRenderer")]
[NativeHeader ("Modules/UI/CanvasRenderer.h")]
public sealed class CanvasRenderer : Component
```

The [CanvasRenderer](#) is a component for drawing drawable UI elements contained in a [Canvas](#), and since it is a native component written in C++ rather than C#, it belongs to the [UnityEngine.UI](#) namespace, it belongs to the [UnityEngine](#) namespace.

Contrary to intuition, [CanvasRenderer](#) is not a subclass of [Renderer](#), so it cannot be cast and converted.

```
/// Error!
var canvasRenderer = GetComponent<Renderer>() as CanvasRenderer;
```

Components of UI elements such as [Images](#) are just components for convenient handling of the [CanvasRenderer](#), and drawing can be controlled by touching the [CanvasRenderer](#) directly.

## Properties of CanvasRenderer

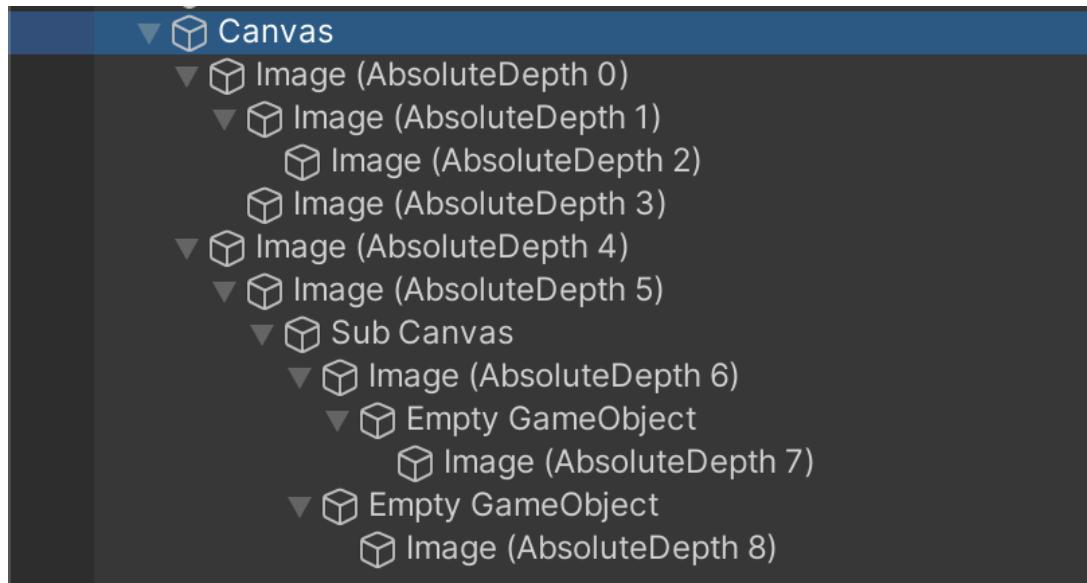
### absoluteDepth

```
public int absoluteDepth { get; }
```

Gets the order of the [CanvasRenderer](#) in the [Canvas](#) hierarchy including [sub-Canvas](#).

The topmost element directly under the [Canvas](#) is **0**, its children or those below it are **1**, and so on. If there is a [GameObject](#) in the hierarchy that does not have a [CanvasRenderer](#), it will not be counted.

The sample **Hierarchy** is shown below.



If the [GameObject](#) is inactive or otherwise not rendered by the [Canvas](#), this property will return **-1**. This property will also return **-1** before the first frame is completed.

### Note

The `absoluteDepth` value should not be referenced in `Awake()` or `Start()`, since it seems that the value is not determined until one frame has passed since the first frame.

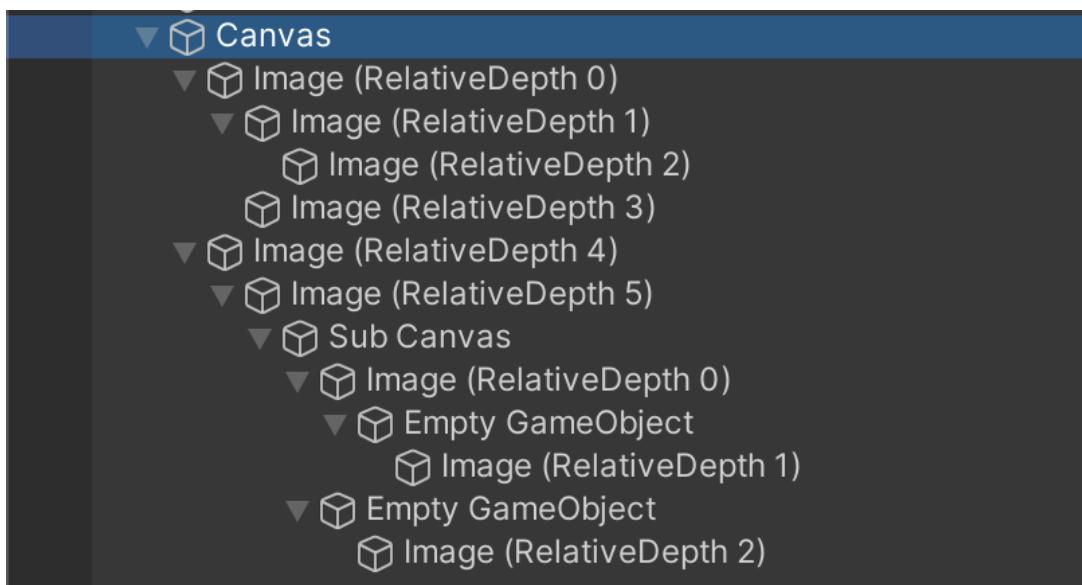
In the uGUI built-in C# code, this property is used by `GraphicRaycaster` via `Graphic.depth`.

### relativeDepth

```
public int relativeDepth { get; }
```

It returns the order of the `CanvasRenderers` in the `Canvas` hierarchy to which this `CanvasRenderer` belongs, with the topmost element directly under the `Canvas` being `0`, its children or those below it being `1`, and so on. If there is a `GameObject` in the hierarchy that does not have a `CanvasRenderer`, it will not be counted.

The sample **Hierarchy** is shown below.



If the `GameObject` is inactive or otherwise not rendered by the `Canvas`, this property will return `-1`. This property will also return `-1` before the first frame is completed.

#### Note

As with `absoluteDepth`, `relativeDepth` value should not be referenced in `Awake()` or `Start()`, since it is not determined until one frame has passed since the first frame.

## clippingSoftness

```
public Vector2 clippingSoftness { get; set; }
```

Gets/Sets the range of the Soft Mask (edge blur function) of `MaskableGraphic` introduced in Unity 2020.1.

This value is passed to `_MaskSoftnessX` and `_MaskSoftnessY` in the UI shader.

## cull

```
public bool cull { get; set; }
```

Indicates whether or not to ignore (cull) the geometry generated by this `CanvasRenderer`.

If you set this property to `true`, it will not be rendered. In that case, no draw call will be generated.

Note, however, that the raycast judgment is still alive, so the touch judgment remains alive even if rendering is not done.

## cullTransparentMesh

```
public bool cullTransparentMesh { get; set; }
```

Gets/Sets whether the geometry will be ignored if the alpha of all vertex colors in the mesh is `0`.

If this property is set to `true` from a script, the image will not be rendered if the alpha of all vertex colors is `0` (for example, if the alpha of the `Color` property of the `Image` is `0`). No draw call will be generated in that case.

Note that, as with `cull`, the touch decision remains alive even if it is not rendered.

## hasMoved

---

```
public bool hasMoved { get; }
```

Returns `true` when a change occurs that invalidates the position of the generated geometry.

"A change that invalidates the position of the generated geometry" is, for example, a change in `anchoredPosition`. Note that if only the size is changed, `true` will not be returned.

## hasPopInstruction

---

```
public bool hasPopInstruction { get; set; }
```

Gets/Sets whether `PopMaterial` should be used to render the child UI elements again after rendering all of them.

The default value is `false`.

Set to `true` in `Mask.GetModifiedMaterial()` when using a stencil mask.

## hasRectClipping

---

```
public bool hasRectClipping { get; }
```

Returns `true` if clipping of the rectangle is enabled.

Clipping can be enabled or disabled with `EnableRectClipping()` or `DisableRectClipping()`.

`RectMask2D` component calls these methods via `MaskableGraphic`.

## materialCount

---

```
public int materialCount { get; set; }
```

Gets/Sets the number of materials used by the [CanvasRenderer](#).

### popMaterialCount

---

```
public int popMaterialCount { get; set; }
```

Gets/Sets the number of materials to use for the pop instruction.

This property is set to [true](#) in [Mask.GetModifiedMaterial\(\)](#) when using a stencil mask.

## Public methods of CanvasRenderer

### Clear

```
public void Clear();
```

Delete all internally cached vertices.

The [Graphic](#) class calls [OnDisable\(\)](#) and others to clear the mesh vertices set by [SetMesh\(\)](#).

### EnableRectClipping / DisableRectClipping

```
public void EnableRectClipping(Rect rect);
public void DisableRectClipping();
```

Enable/Disable rectangle clipping.

If enabled, the outside of the given rectangle will not be rendered.

As a special usage, by passing [Rect.zero](#) as an argument, you can make this [CanvasRenderer](#) stop drawing. To cancel this, simply call [DisableRectClipping\(\)](#).

### SetColor / GetColor / SetAlpha / GetAlpha

```
public void SetColor(Color color);
public Color GetColor();
public void SetAlpha(float alpha);
public float GetAlpha();
```

Set or get the color/alpha value.

The color set here will be multiplied by the vertex color set in the [Graphic](#) class. For example, the color output by [Image](#) is as follows.

(Image's Color) \* (Image's Material's Tint Color) \* (CanvasRender's Color)

Furthermore, in the case of alpha values, the [alpha](#) of the [CanvasGroup](#) will also be multiplied. If the alpha value of the [CanvasRenderer](#) is [0](#), no draw call will be generated.

### GetInheritedAlpha

```
public float GetInheritedAlpha();
```

Get the resulting value by multiplying the alpha values of all parent [CanvasGroups](#).

If a [CanvasGroup](#) is attached to the same [GameObject](#), its alpha value will also be included.

For example, suppose you have a parent named *CanvasGroup2* above your hierarchy, and another parent named *CanvasGroup1*.

```
CanvasGroup1  
  CanvasGroup2  
    MyGameObject
```

In this case, the value obtained by [GetInheritedAlpha\(\)](#) is as follows.

(Alpha of CanvasGroup1) \* (Alpha of CanvasGroup2)

### SetMaterial / GetMaterial

```
public void SetMaterial(Material material, Texture texture);  
public void SetMaterial(Material material, int index);  
public Material GetMaterial(int index);
```

Sets/gets the material to be used by this renderer.

If a texture is specified, that texture will be used instead of the [MainTex](#) of this material.

In a normal [Renderer](#), accessing the [material](#) property will create a copied instance, so [sharedMaterial](#) is sometimes used to prevent the creation of an instance. In [CanvasRenderer](#), copying does not occur due to material access. Conversely, [CanvasRenderer's material](#) behaves like [sharedMaterial](#), so if you want to change individual shader properties, you will need to prepare a separate material.

### SetPopMaterial / GetPopMaterial

---

```
public void SetPopMaterial(Material material, int index);  
public Material GetPopMaterial(int index);
```

Set/Get the material to use for the pop instruction.

This material is used when using a stencil mask.

### SetAlphaTexture

---

```
public void SetAlphaTexture(Texture texture);
```

Sets the alpha texture to be set to the [\\_AlphaTex](#) property of the shader.

It is used when the image format is ETC1.

### SetMesh

---

```
public void SetMesh(Mesh mesh);
```

Set the [Mesh](#) to be used by this renderer.

This [Mesh](#) must be **read/write enabled**.

### SetTexture

---

```
public void SetTexture(Texture texture);
```

Sets the texture to be used by this renderer.

The texture specified here will be set to the `_MainTex` property of the built-in UI shader.

There is no `GetTexture()` method provided, so to get the texture,

- if a material has been set, use `GetMaterial().mainTexture` property
- If no material has been set, use `mainTexture` property of the `Graphic` class

## Static methods of CanvasRenderer

### AddUIVertexStream

```
public static void AddUIVertexStream (List<UIVertex> verts, List<Vector3> positions, List<Color32> colors, List<Vector4> uv0S, List<Vector4> uv1S, List<Vector3> normals, List<Vector4> tangents);  
  
public static void AddUIVertexStream (List<UIVertex> verts, List<Vector3> positions, List<Color32> colors, List<Vector4> uv0S, List<Vector4> uv1S, List<Vector4> uv2S, List<Vector4> uv3S, List<Vector3> normals, List<Vector4> tangents);
```

It takes a list of vertices and splits it into a list of vertex components (positions, colors, uv0s, uv1s, normals, tangents). AddUIVertexStream().

Initialize the Mesh using `m_Positions` obtained by `AddUIVertexStream()`, and pass it to `CanvasRenderer.SetMesh()` to achieve drawing.

The following is a sample code for drawing on a Canvas using `AddUIVertexStream()`.

```
using System.Collections.Generic;  
using UnityEngine;  
  
// Directly access the CanvasRenderer and draw on the Canvas  
public class SetMeshToCanvasRendererSample : MonoBehaviour  
{  
    private void Start()  
    {  
        Canvas.willRenderCanvases += MyRebuild;  
    }  
  
    private void OnDestroy()  
    {  
        Canvas.willRenderCanvases -= MyRebuild;  
    }  
  
    // Do the same thing as Graphic.Rebuild()  
    void MyRebuild()
```

```

{
    var canvasRenderer = gameObject.GetComponent<CanvasRenderer>();
    if (canvasRenderer == null)
    {
        canvasRenderer = gameObject.AddComponent<CanvasRenderer>();
    }

    // List of vertices to be drawn
    List<UIVertex> verts = new List<UIVertex>();
    verts.Add(new UIVertex
    {
        position = new Vector3(-50, -50, 0),
        uv0 = new Vector2(0, 0),
        color = new Color32(255, 0, 0, 255),
    });
    verts.Add(new UIVertex
    {
        position = new Vector3(-50, 50, 0),
        uv0 = new Vector2(0, 1),
        color = new Color32(0, 255, 0, 255),
    });
    verts.Add(new UIVertex
    {
        position = new Vector3(50, 50, 0),
        uv0 = new Vector2(1, 1),
        color = new Color32(0, 0, 255, 255),
    });
}

// List of indices to be drawn
List< int> indices = new List<int>();
indices.Add(0);
indices.Add(1);
indices.Add(2);

// List received from AddUIVertexStream
List<Vector3> positions = new List<Vector3>();
List<Color32> colors = new List<Color32>();
List<Vector2> uv0s = new List<Vector2>();
List<Vector2> uv1s = new List<Vector2>();
List<Vector2> uv2s = new List<Vector2>();
List<Vector2> uv3s = new List<Vector2>();
List<Vector3> normals = new List<Vector3>();

```

```

List<Vector4> tangents = new List<Vector4>();

// Split into a list of verts
CanvasRenderer.AddUIVertexStream(verts, positions, colors, uv0s, uv1s, uv2s, uv3s, normals,
ls, tangents);

// Mesh to be created from the list received from AddUIVertexStream
Mesh mesh = new Mesh();
mesh.SetVertices(positions);
mesh.SetColors(colors);
mesh.SetUVs(0, uv0s);
mesh. SetUVs(1, uv1s);
mesh. SetUVs(2, uv2s);
mesh. SetUVs(3, uv3s);
mesh. SetNormals(normals);
mesh. SetTangents(tangents);
mesh. SetTriangles(indices, 0);
mesh.RecalculateBounds();

// Drawing can be achieved by passing the mesh to be drawn to the CanvasRenderer
canvasRenderer.SetMesh(mesh);
canvasRenderer.materialCount = 1;
canvasRenderer.SetMaterial(Canvas.GetDefaultCanvasMaterial(), 0);
canvasRenderer.SetColor(Color.white);
}

}

```

## CreateUIVertexStream

---

```

public static void CreateUIVertexStream (List<UIVertex> verts, List<Vector3> positions, List<
Color32> colors, List<Vector2> uv0S, List<Vector2> uv1S, List<Vector3> normals, List<Vecto
r4> tangents, List<int> indices);

```

Contrary to [AddUIVertexStream](#), it receives a list of vertex components ([positions](#), [colors](#), [uv0s](#), [uv1s](#), [normals](#), [tangents](#)) and a list of indices, and returns a list of vertices.

## SplitUIVertexStreams

---

```
public static void SplitUIVertexStreams (List<UIVertex> verts, List<Vector3> positions, List<Color32> colors, List<Vector2> uv0S, List<Vector2> uv1S, List<Vector3> normals, List<Vector4> tangents, List<int> indices);
```

Takes a list of triangle vertices and returns a list of vertex components and a list of indices.

The number of vertices to be passed must be divisible by 3.

## Events of CanvasRenderer

### onRequestRebuild

```
public static event OnRequestRebuild onRequestRebuild;
```

Called when the [CanvasRenderer](#) data becomes invalid or needs to be rebuilt.

This event is only valid in the Editor. For example, this event will be fired when an asset is re-imported. [GraphicRebuildTracker](#) class, which calls [MonoBehaviour.OnValidate\(\)](#) on every [Graphic](#) when the [CanvasRenderer](#) data is invalidated.

```
using UnityEngine;

// Detect when assets such as textures are re-imported and the CanvasRenderer data is invalid.
[ExecuteAlways]
public class CanvasRendererOnRequestRebuild : MonoBehaviour
{
    #if UNITY_EDITOR
        void Start()
        {
            CanvasRenderer.onRequestRebuild += OnRebuildRequested;
        }

        private void OnDestroy()
        {
            CanvasRenderer.onRequestRebuild -= OnRebuildRequested;
        }

        public static void OnRebuildRequested()
        {
            Debug.Log("CanvasRenderer data is no longer valid");
            // In this case, we call MonoBehaviour.OnValidate() on all Graphics.OnValidate()
    #endif
}
```

## Graphic related components

The [Graphic](#) class is the base class for all uGUI drawing target classes implemented in C#.

```
[DisallowMultipleComponent]
[RequireComponent(typeof(RectTransform))]
[ExecuteAlways]
public abstract class Graphic : UIBehaviour, ICanvasElement
```

Components that inherit from [Graphic](#) include [RawImage](#), [Image](#), and [Text](#) (actually, these are children of [MaskableGraphic](#), which is a child class of [Graphic](#)). On the other hand, [Button](#), [Slider](#), [ScrollRect](#), etc. are not Graphic-like components.

The main function of [Graphic](#) is to provide the Canvas with the geometry to be drawn. In addition, most components implemented as child classes of the [Graphic](#) class are implemented via [MaskableGraphic](#) subclasses, so they can be masked (i.e., clipped or masked with stencils).

The [Graphic](#) component is explained in detail in *Chapter 4 Graphic*.

## Canvas Rebuild

What is Canvas Rebuild? Canvas Rebuild is

"The process by which Canvas reconfigures the layout and geometry of the underlying UI elements to prepare them for rendering.

The Canvas rebuild consists of a Layout rebuild, a Graphic rebuild, and a Canvas batch build by the [CanvasUpdateRegistry](#) class. In addition, the Graphic rebuild consists of recreating Geometry marked as dirty and reconfiguring Material.

The following is a summary of the process that takes place in a Canvas rebuild.

- Rebuild by [CanvasUpdateRegistry](#)
  - Layout Rebuild (adjust the position)
  - Graphic Rebuild (recreate vertices and materials)
    - Recreate Geometry marked as dirty.
    - Reconfigure Material marked as dirty.
- Canvas batch build (recreate vertices for the entire canvas)

Layout rebuild and Graphic rebuild are done by the [CanvasUpdateRegistry](#) calling [Rebuild\(\)](#) on components that implement the [ICanvasElement](#) interface.

On the other hand, the batch build of Canvas is a batching process to reduce draw calls and SetPasses by grouping the vertices of UI elements in Canvas, and is done on the native side of the game engine.

Canvas rebuilding is a very demanding process, so it is important to reduce the frequency and content of the process.

## Rebuild by CanvasUpdateRegistry

Rebuilding by the [CanvasUpdateRegistry](#) is done by the [CanvasUpdateRegistry](#) calling [Rebuild\(\)](#) on components that implement the [ICanvasElement](#) interface.

The rebuild by [CanvasUpdateRegistry](#) is written in C#, so we can follow it in detail. First, let's take a look at the [ICanvasElement](#) interface that is implemented by the component that is the target of this rebuild.

### ICanvasElement

The [ICanvasElement](#) interface is the interface to be implemented by the component to be rebuilt.

The following is the definition of [ICanvasElement](#).

*Packages/com.unity.ugui/Runtime/UI/CanvasUpdateRegistry.cs*

```
public interface ICanvasElement
{
    /// <summary>
    /// Perform a specific stage rebuild on the element.
    /// </summary>
    /// <param name="executing"> Stage for rebuild</param>.
    void Rebuild(CanvasUpdate executing);

    /// <summary>
    /// Get the transform associated with ICanvasElement.
    /// </summary>
    Transform transform { get; }

    /// <summary>
    /// Callback called when this ICanvasElement has completed its Layout rebuild.
    /// </summary>
    void LayoutComplete();

    /// <summary>
    /// Callback called when this ICanvasElement has completed the Graphic rebuild.
    /// </summary>
    void GraphicUpdateComplete();
}
```

```
/// <summary>
/// Used to check if the entity on the native side of this object has been destroyed.
/// </summary>
/// <returns> Return true if this element has been destroyed</returns>.
bool IsDestroyed();
}
```

Components that implement the [ICanvasElement](#) interface include [Graphic](#), [Slider](#), [ScrollRect](#), etc. [Buttons](#) and [Dropdown](#) do not implement the [ICanvasElement](#) interface and are not eligible for Canvas. Since [Button](#) and [Dropdown](#) do not implement the [ICanvasElement](#) interface, they are not eligible for rebuilding.

[LayoutRebuilder](#), a helper class for Layout, also implements [ICanvasElement](#). [MarkLayoutForRebuild\(\)](#) to register its [RectTransform](#) so that [Rebuild\(\)](#) can be called when rebuilding Layout.

## CanvasUpdateRegistry

```
public class CanvasUpdateRegistry
```

[CanvasUpdateRegistry](#) is a class that keeps track of [ICanvasElement](#) that needs to be rebuilt, and prompts individual elements to update when [Canvas](#) fires the [willRenderCanvases](#) event.

The [CanvasUpdateRegistry](#) manages UI elements that require Layout rebuild and UI elements that require Graphic rebuild separately, as shown below.

*Packages/com.unity.ugui/Runtime/UI/Core/CanvasUpdateRegistry.cs*

```
public class CanvasUpdateRegistry
{
    ...
    ...
    // List of UI elements to rebuild in Layout
    private readonly HashSet<ICanvasElement> m_LayoutRebuildQueue = new HashSet<ICanvasElement>();

    // List of UI elements to be rebuilt for Graphic
    private readonly HashSet<ICanvasElement> m_GraphicRebuildQueue = new HashSet<ICanvasElement>();
    ...
}
```

Note

Although it is called "Graphic rebuild", Graphic rebuild is not limited to **Graphic** components, but to all components that implement **ICanvasElement**. In fact, the **InputField** component, which does not inherit from the Graphic class, is also subject to the Graphic rebuild.

The key method in the **CanvasUpdateRegistry** is **PerformUpdate()**, which is the main body of the Layout and Graphic rebuilds. **PerformUpdate()** is called at the **Canvas.WillRenderCanvases** event that occurs every frame.

Now, let's see how **CanvasUpdateRegistry.PerformUpdate** is called with **Deep Profile** enabled in Profiler.

```
PlayerLoop
  PostLateUpdate.PlayerUpdateCanvases
    UIEvents.WillRenderCanvases
      UGUI.Rendering.UpdateBatches
        Canvas.SendWillRenderCanvases
          CanvasUpdateRegistry.PerformUpdate
```

You can see that **CanvasUpdateRegistry.PerformUpdate()** is called from **PlayerLoop**'s **PostLateUpdate.PlayerUpdateCanvases** through **Canvas**. You can see that **CanvasUpdateRegistry**.

The call to **CanvasUpdateRegistry.PerformUpdate()** is realized by adding an event to **Canvas.willRenderCanvases**, as shown below.

*Packages/com.unity.ugui/Runtime/UI/Core/CanvasUpdateRegistry.cs*

```
/// <summary>
/// Class that CanvasElements will register itself with for rebuild
/// </summary>
public class CanvasUpdateRegistry
{
    ...
    protected CanvasUpdateRegistry()
    {
        Canvas.willRenderCanvases += PerformUpdate;
    }
    ...
    private void PerformUpdate()
    {
        // The actual rebuild process
    ...
}
```

Let's digress for a moment and look at the timing of the original `PostLateUpdate.PlayerUpdateCanvases` call.

## Unity's PlayerLoop

As of Unity 2018.1, there is a new mechanism called `PlayerLoop` that allows you to control the order in which the processes are performed each frame. By checking the contents of `PlayerLoop`, we can check when the internal process of uGUI is called.

The `UnityEngine.PlayerLoop` namespace contains a structure that represents each phase of the PlayerLoop. `GetDefaultPlayerLoop()` to get an array of the phases (subsystems) defined in the PlayerLoop.

```
// Output the predefined PlayerLoops to the log.
public void DumpAllPlayerLoopSubSystems()
{
    UnityEngine.LowLevel.PlayerLoopSystem[] subSystemList1 = UnityEngine.LowLevel.Player
    Loop.GetDefaultPlayerLoop().GetDefaultPlayerLoop().subSystemList;
    for (int i = 0; i < subSystemList1.Length; i++)
    {
        UnityEngine.LowLevel.PlayerLoopSystem[] subSystemList2 = subSystemList1[i].subSyste
```

```

mList;

for (int j = 0; j < subSystemList2.Length; j++)
{
    Debug.LogFormat("{0} {1} {2} {3}", i, j, subSystemList1[i].ToString(), subSystemList2
[j].ToString());
}
}
}

```

*Console output*

```

0 0 TimeUpdate WaitForLastPresentationAndUpdateTime
1 0 Initialization UpdateCameraMotionVectors
1 1 Initialization DirectorSampleTime
1 2 Initialization AsyncUploadTimeSlicedUpdate
1 3 Initialization SynchronizeInputs
1 4 Initialization SynchronizeState
1 5 Initialization XREarlyUpdate
2 0 EarlyUpdate PollPlayerConnection
2 1 EarlyUpdate ProfilerStartFrame
2 2 EarlyUpdate GpuTimestamp
2 3 EarlyUpdate AnalyticsCoreStatsUpdate
2 4 EarlyUpdate UnityWebRequestUpdate
2 5 EarlyUpdate ExecuteMainThreadJobs
2 6 EarlyUpdate ProcessMouseInWindow
2 7 EarlyUpdate ClearIntermediateRenderers
2 8 EarlyUpdate ClearLines
2 9 EarlyUpdate PresentBeforeUpdate
2 10 EarlyUpdate ResetFrameStatsAfterPresent
2 11 EarlyUpdate UpdateAsyncReadbackManager
2 12 EarlyUpdate UpdateStreamingManager
2 13 EarlyUpdate UpdateTextureStreamingManager
2 14 EarlyUpdate UpdatePreloading
2 15 EarlyUpdate RendererNotifyInvisible
2 16 EarlyUpdate PlayerCleanupCachedData
2 17 EarlyUpdate UpdateMainGameViewRect
2 18 EarlyUpdate UpdateCanvasRectTransform
2 19 EarlyUpdate XRUpdate
2 20 EarlyUpdate UpdateInputManager

```

2 21 EarlyUpdate ProcessRemoteInput
2 22 EarlyUpdate ScriptRunDelayedStartupFrame
2 23 EarlyUpdate UpdateKinect
2 24 EarlyUpdate DeliverIosPlatformEvents
2 25 EarlyUpdate ARCoreUpdate
2 26 EarlyUpdate DispatchEventQueueEvents
2 27 EarlyUpdate PhysicResetInterpolatedTransformPosition
2 28 EarlyUpdate SpriteAtlasManagerUpdate
2 29 EarlyUpdate PerformanceAnalyticsUpdate
3 0 FixedUpdate ClearLines
3 1 FixedUpdate NewInputFixedUpdate
3 2 FixedUpdate DirectorFixedSampleTime
3 3 FixedUpdate AudioFixedUpdate
3 4 FixedUpdate ScriptRunBehaviourFixedUpdate
3 5 FixedUpdate DirectorFixedUpdate
3 6 FixedUpdate LegacyFixedAnimationUpdate
3 7 FixedUpdate XRFixedUpdate
3 8 FixedUpdate PhysicsFixedUpdate
3 9 FixedUpdate Physics2DFixedUpdate
3 10 FixedUpdate PhysicsClothFixedUpdate
3 11 FixedUpdate DetectorFixedUpdatePostPhysics
3 12 FixedUpdate ScriptRunDelayedFixedFrameRate
4 0 PreUpdate PhysicsUpdate
4 1 PreUpdate Physics2DUpdate
4 2 PreUpdate CheckTexFieldInput
4 3 PreUpdate IMGUISendQueuedEvents
4 4 PreUpdate NewInputUpdate
4 5 PreUpdate SendMouseEvents
4 6 PreUpdate AIUpdate
4 7 PreUpdate WindUpdate
4 8 PreUpdate UpdateVideo
5 0 Update ScriptRunBehaviourUpdate
5 1 Update ScriptRunDelayedDynamicFrameRate
5 2 Update ScriptRunDelayedTasks
5 3 Update DirectorUpdate
6 0 PreLateUpdate AIUpdatePostScript
6 1 PreLateUpdate DirectorUpdateAnimationBegin
6 2 PreLateUpdate LegacyAnimationUpdate
6 3 PreLateUpdate DirectorUpdateAnimationEnd
6 4 PreLateUpdate DirectorDeferredEvaluate
6 5 PreLateUpdate UIElementsUpdatePanels
6 6 PreLateUpdate EndGraphicsJobsAfterScriptUpdate

6 7 PreLateUpdate ConstraintManagerUpdate  
6 8 PreLateUpdate ParticleSystemBeginUpdateAll  
6 9 PreLateUpdate Physics2DLateUpdate  
6 10 PreLateUpdate ScriptRunBehaviourLateUpdate  
7 0 PostLateUpdate PlayerSendFrameStarted  
7 1 PostLateUpdate DirectorLateUpdate  
7 2 PostLateUpdate ScriptRunDelayedDynamicFrameRate  
7 3 PostLateUpdate PhysicsSkinnedClothBeginUpdate  
7 4 PostLateUpdate UpdateRectTransform  
7 5 PostLateUpdate PlayerUpdateCanvases  
7 6 PostLateUpdate UpdateAudio  
7 7 PostLateUpdate VFXUpdate  
7 8 PostLateUpdate ArticleSystemEndUpdateAll  
7 9 PostLateUpdate EndGraphicsJobsAfterScriptLateUpdate  
7 10 PostLateUpdate UpdateCustomRenderTextures  
7 11 PostLateUpdate UpdateAllRenderers  
7 12 PostLateUpdate UpdateLightProbeProxyVolumes  
7 13 PostLateUpdate EnlightenRuntimeUpdate  
7 14 PostLateUpdate UpdateAllSkinnedMeshes  
7 15 PostLateUpdate ProcessWebSendMessages  
7 16 PostLateUpdate SortingGroupsUpdate  
7 17 PostLateUpdate UpdateVideoTextures  
7 18 PostLateUpdate UpdateVideo  
7 19 PostLateUpdate DirectorRenderImage  
7 20 PostLateUpdate PlayerEmitCanvasGeometry  
7 21 PostLateUpdate PhysicsSkinnedClothFinishUpdate  
7 22 PostLateUpdate FinishFrameRendering  
7 23 PostLateUpdate BatchModeUpdate  
7 24 PostLateUpdate PlayerSendFrameComplete  
7 25 PostLateUpdate UpdateCaptureScreenshot  
7 26 PostLateUpdate PresentAfterDraw  
7 27 PostLateUpdate ClearImmediateRenderers  
7 28 PostLateUpdate PlayerSendFramePostPresent  
7 29 PostLateUpdate UpdateResolution  
7 30 PostLateUpdate InputEndFrame  
7 31 PostLateUpdate TriggerEndOfFrameCallbacks  
7 32 PostLateUpdate GUIClearEvents  
7 33 PostLateUpdate ShaderHandleErrors  
7 34 PostLateUpdate ResetInputAxis  
7 35 PostLateUpdate ThreadedLoadingDebug  
7 36 PostLateUpdate ProfilerSynchronizeStats  
7 37 PostLateUpdate MemoryFrameMaintenance

7 38 PostLateUpdate ExecuteGameCenterCallbacks

7 39 PostLateUpdate ProfilerEndFrame

The above phases are executed every frame in order from the top.

The [PostLateUpdate.PlayerUpdateCanvases](#) is as follows.

7 5 PostLateUpdate PlayerUpdateCanvases

This means that the Canvas rebuild is performed in a much later phase of the frame.

[CanvasUpdateRegistry.PerformUpdate\(\)](#)

[CanvasUpdateRegistry.PerformUpdate\(\)](#) is the entity for Graphic and Layout rebuild. This method is always called every frame, regardless of whether there is a UI element to be rebuilt or not. This method performs three steps.

1. Layout rebuild: Perform Layout pre-processing/actual processing/post-processing on elements in the list of components whose Layout is dirty.
2. Culling for Clipping: Call [ClipperRegistry.Cull\(\)](#) for all [Clipping](#) components such as [RectMask2D](#) to perform culling.
3. Graphic rebuild: Pre-render processing for elements in the list of components whose geometry is dirty.

There are multiple stages to a [Canvas](#) rebuild. This is defined as the [CanvasUpdate](#) enumeration type.

*Packages/com.unity.ugui/Runtime/UI/Core/CanvasUpdateRegistry.cs*

```
namespace UnityEngine.UI
{
    public enum CanvasUpdate
    {
        /// <summary>
        /// Called before the layout process.
        /// </summary>
        Prelayout = 0,
        /// <summary>
```

```

/// Called for the layout process
/// </summary>
Layout = 1,

/// <summary>
/// Called after the layout process
/// </summary>
PostLayout = 2,

/// <summary>
/// Called before rendering
/// </summary>
PreRender = 3,

/// <summary>
/// Called after PreRender, before rendering
/// </summary>
LatePreRender = 4,

/// <summary>
/// Maximum value of enum
/// </summary>
MaxUpdateValue = 5
...

```

`CanvasUpdate` is passed as an argument to `ICanvasElement.Rebuild()` to instruct each UI element to perform each stage of the rebuild.

*Packages/com.unity.ugui/Runtime/UI/Core/CanvasUpdateRegistry.cs*

```

public interface ICanvasElement
{
    /// <summary>
    /// Rebuild the element for each stage
    /// </summary>
    /// <param name="executing"> The current CanvasUpdate stage we're trying to rebuild</param>
    void Rebuild(CanvasUpdate executing);
}

```

Layout rebuild is done in three stages: [PreLayout](#), [Layout](#), and [PostLayout](#). On the other hand, Graphic rebuild is done in two stages, [PreRender](#) and [LatePreRender](#).

[LayoutComplete\(\)](#) is called when the Layout rebuild is complete, and [ICanvasElement.GraphicUpdateComplete\(\)](#) is called when the Graphic rebuild is complete.

## Layout Rebuild

Layout rebuild is a process that adjusts all [RectTransforms](#) in the hierarchy that contains the target UI element, and since most of the process of layout rebuild is Auto Layout related, the load is not so high if you are not using Auto Layout. Most of the Layout rebuild process is related to Auto Layout, so if you are not using Auto Layout, the load is not so high, but if you have a large number of UI elements to rebuild, the load cannot be ignored. Layout rebuild is performed in three stages: [PreLayout](#), [Layout](#), and [PostLayout](#).

Layout rebuilding occurs when a UI element that implements [ICanvasElement](#) decides that its own layout is dirty. The conditions for "dirty layout" differ for each component. For example, in the case of Graphic-related components, the conditions for their own layout to be dirty are as follows.

- [RectTransform](#) was resized. ([OnRectTransformDimensionsChange\(\)](#) was called)
- The component was enabled or disabled. ([OnEnable\(\)](#) or [OnDisable\(\)](#) was called)
- The parent has changed. ([OnBeforeTransformParentChanged\(\)](#) was called)
- The property was modified by [animation](#) or [timeline](#). ([OnDidApplyAnimationProperties\(\)](#) was called)
- The [Sprite](#) of the [Image](#) has been changed.
- The [text](#) in [Text](#) has been changed.
- [Text](#) font, font size, and font texture have been changed.
- Various display settings for [Text](#) have been changed. ([alignment](#), [lineSpacing](#), [WrapMode](#), [fontStyle](#), etc.)

A UI element whose Layout is dirty will pass itself to [RegistarCanvasElementForLayoutRebuild\(\)](#) in the [CanvasUpdateRegistry](#).

```
public static void RegisterCanvasElementForLayoutRebuild(ICanvasElement element);
```

This will add itself to the list of dirty layouts managed by [CanvasUpdateRegistry](#).

In order to recalculate the proper position and size of the components, the layout needs to be adjusted according to the order of the hierarchy. Since the parent component may change the position and size of its children and grandchildren components, the position and size must be

calculated first. For this reason, components whose Layout is marked as dirty are sorted by depth in the hierarchy. UI elements that are higher in the hierarchy are moved to the end of the list.

`Rebuild()` is called on the elements in this sorted list with the arguments `CanvasUpdate.Prelayout`, `CanvasUpdate`. is performed.

`LayoutComplete()` is called by the `CanvasUpdateRegistry` for each UI element when the layout rebuild is complete.

## Graphic Rebuild

---

Graphic rebuild is the process of changing the geometry ( $\approx$  vertices) information and material of the UI element itself. The only components that are processed in the Graphic rebuild are `Graphic` and `InputField`. In the following, we will focus on the `Graphic` component.

For the `Graphic` component, a Graphic rebuild will occur when it determines that its geometry or material is dirty.

The conditions under which the geometry becomes dirty are as follows.

- `RectTransform` was resized. (`OnRectTransformDimensionsChange()` was called)
- The `color` has been changed.
- The parent has changed. (`OnBeforeTransformParentChanged()` was called)
- The component has been enabled. (`OnEnable()` was called.)
- The property was modified by `Animation` or `Timeline`. (`OnDidApplyAnimationProperties()` was called)
- The `texture` of `RawImage` has been changed.
- The `uvRect` of `RawImage` has been changed.
- `Image's sprite` or `overrideSprite` has been changed.
- The various display settings for `images` have been changed. (`type`, `preserveAspect`, `fillMethod`, `useSpriteMesh`, etc.)
- The `text` in `Text` has been changed.
- `Text` font, font size, and font texture have been changed.
- Various display settings for `Text` have been changed. (`alignment`, `lineSpacing`, `WrapMode`, `fontStyle`, etc.)

The conditions under which a material becomes dirty are as follows.

- The `material` has been changed.
- The parent has changed. (`OnBeforeTransformParentChanged()` was called)
- The component has been enabled. (`OnEnable()` was called.)

- The property was modified by [animation](#) or [timeline](#). ([OnDidApplyAnimationProperties\(\)](#) was called)
- The [sprite](#) of the [Image](#) has been [changed](#).
- [Text](#) font and font texture have been changed.

When [Graphic.Rebuild\(\)](#) is called in the [PreRender](#) stage of [CanvasUpdateRegistry.PerformUpdate\(\)](#), the following two processes will take place Rebuild() is called in the PreRender stage of CanvasUpdateRegistry.

1. If the geometry is dirty, rebuild the mesh; [OnPopulateMesh\(\)](#) is called to recreate the vertices, the MeshEffect is applied, and the mesh of this UI element is passed to the [CanvasRenderer](#) in [CanvasRenderer.SetMesh\(\)](#).
2. If the material is dirty, update the [CanvasRenderer](#)'s material; the [CanvasRenderer](#)'s [materialCount](#) will be set to [1](#), and [CanvasRenderer.\(\)](#) will set the material and texture.

## Canvas batch build

So far, we have looked at Layout builds and Graphic rebuilds using [CanvasUpdateRegistry](#). Now we will discuss another mechanism of Canvas rebuild, namely Canvas batch build.

In order to display the UI, we have to create the geometry corresponding to the UI components that will be displayed on the screen. This process involves integrating as much geometry as possible into a single mesh to minimize draw calls.

If any of the drawable UI elements in the Canvas have changed, Canvas will have to re-run the batch build process. This process will re-analyze all the drawable UI elements in the Canvas, regardless of whether they have changed or not. By changes here, we mean any changes that affect the appearance of the UI object, not necessarily Layout or Graphic rebuilds by [CanvasUpdateRegistry](#).

For example, simply changing the [anchoredPosition](#) of [RectTransform](#) of a UI element that is not subject to Auto Layout will not cause a Layout rebuild or a Graphic rebuild, but it will change the position of the vertices to be displayed, so it is necessary to perform a Canvas batch build.

However, the position of the vertices to be displayed will change, so it is necessary to perform a batch build of Canvas.

The batch rebuild of Canvas generates rendering commands that combine the meshes of UI elements and send them to the graphics pipeline. The results of this process are cached and reused until the Canvas becomes dirty. Note that the [CanvasRenderer](#), which is included in the sub-Canvas, is not subject to this Canvas batch rebuild.

To do this, we need to sort the meshes by depth and sort by material.

## Canvas Rebuild Summary

The following is a summary of the most common operations performed on UI elements and whether or not Layout rebuild / Graphic rebuild / Canvas batch build can occur.

	Layout Rebuild	Graphic Rebuild (Geometry)	Graphic Rebuild (Material)	Canvas Batch Build
Change position/scale (without Auto Layout)	No	No.	No	Yes
Change position/scale (Auto Layout available)	Yes	No	No	Yes
Resize (without Auto Layout)	No	Yes	No	Yes
Resize (Auto Layout available)	No	Yes	No	Yes
Enable/Disable component switching	Yes	Yes	Yes	Yes
Change text	No	No	Partially	Yes
Change the color	No	No	Yes	Yes
Animation and Timeline	Yes	Yes	Yes	Yes

“Yes” indicates an occurrence.

As you can see, activating/deactivating components (and activating/deactivating [GameObjects](#)) and changing properties via [Animations](#) and [Timelines](#) are particularly demanding on Canvas rebuilds. Therefore, these should be avoided. Specifically, the following workarounds can be taken.

1. To show/hide the UI, change the `localScale` to `Vector3.one` or `Vector3.zero` instead of using the [GameObject's SetActive\(\)](#). The details are explained in *Chapter 3 Rendering, Switching UI Elements on and off*.
2. If you want to change only the visual size of the UI, change the `localScale` instead of the `RectTransform's sizeDelta`.
3. Don't use [Animations](#) or [Timelines](#) to animate your UI; it's better to use a tween library such as [DOTween](#) and edit only the properties you need from the script.
4. In some cases, the desired layout can be achieved by using `RectTransform` anchors instead of Layout Groups. It is also possible to use Auto Layout only for placement in the Editor to obtain placement information, and not use Auto Layout at runtime.

While these steps are the minimum to improve performance, there are still many things that need to be improved. In the following sections, we will discuss techniques to further reduce the load of Canvas rebuilds.

## Reduce the number of UI elements

If you have a large number of UI elements to draw in Canvas, the batch build calculation itself can become very burdensome. This is because the cost of sorting and calculating the UI elements increases more rapidly than linearly with the number of UI elements. So, it is simple but effective to reduce the number of UI elements.

## Splitting into sub-Canvas

By appropriately dividing a Canvas into sub-Canvas, performance can be improved. A sub-Canvas refers to a child Canvas nested within a Canvas component.

By splitting a portion of a Canvas into sub-Canvas, it is possible to reduce the scope of influence of Layout rebuilds and batch builds. Even if a Layout rebuild or batch build becomes necessary on the child Canvas, it will not affect the parent Canvas. Conversely, if Layout rebuild or batch build is required on the parent Canvas, it will not affect the child Canvas. However, as an exception, changes made to the parent Canvas may cause the child Canvas to be resized.

Splitting a Canvas into multiple sub-Canvas is a common optimization technique. This is most often used when certain parts of the UI need to be at a different depth than other parts. In general, it is more convenient to create a sub-Canvas than to create a completely separate Canvas. This is because a sub-Canvas inherits its display settings from its parent Canvas.

However, if the Canvas is divided into too many sub-Canvas, the rendering batch will be interrupted and draw calls will increase, resulting in worse performance. The Canvas must be divided in consideration of the balance between minimizing the impact of rebuilds and reducing draw calls by batching.

A common way to divide the Canvas is to prepare two Canvas, one for static UI elements such as background images, and one for dynamic UI elements that change frequently. If you have a large number of dynamic elements, you may want to further divide the dynamic elements into those that change regularly (progress bars, timers, animated elements) and those that change occasionally. In reality, three or so Canvas elements should be sufficient.

## Using the SpriteRenderer instead of the CanvasRenderer

As we have seen, if drawing elements that animate frequently are drawn using [CanvasRenderer](#), performance will deteriorate due to Canvas rebuilding. Therefore, we have to choose another method, and [SpriteRenderer](#) is one of the options.

If you use [SpriteRenderer](#) instead of [CanvasRenderer](#), there are a few things to keep in mind

- Unlike drawing with the [CanvasRenderer](#), drawing with the [SpriteRenderer](#) does not automatically batch. Note that different materials will break up the batch.
- The position and size need to be converted appropriately to match the [Camera](#).
- Set the *Projection* of the [Camera](#) that displays the [SpriteRenderer](#) to *Orthographic*.

When using a [SpriteRenderer](#) to replace an image that was originally drawn as an [Image](#), a small calculation is required to make the [SpriteRenderer](#)'s position and size match the original [Image](#). The method of calculation depends on the [Canvas](#)'s [renderMode](#) and [CanvasScaler](#)'s [uiScaleMode](#) settings. The calculation method depends on the [Canvas](#) [renderMode](#) and [CanvasScaler](#) [uiScaleMode](#) settings, etc. Since there are many patterns and it is complicated, I have prepared the following methods for calculation.

```
using UnityEngine;
using UnityEngine.UI;

public class CanvasToSpriteUtils : MonoBehaviour
{
    // Create a Sprite under targetSpriteCamera that matches the location of srcImage
    public static Sprite CreateSpriteForImage(Image srcImage, Camera spriteCamera)
    {
        var go = new GameObject(srcImage.name + "_sprite");
        go.transform.SetParent(spriteCamera.transform, false);

        var spriteRenderer = go.AddComponent<SpriteRenderer>();
        var targetRectTransform = srcImage.transform as RectTransform;

        var sprite = srcImage.sprite;
        if (sprite == null)
        {
            // Create() is very heavy, so be careful.
            sprite = Sprite.Create((Texture2D)srcImage.mainTexture, new Rect(0, 0, 1, 1), new Vector2(0.5f, 0.5f));
        }

        spriteRenderer.sprite = sprite;
        spriteRenderer.sortingOrder = targetRectTransform.sortingOrder;
        spriteRenderer.transform.position = targetRectTransform.position;
        spriteRenderer.transform.rotation = targetRectTransform.rotation;
        spriteRenderer.transform.localScale = targetRectTransform.localScale;
    }
}
```

```

    }

    spriteRenderer.sprite = sprite;
    spriteRenderer.color = srcImage.color;

    SetSpriteTransformByRectTransform(spriteRenderer, spriteCamera, targetRectTransform);

    return sprite;
}

// Set the position/size of the spriteRenderer in the spriteCamera to the same position/size as the rectTransform
public static void SetSpriteTransformByRectTransform(SpriteRenderer spriteRenderer, Camera spriteCamera, RectTransform rectTransform)
{
    var canvas = rectTransform.GetComponentInParent<Canvas>();
    canvas.worldCamera.orthographic = true;

    switch (canvas.renderMode)
    {
        case RenderMode.ScreenSpaceOverlay:
            SetSpriteTransformByRectTransformScreenSpaceOverlay(spriteRenderer, spriteCamera, rectTransform, canvas);
            break;

        case RenderMode.ScreenSpaceCamera:
            SetSpriteTransformByRectTransformScreenSpaceCamera(spriteRenderer, spriteCamera, rectTransform, canvas);
            break;

        case RenderMode.WorldSpace:
            SetSpriteTransformByRectTransformWorldSpace(spriteRenderer, spriteCamera, rectTransform, canvas);
            break;
    }
}

public static void SetSpriteTransformByRectTransformScreenSpaceOverlay(SpriteRenderer spriteRenderer, Camera spriteCamera, RectTransform rectTransform, Canvas canvas)
{
    Sprite sprite = spriteRenderer.sprite;
    float positionRate = 1.0f;
}

```

```

float scaleRate = 1.0f;
Vector3 canvasScale = canvas.GetComponent<RectTransform>().localScale;
var canvasScaler = canvas.GetComponent<CanvasScaler>();
float refX = canvasScaler.referenceResolution.x;
float refY = canvasScaler.referenceResolution.y;
float canvasW = canvas.pixelRect.width;
float canvasH = canvas.pixelRect.height;
float spritePPU = sprite.pixelsPerUnit;
float spriteW = sprite.rect.width;
float spriteH = sprite.rect.height;
float srcX = rectTransform.position.x;
float srcY = rectTransform.position.y;
float baseX = (srcX / canvasW - 0.5f) * canvasW;
float baseY = (srcY / canvasH - 0.5f) * canvasH;
float overlayBaseRate = 2.0f / canvasH * spriteCamera.orthographicSize;
const float spriteOffsetZ = 1;

switch (canvasScaler.uiScaleMode)
{
    case CanvasScaler.ScaleMode.ConstantPixelSize:
        positionRate = overlayBaseRate;
        scaleRate = positionRate * (spritePPU / spriteW);
        break;

    case CanvasScaler.ScaleMode.ConstantPhysicalSize:
        float unitScaleRate = 1.0f;
        float dpi = Screen.dpi;
        if (dpi == 0)
        {
            dpi = canvasScaler.fallbackScreenDPI;
        }

        switch (canvasScaler.physicalUnit)
        {
            case CanvasScaler.Unit.Points:
                unitScaleRate = 72.0f / dpi;
                break;

            case CanvasScaler.Unit.Inches:
                unitScaleRate = 1.0f / dpi;
                break;
        }
}

```

```

case CanvasScaler.Unit.Centimeters:
    unitScaleRate = 1.0f / dpi * 2.54f;
    break;

case CanvasScaler.Unit.Millimeters:
    unitScaleRate = 1.0f / dpi * 2.54f * 10.0f;
    break;

case CanvasScaler.Unit.Picas:
    unitScaleRate = 1.0f / dpi * 6.0f;
    break;
}

positionRate = overlayBaseRate * canvasScale.x * unitScaleRate;
scaleRate = positionRate * (spritePPU / spriteW) / unitScaleRate;
break;

case CanvasScaler.ScaleMode.ScaleWithScreenSize:
    switch (canvasScaler.screenMatchMode)
    {
        case CanvasScaler.ScreenMatchMode.MatchWidthOrHeight:
            float lerpValue = Mathf.Pow(2, canvasScaler.matchWidthOrHeight) - 1;
            positionRate = Mathf.Lerp(overlayBaseRate * canvasScale.x * refX / canvasW,
            overlayBaseRate * canvasScale.y * refY / canvasH, lerpValue);
            scaleRate = Mathf.Lerp(positionRate * (spritePPU / spriteW) * canvasW / refX,
                positionRate * (spritePPU / spriteH) * canvasH / refY,
                canvasScaler.matchWidthOrHeight);
            break;

        case CanvasScaler.ScreenMatchMode.Expand:
            if (canvasW / canvasH < refX / refY)
            {
                positionRate = overlayBaseRate * canvasScale.x * refX / canvasW;
                scaleRate = positionRate * (spritePPU / spriteW) * canvasW / refX;
            }
            else
            {
                positionRate = overlayBaseRate * canvasScale.y * refY / canvasH;
                scaleRate = positionRate * (spritePPU / spriteH) * canvasH / refY;
            }
            break;
    }
}

```

```

case CanvasScaler.ScreenMatchMode.Shrink:
    if (canvasW / canvasH < refX / refY)
    {
        positionRate = overlayBaseRate * canvasScale.y * refY / canvasH;
        scaleRate = positionRate * (spritePPU / spriteH) * canvasH / refY;
    }
    else
    {
        positionRate = overlayBaseRate * canvasScale.x * refX / canvasW;
        scaleRate = positionRate * (spritePPU / spriteW) * canvasW / refX;
    }
    break;
}
break;
}

spriteRenderer.transform.localPosition = new Vector3(baseX * positionRate / spriteCamera.
transform.localScale.x,
    baseY * positionRate / spriteCamera.transform.localScale.y, spriteOffsetZ);
spriteRenderer.transform.localRotation = rectTransform.transform.localRotation;
spriteRenderer.transform.localScale = new Vector3(rectTransform.rect.width * scaleRate / s
priteCamera.transform.localScale.x,
    rectTransform.rect.height * scaleRate / spriteCamera.transform.localScale.y, 1);
}

public static void SetSpriteTransformByRectTransformScreenSpaceCamera(SpriteRenderer s
priteRenderer, Camera spriteCamera, RectTransform rectTransform, Canvas canvas)
{
    Sprite sprite = spriteRenderer.sprite;
    float positionRate = 1.0f;
    float scaleRate = 1.0f;
    Vector3 canvasScale = canvas.GetComponent<RectTransform>().localScale;
    var canvasScaler = canvas.GetComponent<CanvasScaler>();
    float refX = canvasScaler.referenceResolution.x;
    float refY = canvasScaler.referenceResolution.y;
    float canvasW = canvas.pixelRect.width;
    float canvasH = canvas.pixelRect.height;
    float spritePPU = sprite.pixelsPerUnit;
    float spriteW = sprite.rect.width;
    float spriteH = sprite.rect.height;
    Vector3 srcScreenPos = RectTransformUtility.WorldToScreenPoint(canvas.worldCamera, r
ectTransform.position);
}

```

```

float srcX = srcScreenPos.x;
float srcY = srcScreenPos.y;
float baseX = (srcX / canvasW - 0.5f) * canvasW;
float baseY = (srcY / canvasH - 0.5f) * canvasH;
const float spriteOffsetZ = 1;

switch (canvasScaler.uiScaleMode)
{
    case CanvasScaler.ScaleMode.ConstantPixelSize:
        positionRate = canvasScale. x;
        scaleRate = positionRate * (spritePPU / spriteW);
        break;

    case CanvasScaler.ScaleMode.ConstantPhysicalSize:
        float unitScaleRate = 1.0f;
        float dpi = Screen.dpi;
        if (dpi == 0)
        {
            dpi = canvasScaler.fallbackScreenDPI;
        }

        switch (canvasScaler.physicalUnit)
        {
            case CanvasScaler.Unit.Points:
                unitScaleRate = 72.0f / dpi;
                break;

            case CanvasScaler.Unit.Inches:
                unitScaleRate = 1.0f / dpi;
                break;

            case CanvasScaler.Unit.Centimeters:
                unitScaleRate = 1.0f / dpi * 2.54f;
                break;

            case CanvasScaler.Unit.Millimeters:
                unitScaleRate = 1.0f / dpi * 2.54f * 10.0f;
                break;

            case CanvasScaler.Unit.Picas:
                unitScaleRate = 1.0f / dpi * 6.0f;
                break;
        }
}

```

```

        }

positionRate = canvasScale.x * unitScaleRate;
scaleRate = positionRate * (spritePPU / spriteW) / unitScaleRate;
break;

case CanvasScaler.ScaleMode.ScaleWithScreenSize:
    switch (canvasScaler.screenMatchMode)
    {
        case CanvasScaler.ScreenMatchMode.MatchWidthOrHeight:
            float lerpValue = Mathf.Pow(2, canvasScaler.matchWidthOrHeight) - 1;
            positionRate = Mathf.Lerp(canvasScale.x * refX / canvasW, canvasScale.y * refY
            / canvasH, lerpValue);
            scaleRate = Mathf.Lerp(positionRate * (spritePPU / spriteW) * canvasW / refX,
                positionRate * (spritePPU / spriteH) * canvasH / refY,
                canvasScaler.matchWidthOrHeight);
            break;

        case CanvasScaler.ScreenMatchMode.Expand:
            if (canvasW / canvasH < refX / refY)
            {
                positionRate = canvasScale.x * refX / canvasW;
                scaleRate = positionRate * (spritePPU / spriteW) * canvasW / refX;
            }
            else
            {
                positionRate = canvasScale.y * refY / canvasH;
                scaleRate = positionRate * (spritePPU / spriteH) * canvasH / refY;
            }
            break;

        case CanvasScaler.ScreenMatchMode.Shrink:
            if (canvasW / canvasH < refX / refY)
            {
                positionRate = canvasScale.y * refY / canvasH;
                scaleRate = positionRate * (spritePPU / spriteH) * canvasH / refY;
            }
            else
            {
                positionRate = canvasScale.x * refX / canvasW;
                scaleRate = positionRate * (spritePPU / spriteW) * canvasW / refX;
            }
    }
}

```

```

        break;
    }
    break;
}

spriteRenderer.transform.localPosition = new Vector3(baseX * positionRate / spriteCamera.
transform.localScale.x,
baseY * positionRate / spriteCamera.transform.localScale.y, spriteOffsetZ);
spriteRenderer.transform.localRotation = rectTransform.transform.localRotation;
spriteRenderer.transform.localScale = new Vector3(rectTransform.rect.width * scaleRate /
spriteCamera.transform.localScale.x,
rectTransform.rect.height * scaleRate / spriteCamera.transform.localScale.y, 1);
}

public static void SetSpriteTransformByRectTransformWorldSpace(SpriteRenderer spriteRen
derer, Camera spriteCamera, RectTransform rectTransform, Canvas canvas = null)
{
    Sprite sprite = spriteRenderer.sprite;
    float positionRate = 1.0f;
    float scaleRate = 1.0f;
    Vector3 canvasScale = canvas.GetComponent<RectTransform>().localScale;
    var canvasScaler = canvas.GetComponent<CanvasScaler>();
    float refX = canvasScaler.referenceResolution.x;
    float refY = canvasScaler.referenceResolution.y;
    float canvasW = canvas.pixelRect.width;
    float canvasH = canvas.pixelRect.height;
    float spritePPU = sprite.pixelsPerUnit;
    float spriteW = sprite.rect.width;
    float spriteH = sprite.rect.height;
    Vector3 srcScreenPos = RectTransformUtility.WorldToScreenPoint(canvas.worldCamera, r
ectTransform.position);
    float srcX = srcScreenPos.x;
    float srcY = srcScreenPos.y;
    float baseX = (srcX / canvasW - 0.5f) * canvasW;
    float baseY = (srcY / canvasH - 0.5f) * canvasH;
    const float spriteOffsetZ = 1;

    switch (canvasScaler.uiScaleMode)
    {
        case CanvasScaler.ScaleMode.ConstantPixelSize:
            positionRate = canvasScale.x;
            scaleRate = positionRate * (spritePPU / spriteW);
    }
}

```

```

break;

case CanvasScaler.ScaleMode.ConstantPhysicalSize:
    float unitScaleRate = 1.0f;
    float dpi = Screen.dpi;
    if (dpi == 0)
    {
        dpi = canvasScaler.fallbackScreenDPI;
    }

    switch (canvasScaler.physicalUnit)
    {
        case CanvasScaler.Unit.Points:
            unitScaleRate = 72.0f / dpi;
            break;

        case CanvasScaler.Unit.Inches:
            unitScaleRate = 1.0f / dpi;
            break;

        case CanvasScaler.Unit.Centimeters:
            unitScaleRate = 1.0f / dpi * 2.54f;
            break;

        case CanvasScaler.Unit.Millimeters:
            unitScaleRate = 1.0f / dpi * 2.54f * 10.0f;
            break;

        case CanvasScaler.Unit.Picas:
            unitScaleRate = 1.0f / dpi * 6.0f;
            break;
    }

    positionRate = canvasScale. x * unitScaleRate;
    scaleRate = positionRate * (spritePPU / spriteW) / unitScaleRate;
    break;

case CanvasScaler. ScaleMode.ScaleWithScreenSize:
    switch (canvasScaler.screenMatchMode)
    {
        case CanvasScaler.ScreenMatchMode.MatchWidthOrHeight:
            float lerpValue = Mathf.Pow(2, canvasScaler.matchWidthOrHeight) - 1;

```

```

positionRate = Mathf.Lerp(canvasScale.x * refX / canvasW, canvasScale.y * refY
/ canvasH, lerpValue);
scaleRate = Mathf.Lerp(positionRate * (spritePPU / spriteW) * canvasW / refX,
positionRate * (spritePPU / spriteH) * canvasH / refY,
canvasScaler.matchWidthOrHeight);
break;

case CanvasScaler.ScreenMatchMode.Expand:
if (canvasW / canvasH < refX / refY)
{
    positionRate = canvasScale.x * refX / canvasW;
    scaleRate = positionRate * (spritePPU / spriteW) * canvasW / refX;
}
else
{
    positionRate = canvasScale.y * refY / canvasH;
    scaleRate = positionRate * (spritePPU / spriteH) * canvasH / refY;
}
break;

case CanvasScaler.ScreenMatchMode.Shrink:
if (canvasW / canvasH < refX / refY)
{
    positionRate = canvasScale.y * refY / canvasH;
    scaleRate = positionRate * (spritePPU / spriteH) * canvasH / refY;
}
else
{
    positionRate = canvasScale.x * refX / canvasW;
    scaleRate = positionRate * (spritePPU / spriteW) * canvasW / refX;
}
break;
}
break;
}

spriteRenderer.transform.localPosition = new Vector3(baseX * positionRate / spriteCamera
.transform.localScale.x,
baseY * positionRate / spriteCamera.transform.localScale.y, spriteOffsetZ);
spriteRenderer.transform.localRotation = rectTransform.transform.localRotation;
spriteRenderer.transform.localScale = new Vector3(rectTransform.rect.width * scaleRate / s
priteCamera.transform.localScale.x,

```

```
    rectTransform.rect.height * scaleRate / spriteCamera.transform.localScale.y, 1);  
}  
}
```

It is worth considering the method of creating the UI on Canvas when designing the UI, and replacing only the parts that need to be animated frequently with [SpriteRenderer](#) rendering.

## Chapter 3 Rendering

### Drawing Order in Unity

Objects in Unity are rendered first if they meet the following conditions in order from the top.

- Camera with small **Depth**
  - SpriteRenderer or Canvas with a small **Sorting Layer**
    - Renderer or Canvas with small **Order in Layer**
    - Material with small **RenderQueue**
      - For opaque RenderQueue, a mesh close to the Camera
      - For transparent RenderQueue, a mesh far from the Camera
- Canvas with **Render Mode** set to **Screen Space - Overlay**
  - Canvas with small **Sorting Order**

#### Camera Depth

If there are multiple cameras in the scene, the camera with the lowest **Depth** will be drawn first, and the **Depth** can be accessed from the script as the `depth` property.

##### Note

Although the name is **Depth**, it has nothing to do with depth buffers or depth textures.

## Sorting Layer

**Sorting Layer** is a [struct](#) that determines the drawing order of the [SpriteRenderer](#) and [Canvas](#) within the same camera. [Sorting Layer](#) is defined in [UnityEngine](#) namespace.

```
namespace UnityEngine
{
    public struct SortingLayer
    {
        private int m_Id;

        public int id => m_Id;
        public string name => IDToName(m_Id);
        public int value => GetLayerValueFromID(m_Id);
        public static SortingLayer[] layers
        {
            ...
        }

        private static extern int[] GetSortingLayerIDsInternal();
        public static extern int GetLayerValueFromID(int id);
        public static extern int GetLayerValueFromName(string name);
        public static extern int NameToID(string name);
        public static extern string IDToName(int id);
        public static extern bool IsValid(int id);
    }
}
```

By default, only one [Sorting Layer](#), *Default*, is defined. To add a definition, click on the **Sorting Layer** pull-down in the [SpriteRenderer](#) or [Canvas Inspector](#), and select **Add Sorting Layer**. Or go to *Edit -> Project Settings*, and select **Tags and Layers** to add them.

**Sorting layers** will be drawn first (i.e., with the smallest array index). Note that the **Sorting Layer** of [Canvas](#) is only taken into account when the **Render Mode** is set to **World Space**.

If you want to log all **Sorting Layers** from the script, you can write the following code.

```
public static void DumpSortingLayers()
{
```

```

foreach (var layer in UnityEngine.SortingLayer.layers)
{
    Debug.LogFormat("name:{0}, value:{1}, id:{2}", layer.name, layer.value, layer.id);
}
}

```

The output result is as follows.

```

name:SortingLayerMinus1, value:-1, id:885640979
name:Default, value:0, id:0
name:SortingLayerPlus1, value:1, id:813054017

```

`SortingLayer.layers` are sorted in ascending order by `value`, with the lowest `value` being drawn first, and the `Default` value is fixed at `0`. `id` must be unique to each **Sorting Layer**.

Adding a **Sorting Layer** from a script is a bit tricky: the **Sorting Layer** is stored in the `TagManager.asset` in the `ProjectSettings` folder, so you have to open it as a `SerializedObject` and add elements to it. This is done in Editor mode. This needs to be done in Editor mode (even if you do it in Play mode, it will revert to the original state when you stop).

The following is a sample code for adding a **Sorting Layer** from a script.

```

using UnityEngine;
#if UNITY_EDITOR
using UnityEditor;

public class AddSortingLayer : MonoBehaviour
{
    public static void Add(string layerName, int index = -1)
    {
        SerializedObject serializedObject = new UnityEditor.SerializedObject(UnityEditor.AssetDatabase.LoadMainAssetAtPath("ProjectSettings/TagManager.asset"));
        SerializedProperty sortingLayers = serializedObject.FindProperty("m_SortingLayers");

        for (int i = 0; i < sortingLayers.arraySize; i++)
        {
            string nameStringValue = sortingLayers.FindPropertyRelative("name").stringValue;
            if (nameStringValue == layerName)
            {

```

```

        return;
    }

    if (index < 0 || sortingLayers.arraySize < index)
    {
        index = sortingLayers.arraySize;
    }

    sortingLayers.InsertArrayElementAtIndex(index);
    SerializedProperty newLayer = sortingLayers.GetArrayElementAtIndex(index);

    newLayer.FindPropertyRelative("name").stringValue = layerName;
    newLayer.FindPropertyRelative("uniqueID").intValue = Random.Range(int.MinValue, int.MaxValue);

    serializedObject.ApplyModifiedProperties();
}
#endif

```

SortingLayer.layers is automatically set based on the array index. The value of the Sorting Layer before the Default is negative, and the value of the **Sorting Layer** after the Default is positive.

## Order in Layer (sortingOrder)

**Order in Layer** is an integer value ([int](#) type) that determines the drawing order of the Renderer or Canvas in the same **Sorting Layer**.

The default value is [0](#).

It can be accessed from scripts as the [sortingOrder](#) property.

*Definition of sortingOrder in the Renderer*

```
namespace UnityEngine
{
    public class Renderer : Component
    {
        ...
        public int sortingOrder
        {
            ...
        }
    }
}
```

*Definition of sortingOrder in Canvas*

```
namespace UnityEngine
{
    public sealed class Canvas : Behaviour
    {
        public int sortingOrder
        {
            ...
        }
    }
}
```

## Render Queue

RenderQueue is defined as an enumeration type in UnityEngine.

```
/// <summary>
/// <para> Determine the order in which objects are rendered</para>
/// </summary>
public enum RenderQueue
{
    /// <summary>
    /// <para> This render queue will be rendered before any of the others</para>
    /// </summary>
    Background = 1000,
    /// <summary>
    /// <para> Opaque geometry uses this queue</para>
    /// </summary>
    Geometry = 2000,
    /// <summary>
    /// <para> Alpha tested geometry uses this queue</para>
    /// </summary>
    AlphaTest = 2450,
    /// <summary>
    /// <para> Last render queue for opacity</para>
    /// </summary>
    GeometryLast = 2500,
    /// <summary>
    /// <para> Rendered after Geometry and AlphaTest, from back to front</para>
    /// </summary>
    Transparent = 3000,
    /// <summary>
    /// <para> Render queue for overlay effect</para>
    /// </summary>
    Overlay = 4000
}
```

`RenderQueue` can be specified in the `Material's renderQueue` property or in the shader; if the `Material's renderQueue` is `-1`, which is the default value, the `Queue` value set in the shader will be used. If the `Material's renderQueue` is set to `-1`, which is the default value, the `Queue` value set in the shader will be used; conversely, if the `Material's renderQueue` is set to anything other than `-1`, the `Queue` value in the shader will be ignored.

```
Shader "Transparent Queue Example"
{
    SubShader
    {
        "Queue" = "Transparent"
        Tags { "Queue" = "Transparent" }
        ...
    }
}
```

Queues up to `2500 (= Geometry + 500)` are provided for opaque objects, and the order in which objects are drawn is optimized accordingly for performance.

`Skybox` set in `Camera` will be drawn between opaque and transparent objects (timing between `GeometryLast` of `2500` and `2501`). By drawing the skybox after the opaque object, we can save the fill rate.

Render queues larger than `2500` are considered for transparent objects, and objects are rendered in order from furthest to closest.

It is also possible to define a new queue between the defined queues.

#### *Custom queue settings*

```
Tags { "Queue" = "Geometry+1" }
```

The `RenderQueue` in this case will be `2001`, and will be drawn after the other normal opaque objects (`Geometry` queues).

Geometry drawn by the `Canvas` is drawn with a `Transparent` queue. In other words, it is always drawn with alpha blending from back to front. In this case, rendering an object that is completely covered by opaque polygons is useless (i.e., overdraw) and consumes the fill-rate capability of the GPU.

## Improved rendering performance

### Fill rate

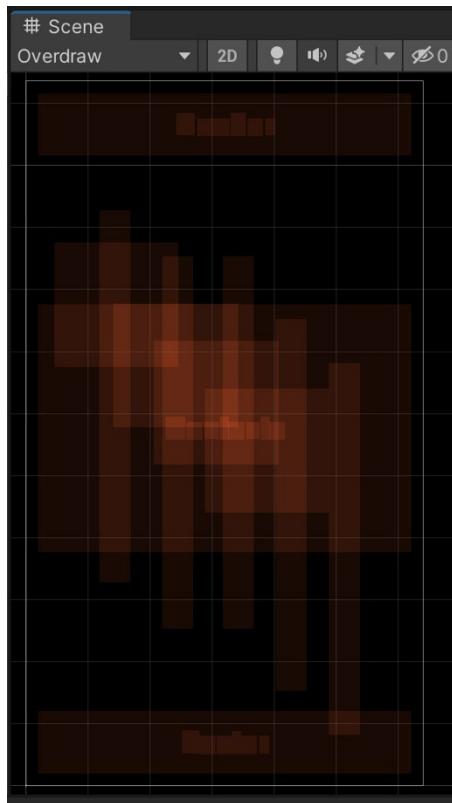
The fill rate is simply the maximum speed at which a fragment ( $\approx$  polygon pixels) can be filled. Since the number of vertices used in UI rendering is not very large, the fill rate is often the performance bottleneck. Therefore, it is important to reduce the load on the GPU fragment pipeline, i.e., to reduce fragmentation. Especially in mobile devices, the fill rate is kept low to reduce power consumption, so reducing fragmentation is a top priority.

To reduce the load on the fragment pipeline, it is necessary to simplify the fragment shader or reduce the number of pixels to be sampled.

In most cases, the shaders for uGUI will be the same as the built-in UI shaders, so in order to reduce the load on the fragment pipeline, we will focus on reducing the number of pixels sampled. To reduce the number of pixels to be sampled, we can simply reduce the texture resolution and drawing area. This means reviewing whether higher-resolution textures are being used than necessary, whether overlapping UI elements are causing overdraws, and so on.

## Overdraw

The redrawing of the same pixel is called overdraw. Overdraw is displayed as a brightly colored area by selecting **OverDraw** from the drop-down menu in the upper left corner of the **Scene View**.



In order to reduce overdraws, it is important to disable elements that are not visible to the player. Try to hide all UI elements that are hidden underneath the opaque background UI so that they are not drawn. The specifics of how to do this are described later in the section on *showing/hiding UI elements*.

Also, make sure that there are no UI elements that have an alpha value of **0** for the color of **Graphic** (e.g. **Image**) or the Material alpha value. These will trigger a draw call even if they are not displayed on the screen.

## Show/Hide UI elements

When showing or hiding individual parts of the UI, it is common to activate or deactivate the [GameObject](#) at the root of the UI. This way, the disabled UI will not receive any input or callbacks.

However, this will cause [Canvas](#) to discard the *VBO (Vertex Buffer Object)* data. Re-enabling this UI will result in a rebuild of [Canvas](#) (all of the Graphic rebuild, Layout rebuild, and Canvas batch build).

If this situation occurs frequently, the increased CPU usage will reduce the frame rate of the application. In addition, activating/deactivating [GameObjects](#) and activating/deactivating components such as [Image](#) will cause the relatively high-load [Graphic.OnEnable\(\)](#) is called, which consumes CPU time. In addition, *GC Alloc* occurs only in Editor, and since it is only in Editor, *GC Alloc* does not occur in the actual device, but it interferes with profiling, so let's consider another way to show/hide the UI.

### Note

When [GetComponent\(\)](#) is called on a non-existent component, a *GC Alloc* of about 0.5 KB is generated only on Editor. The *GC Alloc* shown above is due to this behavior.

*Packages/com.unity.ugui/Runtime/UI/Core/MaskableGraphic.cs*

```
protected override void OnDisable()
{
    ...
    // If Mask is not attached, GC Alloc will occur only on Editor
    if (GetComponent<Mask>() != null)
```

**Note**

Note that [TryGetComponent\(\)](#) does not generate a *GC Alloc* even if it is called on a non-existent component.

The following are a few ways to show/hide UI elements.

1. Set the [alpha](#) of [CanvasGroup](#) to 0.

As already explained, by setting the [alpha](#) of the [CanvasGroup](#) to 0, it is possible to hide itself and its child elements. In this case, if they are hidden, no draw calls will be generated. Note, however, that the Raycast itself is still alive and will respond to touch. To disable touch, we need to set [blocksRaycasts](#) in [CanvasGroup](#) to [false](#). Also, since each component itself is still active, [Update\(\)](#) and coroutines will continue to be called.

2. Set the [alpha](#) of the [CanvasRenderer](#) to 0.

As already explained, by setting the [alpha](#) of the [CanvasRenderer](#) to 0, it is possible to make itself invisible. In this case, if it is hidden, no draw call will be generated. Note, however, that if this is not done, the Raycast itself is still alive and will respond to touch. Also, since other components are still active, [Update\(\)](#) and coroutines will continue to be called.

3. Change the [localScale](#) of [RectTransform](#)

By setting the [localScale](#) of [RectTransform](#) to [Vector3.one](#) or [Vector3.zero](#), the UI element can be shown or hidden. In this case, the raycast rectangle is also (usually) set to zero, so it will not respond to touch. In this case, the raycast rectangle is also (usually) set to zero, so it will not respond to touch. However, each component itself will remain active, as in the other methods.

Graphic and Layout rebuilds can be avoided by any of the above methods 1 to 3, but Canvas batch builds will still occur.

## Graphic scaling

When scaling UI elements, you can change the `sizeDelta` or `localScale` of `RectTransform`, but changing the `localScale` has better performance. If you change the `sizeDelta`, the vertex data will be dirty, which will result in a Graphic rebuild. If you change `sizeDelta`, the vertex data will become dirty, which will cause a Graphic rebuild, so it is better to scale via `localScale` whenever possible, especially when animating.

## Using shaders to change colors in the UI

To indicate that a button cannot be pressed, the color of the button is often made monotone (gray). If we prepare two buttons to achieve this and switch them each time, the number of UI elements will naturally increase and the rebuild time will become longer. In this case, writing a shader that displays a monotone image will fulfill the requirement without increasing the number of UI elements. This shader is described later in the *HSV color space shader* section of this chapter.

## Don't create GameObjects instead of folders

Placing empty `GameObjects` in a `Canvas` to group UI elements together will increase the time required for a Canvas rebuild. If you have a Canvas that changes frequently, try to avoid creating `GameObjects` in place of such folders.

## Omit rendering of the 3D world space behind the UI.

If an opaque fullscreen UI is displayed in front of the camera that is showing the 3D world space, rendering the world space is completely useless. In this case, the rendering load can be reduced by disabling the camera showing the 3D world space.

Even if the case is not that extreme, if most of the screen is covered with UI and the 3D world space (camera and objects) does not move, you can reduce the CPU and GPU load (although memory usage for `RenderTexture` will increase) by rendering the 3D world space to `RenderTexture` beforehand and using that texture as the background UI. If the 3D world space is not moving (camera and objects), rendering the 3D world space into a `RenderTexture` in advance and using that texture as the background UI (although memory usage for the `RenderTexture` will increase) will reduce CPU and GPU load. In this case, you can reduce the overdraw by rendering only the visible area instead of the entire background UI.

## Integrate images

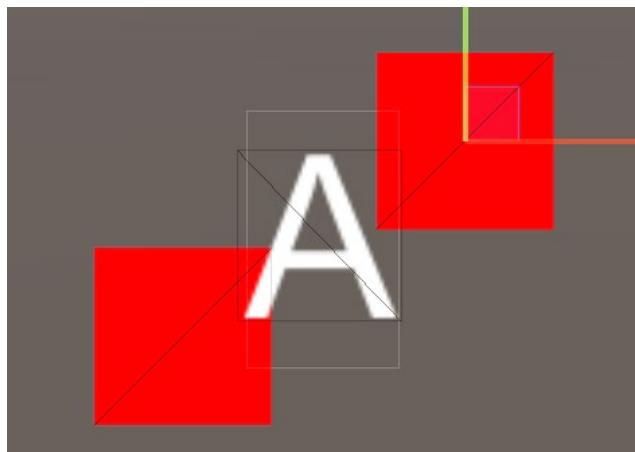
It is common practice to combine standard UI parts and decorate them to create the final UI look. However, since [CanvasRenderer](#) is a [Transparent](#) queue, it is rendered from back to front, and this approach leads to increased draw calls and overdraws.

Therefore, if possible, it is effective to merge the parts into a single image to improve performance. This will reduce the CPU/GPU load, but on the other hand, will increase the texture memory usage. The balance between the two is a case-by-case basis, but if you are concerned about the frame rate, you may want to consider integrating the parts. For information on how to reduce the increased texture memory usage in this case, the *Texture Format* section of this chapter may be useful.

## Adjust the order in the hierarchy.

UI elements are built in order from back to front, and this order is determined by the order of the objects in the hierarchy; objects higher in the hierarchy are considered to be further back in the hierarchy. Canvas batch builds go through the hierarchy from top to bottom, lumping together objects that have the same material and texture, but nothing in between.

As described later in *Chapter 11 Profiling*, the **Profiler** and **Frame Debugger** can be used to investigate if something is stuck between objects. In the figure below, a drawable object is stuck between two other objects that should be able to batch together.



This problem often occurs when the [Image](#) and [Text](#) are placed close together, and it is common for the [Text](#) border to inadvertently overlap the nearby [Image](#). This is because most of the polygons of the [Text](#) are transparent, so it is difficult to notice. This can be solved in two ways.

1. Move objects that interfere with the batch to the front or back of the group of objects that will be grouped into the batch.
2. Adjust the position of the object to eliminate the invisible overlap area.

Just by observing the number of draw calls in the **Frame Debugger**, you can probably find the order and position that can reduce the number of useless draw calls caused by overlapping UI elements.

## Shader

The UI shaders used for uGUI are pre-installed in Unity, and the default *UI/Default* shaders are used if no material is set for the [Graphic](#) component.

If you want to see what's inside the built-in UI shaders, you can download and view each version of the built-in shaders from Unity's download page.

<https://unity3d.com/get-unity/download/archive>

### Default UI shaders

Let's take a look at the contents of the default *UI/Default* shader, which is the *UI-Default.shader* in the *DefaultResourcesExtra/UI* folder of the built-in shaders you downloaded. Here is the default shader with Japanese comments.

*DefaultResourcesExtra/UI/UI-Default.shader*

```
Shader "UI/Default"
{
    Properties
    {
        // In order for batching to work when the textures match, the
        // Set the texture via MaterialPropertyBlock instead of directly from Material.
        // (Note that UI shaders basically cannot use MaterialPropertyBlocks.)
        // It is the CanvasRenderer that actually sets the texture.
        [PerRendererData] _MainTex ("Sprite Texture", 2D) = "white" {}

        // Color
        _Color ("Tint", Color) = (1,1,1,1)

        // Stencil comparison function
        // Defined in UnityEngine.Rendering.CompareFunction
        // https://docs.unity3d.com/ScriptReference/Rendering.CompareFunction.html
        // 8 is Always, which means the stencil test always succeeds
        _StencilComp ("Stencil Comparison", Float) = 8

        // Base value for stencil test (0-255)
        _Stencil ("Stencil ID", Float) = 0
    }
}
```

```

// Behavior on successful stencil test
// Defined in UnityEngine.Rendering.StencilOp
// https://docs.unity3d.com/ScriptReference/Rendering.StencilOp.html
// 0 is Keep, no changes will be made
_SStencilOp ("Stencil Operation", Float) = 0

// Mask that specifies the bit to write the reference value to the buffer after the stencil test is
performed.
// 0xFF, so reference value and buffer contents are compared as is
_SStencilWriteMask ("Stencil Write Mask", Float) = 255

// OR mask to apply to both the base value and the buffer contents before stencil testing
// 0xFF, so both the base value and buffer contents are compared as is
_SStencilReadMask ("Stencil Read Mask", Float) = 255

// Set the color channel to not reflect drawing
// Defined in UnityEngine.Rendering.ColorWriteMask
// https://docs.unity3d.com/ScriptReference/Rendering.ColorWriteMask.html
// 15 is All, which outputs all channels (A/B/G/R)
_ColorMask ("Color Mask", Float) = 15

// Whether to define UNITY_UI_ALPHA_CLIP or not.
// If 0, don't define it.
// Not necessary if you don't use mask
[Toggle(UNITY_UI_ALPHA_CLIP)] _UseUIAlphaClip ("Use Alpha Clip", Float) = 0
}

SubShader
{
    // Use tags to specify when/how to render
    // https://docs.unity3d.com/Manual/SL-SubShaderTags.html
    Tags
    {
        // RenderQueue is Transparent because it's for UI
        "Queue"="Transparent"

        // not affected by the Projector component
        "IgnoreProjector"="True"

        // Shader classification, different from RenderQueue
        "RenderType"="Transparent"
}

```

```

// How to display the material view under the Inspector.
// Default is Sphere, but you can choose Plane (2D) or Skybox
"PreviewType"="Plane"

// Set this to False if you want to make it clear that this shader will not work with Sprite
// and atlases
// Basically, True is fine.
"CanUseSpriteAtlas"="True"
}

// actually set the stencil value specified by the property
// https://docs.unity3d.com/Manual/SL-Stencil.html
Stencil
{
    // base value for stencil test
    Ref [_Stencil]

    // Comparison function
    Comp [_StencilComp]

    // behavior on successful stencil test
    Pass [_StencilOp]

    // Bit mask for buffer read
    ReadMask [_StencilReadMask]

    // bitmask for buffer write
    WriteMask [_StencilWriteMask]
}

// https://docs.unity3d.com/Manual/SL-CullAndDepth.html
// No culling needed because of UI
Cull Off

// Legacy fixed-function writing (deprecated)
// https://docs.unity3d.com/Manual/SL-Material.html
// Off is basically fine now (not just for UI)
Lighting Off

// Write to depth buffer
// https://docs.unity3d.com/Manual/SL-CullAndDepth.html
// Transparent, so no need for ZWrite

```

```
ZWrite Off
```

```
// How to do a depth test
// https://docs.unity3d.com/Manual/SL-CullAndDepth.html
// If Canvas is Overlay, Always (always draw)
// otherwise LEqual (draw if distance to drawn object is equal to or closer than distance)
ZTest [unity_GUIZTestMode]
```

```
// https://docs.unity3d.com/Manual/SL-Blend.html
// Pixel blending is now premultiplied transparency since Unity 2020.1
// https://issuetracker.unity3d.com/issues/transparent-ui-gameobject-ignores-opaque-ui-gameobject-when-using-rendertexture
// If you want the old blend, use Blend SrcAlpha OneMinusSrcAlpha to remove the multiplied transparency from the fragment shader
Blend One OneMinusSrcAlpha
```

```
// Use the value set in the property to set the color channel that does not reflect the drawing
ColorMask [_ColorMask]
```

```
Pass
```

```
{
    // name used in UsePass
    // https://docs.unity3d.com/Manual/SL-Name.html
    Name "Default"
```

```
// Start Cg/HLSL
CGPROGRAM
```

```
// HLSL snippet
// https://docs.unity3d.com/Manual/SL-ShaderPrograms.html
```

```
// Specify the function name of the vertex shader
#pragma vertex vert
```

```
// specify the function name of the fragment shader
#pragma fragment frag
```

```
// target level is for all platforms
// https://docs.unity3d.com/Manual/SL-ShaderCompileTargets.html
#pragma target 2.0
```

```
// Specify the include file
```

```

// The location of the include file is (where Unity is installed)/Editor/Data/CGIncludes
#include "UnityCG.cginc"
#include "UnityUI.cginc"

// Shader variants to toggle clipping functionality on and off depending on whether Mask
or RectMask2D is enabled
// The UNITY_UI_CLIP_RECT keyword is not global, it's local because it's only OK for
this shader
#pragma multi_compile_local _ UNITY_UI_CLIP_RECT

// We have two shader variants, one without alpha clipping and one with
// The UNITY_UI_ALPHACLIP keyword is not global, it is local because it is OK for thi
s shader only.
#pragma multi_compile_local _ UNITY_UI_ALPHACLIP

// Define the mesh vertex data
struct appdata_t
{
    // position
    float4 vertex : POSITION;

    // vertex color
    float4 color : COLOR;

    // 1st UV coordinate
    float2 texcoord : TEXCOORD0;

    // when instancing is enabled
    // uint instanceID : SV_InstanceID
    // is added.
    // See UnityInstancing.cginc for details.
    UNITY_VERTEX_INPUT_INSTANCE_ID
};

// Data passed from the vertex shader to the fragment shader
struct v2f
{
    // clip coordinates of the vertex
    // SV (System Value) because it is the value used by the system (used by the GPU for r
asterization)
    float4 vertex : SV_POSITION;

```

```

// color
fixed4 color : COLOR;

// 1st UV coordinate
float2 texcoord : TEXCOORD0;

// the second UV coordinate is passed with the vertex's position in world space
float4 worldPosition : TEXCOORD1;

// store and pass the mask data in the third UV coordinate
half4 mask : TEXCOORD2;

// For VR. Enables rendering of both eyes in a single pass.
UNITY_VERTEX_OUTPUT_STEREO
};

// To reference texture data, receive texture sampler type values via properties
sampler2D _MainTex;

// color
fixed4 _Color;

// Automatically set by Unity for the UI.
// (1,1,1,0) if the texture used is of type Alpha8, otherwise (0,0,0,0)
fixed4 _TextureSampleAdd;

// SetClipRect(), etc. with CanvasRenderer.EnableRectClipping()
float4 _ClipRect;

// Add _ST to the texture variable name to get the Tiling and Offset values.
// x, y are the Tiling values x, y, z, w are the Offset values z, w
float4 _MainTex_ST;

// The range of the soft mask (edge blur function) introduced in Unity 2020.1
// SetClipSoftness(), etc. to CanvasRenderer.clippingSoftness
float _MaskSoftnessX;
float _MaskSoftnessY;

// Vertex shader
// receive appdata_t and return v2f
v2f vert(appdata_t v)
{

```

```

// variable to pass to fragment shader
v2f OUT;

// Reflect the eye information for VR and the coordinates for each instancing for GPU i
nstancing
// See UnityInstancing.cginc
UNITY_SETUP_INSTANCE_ID(v);

// Tell the GPU the eye of the texture array for VR
UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(out);

// Convert the coordinates of the vertices in object space to the clip space of the camera
// from UnityShaderUtilities.cginc
// mul(UNITY_MATRIX_VP, float4(mul(unity_ObjectToWorld, float4(inPos, 1.0))).x
yz, 1.0);
// UNITY_MATRIX_VP : current view * projection matrix
// unity_ObjectToWorld : current model matrix
// https://docs.unity3d.com/ja/2018.4/Manual/SL-UnityShaderVariables.html
// In practice, this is equal to mul(UNITY_MATRIX_MVP, v.vertex)
float4 vPosition = UnityObjectToClipPos(v.vertex);

// Pass the world space of the vertex to the second UV coordinate
OUT.worldPosition = v.vertex;

// pass the clip coordinates of the transformed vertex
OUT.vertex = vPosition;

// w is the distance from the camera
float2 pixelSize = vPosition.w;

// _ScreenParams : current screen (render target) size
// UNITY_MATRIX_P : current projection matrix
// find the size corresponding to 1 pixel
pixelSize /= float2(1, 1) * abs(mul((float2x2)UNITY_MATRIX_P, _ScreenParams.xy))
);

// Create UV for mask texture with reduced precision
float4 clampedRect = clamp(_ClipRect, -2e10, 2e10);
float2 maskUV = (v.vertex.xy - clampedRect.xy) / (clampedRect.zw - clampedRect.xy)
;

// Cut off the masked area and create the UV of the texture

```

```

OUT.texcoord = float4(v.texcoord.x, v.texcoord.y, maskUV.x, maskUV.y);

// Data for clipping with soft mask taken into account
OUT.mask = half4(v.vertex.xy * 2 - clampedRect.xy - clampedRect.zw,
0.25 / (0.25 * half2(_MaskSoftnessX, _MaskSoftnessY) + abs(pixelSize.xy)));

// Multiply the vertex color by the property color
OUT.color = v.color * _Color;

return OUT;
}

// fragment shader
fixed4 frag(v2f IN) : SV_Target
{
    // sample color from texture
    half4 color = (tex2D(_MainTex, IN.texcoord) + _TextureSampleAdd) * IN.color;

    #ifdef UNITY_UI_CLIP_RECT
    // Soft mask aware clipping
    half2 m = saturate(_ClipRect.zw - _ClipRect.xy - abs(IN.mask.xy)) * IN.mask.zw;
    color.a *= m.x * m.y;
    #endif

    #ifdef UNITY_UI_ALPHA_CLIP
    // If the alpha is less than 0.001 (= almost transparent), discard the pixel.
    // If the edges are messy, you may want to increase the number
    clip(color.a - 0.001);
    #endif

    // Premultiplied transparency, so multiply RGB by Alpha
    color.rgb *= color.a;

    return color;
}
// End Cg/HLSL
ENDCG
}
}
}
}

```

## Custom UI shaders

If you don't need features such as stencil masks or clipping, you can use your own customized UI shaders. Also, as mentioned in the comment in the default UI shader above, pixel blending is now premultiplied transparency in Unity 2020.1, which may cause a slight performance degradation (although premultiplied transparency is the color that should be displayed). So, it is worth customizing.

### Lightweight shaders

For example, let's create a lightweight UI shader using the following policy.

- Remove [Mask](#) and [RectMask2D](#) support.
- Make blending like in Unity 2019.4 and earlier, instead of multiplied transparency.
- Don't use the material's color (output the texture color as is).

The actual shader code looks like the following.

```
Shader "UI/LightWeight"
{
    Properties
    {
        [PerRendererData] _MainTex("Base (RGB), Alpha (A)", 2D) = "white" {}
    }

    SubShader
    {
        Tags
        {
            "Queue" = "Transparent"
            "IgnoreProjector" = "True"
            "RenderType" = "Transparent"
            "PreviewType" = "Plane"
            "CanUseSpriteAtlas" = "True"
        }

        Cull Off
        Lighting Off
        ZWrite Off
        ZTest[unity_GUIZTestMode]
```

```

Blend SrcAlpha OneMinusSrcAlpha

Pass
{
    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag

    #include "UnityCG.cginc"
    #include "UnityUI.cginc"

    struct appdata_t
    {
        float4 vertex : POSITION;
        float2 texcoord : TEXCOORD0;
    };

    struct v2f
    {
        float4 vertex : SV_POSITION;
        half2 texcoord : TEXCOORD0;
        float4 worldPosition : TEXCOORD1;
    };

    v2f vert(appdata_t IN)
    {
        v2f OUT;
        OUT.worldPosition = IN.vertex;
        OUT.vertex = UnityObjectToClipPos(OUT.worldPosition);
        OUT.texcoord = IN.texcoord;

        return OUT;
    }

    sampler2D _MainTex;
    fixed4 frag(v2f IN) : SV_Target
    {
        return tex2D(_MainTex, IN.texcoord);
    }
    ENDCG
}

```

$\left. \begin{array}{c} \\ \end{array} \right\}$

## HSV color space shader

By converting RGB color space to HSV space, a variety of expressions become possible. For example, by reducing saturation, you can make an image monotone to express that the button cannot be pressed, or reduce image resources by making a blue image into a red image.

The following is an HSV color space shader based on the default shader, and you can move the **Hue**, **Saturation**, **Brightness**, and Contrast slide bars from **Inspecotr** to see how they look different. We have prepared two versions, one for Unity 2020.1 and later, and one for Unity 2019.4 and earlier, so please use them accordingly.

*HSV color space shader based on the UI default shader in Unity 2020.1 and later*

```
Shader "UI/HSV"
{
    Properties
    {
        [PerRendererData] _MainTex("Sprite Texture", 2D) = "white" {}
        _Color("Tint", Color) = (1,1,1,1)

        _Hue("Hue", Range(0.0, 1.0)) = 0.0
        _Saturation("Saturation", Range(0.0, 1.0)) = 0.5
        _Brightness("Brightness", Range(0.0, 1.0)) = 0.5
        _Contrast("Contrast", Range(0, 1.0)) = 0.5

        _StencilComp("Stencil Comparison", Float) = 8
        _Stencil("Stencil ID", Float) = 0
        _StencilOp("Stencil Operation", Float) = 0
        _StencilWriteMask("Stencil Write Mask", Float) = 255
        _StencilReadMask("Stencil Read Mask", Float) = 255

        _ColorMask("Color Mask", Float) = 15

        [Toggle(UNITY_UI_ALPHACLIP)] _UseUIAlphaClip("Use Alpha Clip", Float) = 0
    }

    SubShader
    {
        Tags
        {
            
```

```

"Queue" = "Transparent"
"IgnoreProjector" = "True"
"RenderType" = "Transparent"
"PreviewType" = "Plane"
"CanUseSpriteAtlas" = "True"
}

Stencil
{
    Ref[_Stencil]
    Comp[_StencilComp]
    Pass[_StencilOp]
    ReadMask[_StencilReadMask]
    WriteMask[_StencilWriteMask]
}

Cull Off
Lighting Off
ZWrite Off
ZTest[unity_GUIZTestMode]
Blend One OneMinusSrcAlpha
ColorMask[_ColorMask]

Pass
{
    CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #pragma target 2.0

        #include "UnityCG.cginc"
        #include "UnityUI.cginc"

        #pragma multi_compile_local _ UNITY_UI_CLIP_RECT
        #pragma multi_compile_local _ UNITY_UI_ALPHACLIP

        // Change the color according to the hue
        inline float3 tweekHue(float3 color, float hue)
    {
        float3 k = float3(0.57735, 0.57735, 0.57735);
        float hueAngle = radians(hue);
        float cosHue = cos(hueAngle);
    }
}

```

```

        float sinHue = sin(hueAngle);

        return color * cosHue + cross(k, color) * sinHue + k * dot(k, color) * (1 - cosHue);
    }

// Convert RGB color to HSV
inline float4 convertToHSV(float4 rgbaColor, fixed4 hsvc)
{
    float hue = 360 * hsvc.r;
    float saturation = hsvc.g * 2;
    float brightness = hsvc.b * 2 - 1;
    float contrast = hsvc.a * 2;

    float4 hsvcColor;
    hsvcColor.rgb = tweekHue(rgbaColor.rgb, hue);
    hsvcColor.rgb = (hsvcColor.rgb - 0.5f) * contrast + 0.5f;
    hsvcColor.rgb = hsvcColor.rgb + brightness;
    float3 intensity = dot(hsvcColor.rgb, float3(0.39, 0.59, 0.11));
    hsvcColor.rgb = lerp(intensity, hsvcColor.rgb, saturation);

    return hsvcColor;
}

struct appdata_t
{
    float4 vertex : POSITION;
    float4 color : COLOR;
    float2 texcoord : TEXCOORD0;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};

struct v2f
{
    float4 vertex : SV_POSITION;
    fixed4 color : COLOR;
    float2 texcoord : TEXCOORD0;
    float4 worldPosition : TEXCOORD1;
    half4 mask : TEXCOORD2;
    UNITY_VERTEX_OUTPUT_STEREO
};

sampler2D _MainTex;

```

```

fixed4 _Color;

// Receive hue/saturation/lightness/contrast from properties
fixed _Hue;
fixed _Saturation;
fixed _Brightness;
fixed _Contrast;

fixed4 _TextureSampleAdd;
float4 _ClipRect;
float4 _MainTex_ST;
float _MaskSoftnessX;
float _MaskSoftnessY;

// Vertex shader is the same as the default shader
v2f vert(appdata_t v)
{
    v2f OUT;
    UNITY_SETUP_INSTANCE_ID(v);
    UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(OUT);
    float4 vPosition = UnityObjectToClipPos(v.vertex);
    OUT.worldPosition = v.vertex;
    OUT.vertex = vPosition;

    float2 pixelSize = vPosition.w;
    pixelSize /= float2(1, 1) * abs(mul((float2x2)UNITY_MATRIX_P, _ScreenParams.xy));

    float4 clampedRect = clamp(_ClipRect, -2e10, 2e10);
    float2 maskUV = (v.vertex.xy - clampedRect.xy) / (clampedRect.zw - clampedRect.xy);
    OUT.texcoord = float4(v.texcoord.x, v.texcoord.y, maskUV.x, maskUV.y);
    OUT.mask = half4(v.vertex.xy * 2 - clampedRect.xy - clampedRect.zw, 0.25 / (0.25 * hal
f2(_MaskSoftnessX, _MaskSoftnessY) + abs(pixelSize.xy)));

    OUT.color = v.color * _Color;
    return OUT;
}

fixed4 frag(v2f IN) : SV_Target
{
    half4 color = (tex2D(_MainTex, IN.texcoord) + _TextureSampleAdd) * IN.color;

    #ifdef UNITY_UI_CLIP_RECT

```

```

half2 m = saturate(_ClipRect.zw - _ClipRect.xy - abs(IN.mask.xy)) * IN.mask.zw;
color.a *= m.x * m.y;
#endif

#ifndef UNITY_UI_ALPHACLIP
clip(color.a - 0.001);
#endif

// Up to this point, the same as the default shader

// Adjust HSV color space from here
fixed4 hsvc = fixed4(_Hue, _Saturation, _Brightness, _Contrast);
float4 hsvcColor = convertToHSV(color, hsvc);
color.rgb = hsvcColor * color.a;

return color;
}

ENDCG
}
}

```

*HSV colorspace shader based on the UI default shader in Unity 2019.4 and earlier*

```

Shader "UI/LegacyHSV"
{
Properties
{
    [PerRendererData] _MainTex("Sprite Texture", 2D) = "white" {}
    _Color("Tint", Color) = (1,1,1,1)

    _Hue("Hue", Range(0.0, 1.0)) = 0.0
    _Saturation("Saturation", Range(0.0, 1.0)) = 0.5
    _Brightness("Brightness", Range(0.0, 1.0)) = 0.5
    _Contrast("Contrast", Range(0, 1.0)) = 0.5

    _StencilComp("Stencil Comparison", Float) = 8
    _Stencil("Stencil ID", Float) = 0
    _StencilOp("Stencil Operation", Float) = 0
    _StencilWriteMask("Stencil Write Mask", Float) = 255
    _StencilReadMask("Stencil Read Mask", Float) = 255
}

```

```

        _ColorMask("Color Mask", Float) = 15

        [Toggle(UNITY_UI_ALPHACLIP)] _UseUIAlphaClip("Use Alpha Clip", Float) = 0
    }

SubShader
{
    Tags
    {
        "Queue" = "Transparent"
        "IgnoreProjector" = "True"
        "RenderType" = "Transparent"
        "PreviewType" = "Plane"
        "CanUseSpriteAtlas" = "True"
    }

    Stencil
    {
        Ref[_Stencil]
        Comp[_StencilComp]
        Pass[_StencilOp]
        ReadMask[_StencilReadMask]
        WriteMask[_StencilWriteMask]
    }

    Cull Off
    Lighting Off
    ZWrite Off
    ZTest[unity_GUIZTestMode]
    Blend SrcAlpha OneMinusSrcAlpha
    ColorMask[_ColorMask]

    Pass
    {
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #pragma target 2.0

        #include "UnityCG.cginc"
        #include "UnityUI.cginc"
    }
}

```

```

#pragma multi_compile_local _ UNITY_UI_CLIP_RECT
#pragma multi_compile_local _ UNITY_UI_ALPHACLIP

// Change the color according to the hue
inline float3 tweekHue(float3 color, float hue)
{
    float3 k = float3(0.57735, 0.57735, 0.57735);
    float hueAngle = radians(hue);
    float cosHue = cos(hueAngle);
    float sinHue = sin(hueAngle);

    return color * cosHue + cross(k, color) * sinHue + k * dot(k, color) * (1 - cosHue);
}

// Convert RGB color to HSV
inline float4 convertToHSV(float4 rgbaColor, fixed4 hsvc)
{
    float hue = 360 * hsvc.r;
    float saturation = hsvc.g * 2;
    float brightness = hsvc.b * 2 - 1;
    float contrast = hsvc.a * 2;

    float4 hsvcColor;
    hsvcColor.rgb = tweekHue(rgbaColor.rgb, hue);
    hsvcColor.rgb = (hsvcColor.rgb - 0.5f) * contrast + 0.5f;
    hsvcColor.rgb = hsvcColor.rgb + brightness;
    float3 intensity = dot(hsvcColor.rgb, float3(0.39, 0.59, 0.11));
    hsvcColor.rgb = lerp(intensity, hsvcColor.rgb, saturation);

    return hsvcColor;
}

struct appdata_t
{
    float4 vertex : POSITION;
    float4 color : COLOR;
    float2 texcoord : TEXCOORD0;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};

struct v2f
{

```

```

float4 vertex : SV_POSITION;
fixed4 color : COLOR;
float2 texcoord : TEXCOORD0;
float4 worldPosition : TEXCOORD1;
half4 mask : TEXCOORD2;
UNITY_VERTEX_OUTPUT_STEREO
};

sampler2D _MainTex;
fixed4 _Color;

// Receive hue/saturation/lightness/contrast from properties
fixed _Hue;
fixed _Saturation;
fixed _Brightness;
fixed _Contrast;

fixed4 _TextureSampleAdd;
float4 _ClipRect;
float4 _MainTex_ST;
float _MaskSoftnessX;
float _MaskSoftnessY;

// Vertex shader is the same as the default shader
v2f vert(appdata_t v)
{
    v2f OUT;
    UNITY_SETUP_INSTANCE_ID(v);
    UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(OUT);
    float4 vPosition = UnityObjectToClipPos(v.vertex);
    OUT.worldPosition = v.vertex;
    OUT.vertex = vPosition;

    float2 pixelSize = vPosition.w;
    pixelSize /= float2(1, 1) * abs(mul((float2x2)UNITY_MATRIX_P, _ScreenParams.xy));

    float4 clampedRect = clamp(_ClipRect, -2e10, 2e10);
    float2 maskUV = (v.vertex.xy - clampedRect.xy) / (clampedRect.zw - clampedRect.xy);
    OUT.texcoord = float4(v.texcoord.x, v.texcoord.y, maskUV.x, maskUV.y);
    OUT.mask = half4(v.vertex.xy * 2 - clampedRect.xy - clampedRect.zw, 0.25 / (0.25 * hal
f2(_MaskSoftnessX, _MaskSoftnessY) + abs(pixelSize.xy)));
}

```

```

        OUT.color = v.color * _Color;
        return OUT;
    }

fixed4 frag(v2f IN) : SV_Target
{
    half4 color = (tex2D(_MainTex, IN.texcoord) + _TextureSampleAdd) * IN.color;

    #ifdef UNITY_UI_CLIP_RECT
    half2 m = saturate({_ClipRect.zw - _ClipRect.xy - abs(IN.mask.xy)) * IN.mask.zw);
    color.a *= m.x * m.y;
    #endif

    #ifdef UNITY_UI_ALPHACLIP
    clip(color.a - 0.001);
    #endif
    // Up to this point, the same as the default shader

    // Adjust HSV color space from here
    fixed4 hsvc = fixed4(_Hue, _Saturation, _Brightness, _Contrast);
    float4 hsvcColor = convertToHSV(color, hsvc);
    color.rgb = hsvcColor;

    return color;
}
ENDCG
}
}

```

## Texture format

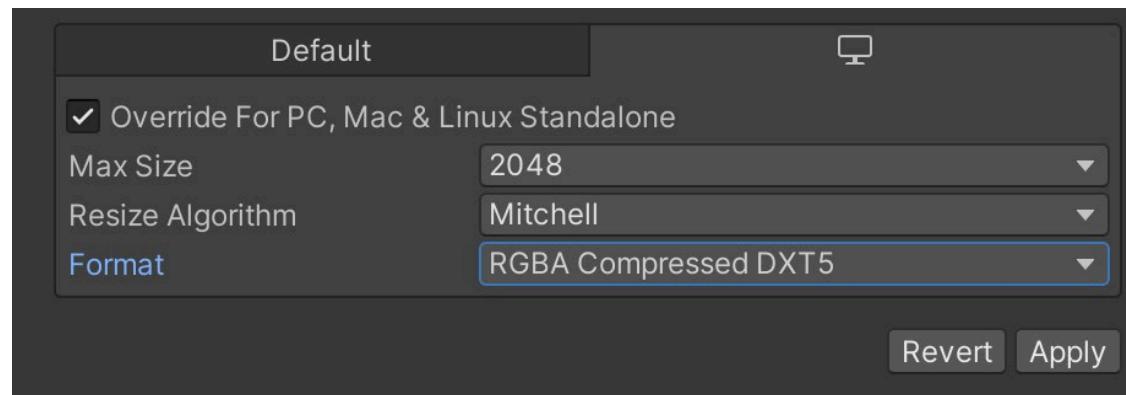
Unity supports a variety of texture formats, which are defined as the `TextureFormat` enumeration type.

The list of supported texture formats can be found in the official documentation.

<https://docs.unity3d.com/2021.2/Documentation/ScriptReference/TextureFormat.htm>

Typical texture formats are uncompressed 32 bit RGBA32, 16 bit RGB565 / RGBA4444, PRVTC for iOS, ETC1 / ETC2 / ATC for Android, ASTC for both iOS and Android, and DXT / BC for DirectX. In addition, there are three other types of software.

The texture format of the texture file can be checked in the **Import Settings** displayed in the **Inspector**. In the platform-specific override panel, there is a tab for each platform in addition to the **Default** tab, and individual import settings used.



In the following, we will look at typical formats from the perspective of UI use.

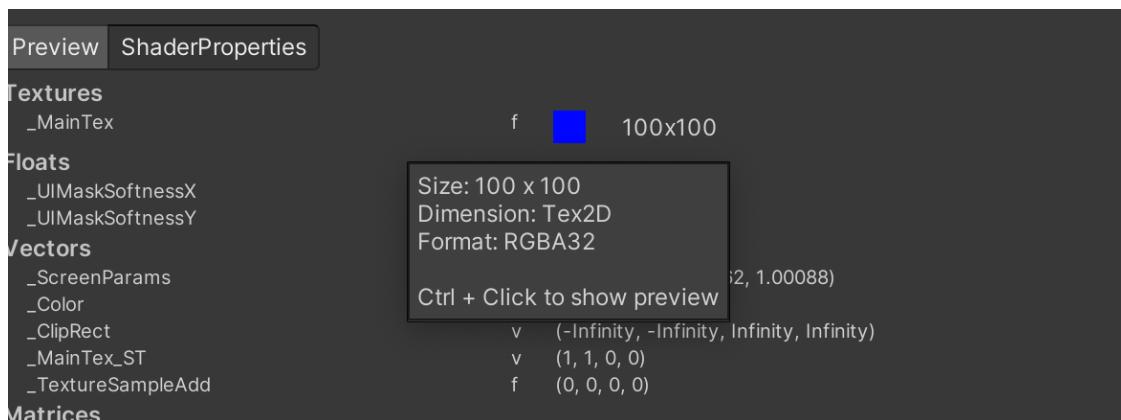
## Uncompressed 32 bit

The uncompressed 32-bit format is the texture format when no compression is applied. Since 32 bits = 4 bytes per pixel, a 1024 x 1024 size texture will consume  $1024 \times 1024 \times 4 = 4 \text{ MB}$  of memory.

Keep in mind that file size does not equal memory consumption; even a small file size format such as jpeg can consume 4MB of memory for a 1024 x 1024 size if the format in memory is True Color 32bit. PNG files can also be compressed. PNG files can also be compressed to reduce the file size.

To see what texture format is actually being used, hover the mouse cursor over the texture in the **Frame Debugger**. The texture format will be displayed in the pop-up Format.

In the case of the **RGBA32** texture format, the following is displayed.



Basically, there are few opportunities to use the uncompressed 32 bit format in actual games. The exceptions are key visuals such as the title screen and CG of important events. These images need to be pixel-perfect, exactly as the designer or illustrator intended. Also, since these images are often published as is in magazines and web media, we want to display them with as little degradation as possible. In this case, the uncompressed 32-bit format is the best choice.

## Uncompressed 16bit

The uncompressed 16-bit format is a texture format in which colors are reduced from 32 bits to 16 bits to reduce the amount of memory used. It is common to use [RGB4444](#) with alpha, and [RGB565](#) without alpha.

If you use Unity's import settings to set a 16-bit texture, it will simply drop the lower bits and reduce the color, resulting in a dirty gradient. To solve this problem, there are color reduction tools such as Optpix. Another known method is to apply the error diffusion method during import to prevent image degradation.

- keijiro's conversion tool for [RGBA444](#)

<https://github.com/keijiro/unity-dither4444>

- A conversion tool for [RGB565](#) that improves on the above

<https://blog.jp.uwa4d.com/2019/11/04/unity%E7%94%BB%E5%83%8F%E6%9C%80%E9%81%A9%E5%8C%96%E3%83%84%E3%83%BC%E3%83%AB/>

Although 16-bit textures may often be used for UI textures, a combination of compression formats such as PVRTC/ETC/ASTC/DXT (and alpha-only textures) may also provide sufficient quality. It is important to consult with designers, illustrators, and art directors before deciding which format to use. This is an important point, so I would like to reiterate that the decision of which format to use should not be made at the sole discretion of the engineer, but should always be discussed with the designer, illustrator, or art director.

## PVRTC

PVRTC is a texture format mainly for iOS, and there are four different formats depending on the number of bits per pixel and the presence of alpha.

Format	Number of bits per pixel	Have alpha?
PVRTC_RGBA4	4	Yes
PVRTC_RGB4	4	No
PVRTC_RGBA2	2	Yes
PVRTC_RGB2	2	No

[PVRTC\\_RGBA4](#) and [PVRTC\\_RGBA2](#) are also supported by Android devices with PowerVR, but PVRTC can be thought of as a texture format that is practically exclusive to iOS.

Although [PVRTC\\_RGBA2](#) may be able to represent textures with little variation, in reality, even [PVRTC\\_RGBA4](#) is quite rough. If you include the alpha component, you will lose a lot of expressiveness, so it is better to use two separate files, one for RGB only and one for alpha only. Even so, it is 8 bits per pixel, which is a quarter of the memory saving of the 32 bit format.

The following is a shader for uGUI that can be used with the method of having a separate alpha texture in PVRTC and ETC1 by keijiro (<https://github.com/keijiro/unity-alphamask>). The implementation only uses the R component of the texture prepared for the alpha mask to generate the final alpha value, so it can be used for other applications than PVRTC and ETC1.

*Shaders that use alpha masks based on the UI default shaders in Unity 2020.1 and later*

```
Shader "UI/AlphaMask"
{
    Properties
    {
        [PerRendererData] _MainTex("Sprite Texture", 2D) = "white" {}

        // Texture for alpha mask (use only R component)
        _AlphaTex("Alpha", 2D) = "white" {}

        _Color("Tint", Color) = (1,1,1,1)

        _StencilComp("Stencil Comparison", Float) = 8
        _Stencil("Stencil ID", Float) = 0
        _StencilOp("Stencil Operation", Float) = 0
    }
}
```

```

    _StencilWriteMask("Stencil Write Mask", Float) = 255
    _StencilReadMask("Stencil Read Mask", Float) = 255

    _ColorMask("Color Mask", Float) = 15

    [Toggle(UNITY_UI_ALPHA_CLIP)] _UseUIAlphaClip("Use Alpha Clip", Float) = 0
}

SubShader
{
    Tags
    {
        "Queue" = "Transparent"
        "IgnoreProjector" = "True"
        "RenderType" = "Transparent"
        "PreviewType" = "Plane"
        "CanUseSpriteAtlas" = "True"
    }

    Stencil
    {
        Ref[_Stencil]
        Comp[_StencilComp]
        Pass[_StencilOp]
        ReadMask[_StencilReadMask]
        WriteMask[_StencilWriteMask]
    }

    Cull Off
    Lighting Off
    ZWrite Off
    ZTest[unity_GUIZTestMode]
    Blend One OneMinusSrcAlpha
    ColorMask[_ColorMask]

    Pass
    {
        Name "Default"
    }
}

CGPROGRAM
#pragma vertex vert
#pragma fragment frag
#pragma target 2.0

```

```

#include "UnityCG.cginc"
#include "UnityUI.cginc"

#pragma multi_compile_local _ UNITY_UI_CLIP_RECT
#pragma multi_compile_local _ UNITY_UI_ALPHACLIP

struct appdata_t
{
    float4 vertex : POSITION;
    float4 color : COLOR;
    float2 texcoord : TEXCOORD0;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};

struct v2f
{
    float4 vertex : SV_POSITION;
    fixed4 color : COLOR;
    float2 texcoord : TEXCOORD0;

    // UV for alpha mask
    float2 alphacoord : TEXCOORD1;

    float4 worldPosition : TEXCOORD2;
    half4 mask : TEXCOORD3;
    UNITY_VERTEX_OUTPUT_STEREO
};

sampler2D _MainTex;

// Receive texture for alpha mask from property
sampler2D _AlphaTex;

fixed4 _Color;
fixed4 _TextureSampleAdd;
float4 _ClipRect;
float4 _MainTex_ST;

// Tiling and Offset of textures for alpha masks
float4 _AlphaTex_ST;

```

```

float _MaskSoftnessX;
float _MaskSoftnessY;

v2f vert(appdata_t v)
{
    v2f OUT;
    UNITY_SETUP_INSTANCE_ID(v);
    UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(OUT);
    float4 vPosition = UnityObjectToClipPos(v.vertex);
    OUT.worldPosition = v.vertex;
    OUT.vertex = vPosition;

    float2 pixelSize = vPosition.w;
    pixelSize /= float2(1, 1) * abs(mul((float2x2)UNITY_MATRIX_P, _ScreenParams.xy));
);

    float4 clampedRect = clamp(_ClipRect, -2e10, 2e10);
    float2 maskUV = (v.vertex.xy - clampedRect.xy) / (clampedRect.zw - clampedRect.xy);
;
    OUT.texcoord = float4(v.texcoord.x, v.texcoord.y, maskUV.x, maskUV.y);

    // UV setting for alpha mask
    OUT.alphacoord = TRANSFORM_TEX(v.texcoord.xy, _AlphaTex);

    OUT.mask = half4(v.vertex.xy * 2 - clampedRect.xy - clampedRect.zw, 0.25 / (0.25 *
half2(_MaskSoftnessX, _MaskSoftnessY) + abs(pixelSize.xy)));
;

    OUT.color = v.color * _Color;
    return OUT;
}

fixed4 frag(v2f IN) : SV_Target
{
    half4 color = (tex2D(_MainTex, IN.texcoord) + _TextureSampleAdd) * IN.color;

    // Receive the color of the alpha mask
    half4 alphamask = tex2D(_AlphaTex, IN.alphacoord);

    // Apply the R component of the alpha mask to the alpha
    color.a *= alphamask.r * alphamask.r * alphamask.r;

#define UNITY_UI_CLIP_RECT

```

```

half2 m = saturate(_ClipRect.zw - _ClipRect.xy - abs(IN.mask.xy)) * IN.mask.zw;
color.a *= m.x * m.y;
#endif

#ifndef UNITY_UI_ALPHA_CLIP
clip(color.a - 0.001);
#endif

color.rgb *= color.a;

return color;
}
ENDCG
}
}

}

```

*Shaders that use alpha masks based on the UI default shaders in Unity 2019.4 and earlier*

```

Shader "UI/LegacyAlphaMask"
{
Properties
{
    [PerRendererData] _MainTex("Base", 2D) = "white" {}

    // Texture for alpha mask (use only R component)
    _AlphaTex("Alpha", 2D) = "white" {}

    _Color("Tint", Color) = (1,1,1,1)

    _StencilComp("Stencil Comparison", Float) = 8
    _Stencil("Stencil ID", Float) = 0
    _StencilOp("Stencil Operation", Float) = 0
    _StencilWriteMask("Stencil Write Mask", Float) = 255
    _StencilReadMask("Stencil Read Mask", Float) = 255

    _ColorMask("Color Mask", Float) = 15

    [Toggle(UNITY_UI_ALPHA_CLIP)] _UseUIAlphaClip("Use Alpha Clip", Float) = 0
}

```

```

SubShader
{
    Tags
    {
        "Queue" = "Transparent"
        "IgnoreProjector" = "True"
        "RenderType" = "Transparent"
        "PreviewType" = "Plane"
        "CanUseSpriteAtlas" = "True"
    }

    Stencil
    {
        Ref[_Stencil]
        Comp[_StencilComp]
        Pass[_StencilOp]
        ReadMask[_StencilReadMask]
        WriteMask[_StencilWriteMask]
    }

    Cull Off
    Lighting Off
    ZWrite Off
    ZTest[unity_GUIZTestMode]
    Blend SrcAlpha OneMinusSrcAlpha
    ColorMask[_ColorMask]

    Pass
    {
        Name "Default"
        CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma target 2.0

            #include "UnityCG.cginc"
            #include "UnityUI.cginc"

            #pragma multi_compile_local _ UNITY_UI_CLIP_RECT
            #pragma multi_compile_local _ UNITY_UI_ALPHACLIP
    }
}

```

```

struct appdata_t
{
    float4 vertex : POSITION;
    float4 color : COLOR;
    float2 texcoord : TEXCOORD0;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};

struct v2f
{
    float4 vertex : SV_POSITION;
    fixed4 color : COLOR;
    float2 texcoord : TEXCOORD0;

    // UV for alpha mask
    float2 alphacoord : TEXCOORD1;

    float4 worldPosition : TEXCOORD2;
    half4 mask : TEXCOORD3;
    UNITY_VERTEX_OUTPUT_STEREO
};

sampler2D _MainTex;

// Receive texture for alpha mask from property
sampler2D _AlphaTex;

fixed4 _Color;
fixed4 _TextureSampleAdd;
float4 _ClipRect;
float4 _MainTex_ST;

// Tiling and Offset of textures for alpha masks
float4 _AlphaTex_ST;

float _UIMaskSoftnessX;
float _UIMaskSoftnessY;

v2f vert(appdata_t v)
{
    v2f OUT;
    UNITY_SETUP_INSTANCE_ID(v);
}

```

```

UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(OUT);
float4 vPosition = UnityObjectToClipPos(v.vertex);
OUT.worldPosition = v.vertex;
OUT.vertex = vPosition;

float2 pixelSize = vPosition.w;
pixelSize /= float2(1, 1) * abs(mul((float2x2)UNITY_MATRIX_P, _ScreenParams.xy))
);

float4 clampedRect = clamp(_ClipRect, -2e10, 2e10);
float2 maskUV = (v.vertex.xy - clampedRect.xy) / (clampedRect.zw - clampedRect.xy)
;
OUT.texcoord = TRANSFORM_TEX(v.texcoord.xy, _MainTex);

// UV setting for alpha mask
OUT.alphacoord = TRANSFORM_TEX(v.texcoord.xy, _AlphaTex);

OUT.mask = half4(v.vertex.xy * 2 - clampedRect.xy - clampedRect.zw, 0.25 / (0.25 *
half2(_UIMaskSoftnessX, _UIMaskSoftnessY) + abs(pixelSize.xy )));

OUT.color = v.color * _Color;
return OUT;
}

fixed4 frag(v2f IN) : SV_Target
{
    half4 color = IN.color * tex2D(_MainTex, IN.texcoord);

    // Receive the color of the alpha mask
    half4 alphamask = tex2D(_AlphaTex, IN.alphacoord);

    // Apply the R component of the alpha mask to the alpha
    color.a *= alphamask.r * alphamask.r * alphamask.r;

#ifdef UNITY_UI_CLIP_RECT
    half2 m = saturate(_ClipRect.zw - _ClipRect.xy - abs(IN.mask.xy)) * IN.mask.zw;
    color.a *= m.x * m.y;
#endif

#ifdef UNITY_UI_ALPHA_CLIP
    clip(color.a - 0.001);
#endif
}

```

```
        return color;
    }
ENDCG
}
}
```

Note that the ASTC format can be used in iOS devices with A8 processor (iPhone 6) or later, so if you do not support older devices, it is recommended to use ASTC instead of PVRTC.

## ETC1

ETC1 is a texture format that can be used on Android devices that support OpenGL ES 2.0 or higher, and on iOS devices with Unity 2017.3 or later, OpenGL ES 2.0 or higher on A7 chips (iPhone 5S) or later, or Metal. However, from the standpoint of quality and usability, it is not possible to use it on iOS. However, there are few opportunities to use ETC1 on iOS due to its quality and usability. However, if you want to reduce the hassle of managing assets for each platform, there is no reason not to use ETC1 on iOS, but we do not recommend it because it may lead to unknown problems in environments with little experience.

`ETC_RGB4` and `ETC_RGB4Crunched` are defined as `TextureFormat` enums, and as the name suggests, they use 4 bits per pixel. The most significant feature is that there is no alpha component. In other words, it is the same as `PVRTC_RGB4` in PVRTC. Newer Android devices can use ETC2 and ASTC, so there is no reason to actively use ETC1 if you want to cut off support for older devices (but in fact it is still used).

Format	Number of bits per pixel	Have alpha?	Annotation
<code>ETC_RGB4</code>	4	No	
<code>ETC_RGB4Crunched</code>	4	No	Crunch compression of <code>ETC_RGB4</code>

To solve the ETC1's shortcoming of having no alpha component, Unity's built-in shaders include a UI default shader specifically for the ETC1.

*ETC1 dedicated UI default shader*

```
Shader "UI/DefaultETC1"
{
    Properties
    {
        [PerRendererData] _MainTex ("Sprite Texture", 2D) = "white" {}
        // Texture for Alpha
        [PerRendererData] _AlphaTex ("Sprite Alpha Texture", 2D) = "white" {}
        _Color ("Tint", Color) = (1,1,1,1)

        _StencilComp ("Stencil Comparison", Float) = 8
        _Stencil ("Stencil ID", Float) = 0
        _StencilOp ("Stencil Operation", Float) = 0
        _StencilWriteMask ("Stencil Write Mask", Float) = 255
    }
}
```

```

StencilReadMask ("Stencil Read Mask", Float) = 255

_ColorMask ("Color Mask", Float) = 15

[Toggle(UNITY_UI_ALPHACLIP)] _UseUIAlphaClip ("Use Alpha Clip", Float) = 0
}

SubShader
{
    Tags
    {
        "Queue"="Transparent"
        "IgnoreProjector"="True"
        "RenderType"="Transparent"
        "PreviewType"="Plane"
        "CanUseSpriteAtlas"="True"
    }

    Stencil
    {
        Ref [_Stencil]
        Comp [_StencilComp]
        Pass [_StencilOp]
        ReadMask [_StencilReadMask]
        WriteMask [_StencilWriteMask]
    }

    Cull Off
    Lighting Off
    ZWrite Off
    ZTest [unity_GUIZTestMode]
    Blend One OneMinusSrcAlpha
    ColorMask [_ColorMask]

    Pass
    {
        Name "Default"
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #pragma target 2.0
    }
}

```

```

#include "UnityCG.cginc"
#include "UnityUI.cginc"

#pragma multi_compile_local _ UNITY_UI_CLIP_RECT
#pragma multi_compile_local _ UNITY_UI_ALPHA_CLIP

struct appdata_t
{
    float4 vertex : POSITION;
    float4 color : COLOR;
    float2 texcoord : TEXCOORD0;
};

struct v2f
{
    float4 vertex : SV_POSITION;
    fixed4 color : COLOR;
    // UV for alpha is the same as the main texture UV
    float2 texcoord : TEXCOORD0;
    float4 worldPosition : TEXCOORD1;
    half4 mask : TEXCOORD2;
};

sampler2D _MainTex;
fixed4 _Color;
fixed4 _TextureSampleAdd;
float4 _ClipRect;

// Use main texture's tiling and Offset for alpha
float4 _MainTex_ST;
float _MaskSoftnessX;
float _MaskSoftnessY;

v2f vert(appdata_t IN)
{
    v2f OUT;
    float4 vPosition = UnityObjectToClipPos(IN.vertex);
    OUT.worldPosition = IN.vertex;
    OUT.vertex = vPosition;

    OUT.texcoord = TRANSFORM_TEX(IN.texcoord, _MainTex);
}

```

```

#define UNITY_HALF_TEXEL_OFFSET
OUT.vertex.xy += (_ScreenParams.zw-1.0) * float2(-1,1) * OUT.vertex.w;
#endif

float2 pixelSize = vPosition.w;
pixelSize /= float2(1, 1) * abs(mul((float2x2)UNITY_MATRIX_P, _ScreenParams.xy))
);

float4 clampedRect = clamp(_ClipRect, -2e10, 2e10);
float2 maskUV = (IN.vertex.xy - clampedRect.xy) / (clampedRect.zw - clampedRect.x
y);
OUT.texcoord = float4(IN.texcoord.x, IN.texcoord.y, maskUV.x, maskUV.y);
OUT.mask = half4(IN.vertex.xy * 2 - clampedRect.xy - clampedRect.zw, 0.25 / (0.25 *
half2(_MaskSoftnessX, _MaskSoftnessY) + abs(pixelSize.xy)));

OUT.color = IN.color * _Color;
return OUT;
}

sampler2D _AlphaTex;

fixed4 frag(v2f IN) : SV_Target
{
    fixed4 color = UnityGetUIDiffuseColor(IN.texcoord, _MainTex, _AlphaTex, _Texture
SampleAdd) * IN.color;

#define UNITY_UI_CLIP_RECT
    half2 m = saturate((_ClipRect.zw - _ClipRect.xy - abs(IN.mask.xy)) * IN.mask.zw);
    color.a *= m.x * m.y;
#endif

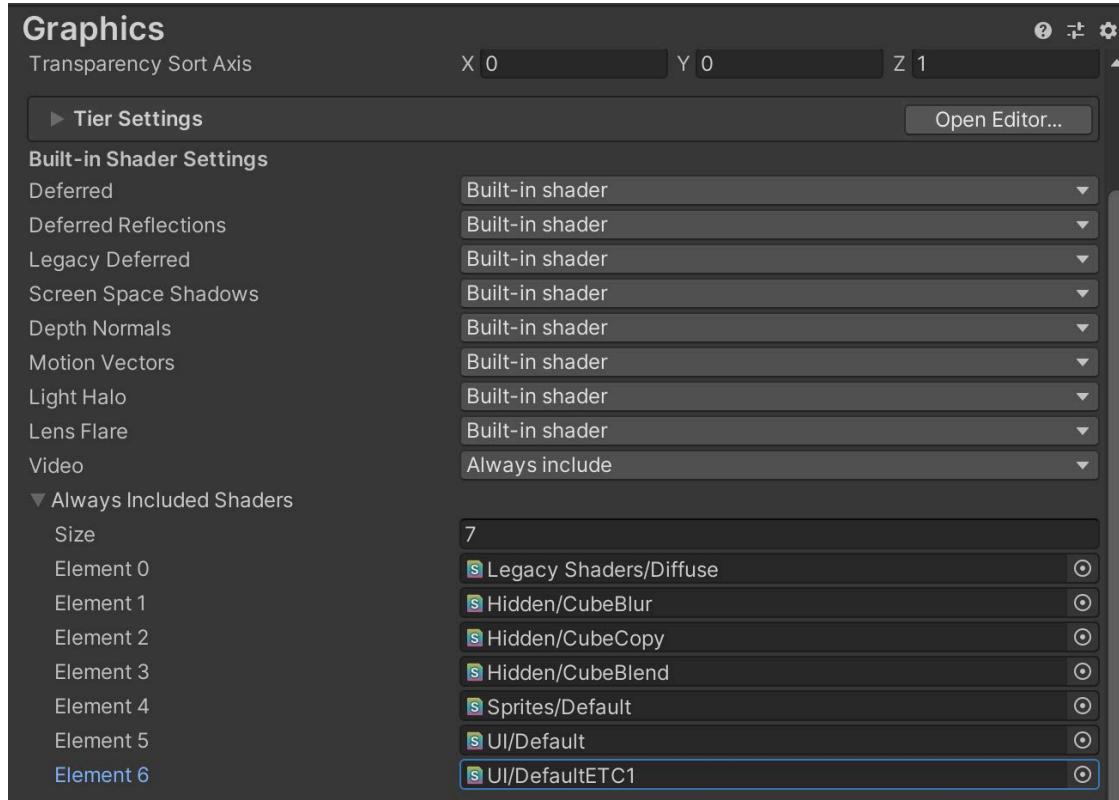
#define UNITY_UI_ALPHA_CLIP
    clip (color.a - 0.001);
#endif

    color.rgb *= color.a;
    return color;
}
ENDCG
}

```

```
}
```

If you add the [UI/DefaultETC1](#) shader to *Always Included Shaders* in the *Graphic* section of *ProjectSettings*, the [Image](#) component will use this shader as the UI default shader when using ETC1.



[GetETC1SupportedCanvasMaterial\(\)](#) will return a material that uses [UI/DefaultETC1](#), and the [Image](#) class will use it as a material.

```
/// <summary>
/// Default material cache with ETC1 and alpha textures.
/// </summary>
/// <remarks>
/// Stores the material of the Canvas that supports ETC1, obtained from GetETC1Supported
CanvasMaterial().
/// Note: In order to use ETC1 and alpha textured materials.
/// Make sure to specify UI/DefaultETC1 shaders in the Always Included Shaders list.
/// </remarks>
static public Material defaultETC1GraphicMaterial
{
    get
    {
        if (s_ETC1DefaultUI == null)
            s_ETC1DefaultUI = Canvas.GetETC1SupportedCanvasMaterial();
        return s_ETC1DefaultUI;
    }
}

/// <summary>
/// The material this Image will use.
/// If no material is specified, the default material will be used.
/// </summary>
public override Material material
{
    get
    {
        if (m_Material != null)
            return m_Material;
#if UNITY_EDITOR
        if (Application.isPlaying && activeSprite && activeSprite.associatedAlphaSplitTexture != null)
            return defaultETC1GraphicMaterial;
#else
        if (activeSprite && activeSprite.associatedAlphaSplitTexture != null)
            return defaultETC1GraphicMaterial;
#endif
```

```
        return defaultMaterial;
    }

    set
    {
        base.material = value;
    }
}
```

If you want to use ETC1 to draw with alpha, you can also use the shader with alpha mask described in the PVRTC section. However, the [UI/DefaultETC1](#) shader is lighter than the alpha mask shader described in the PVRTC section because it does not require a separate UV for the alpha, so the [UI/DefaultETC1](#) shader may be sufficient if you do not want to do anything elaborate.

The [ETC\\_RGB4Crunched](#) format, which is a Crunch compressed ETC1, is also available in Unity 2017.3 and later. If you check the Use crunch compression checkbox in the Import Settings of Inspector, Crunch compression, which is irreversible compression, will be enabled, and the texture file size will be reduced.

When using the [ETC\\_RGB4Crunched](#) format, the image is compressed to about 1.3 bits per pixel, depending on the image. This can save space and shorten the loading time of the texture file. At runtime, the image is converted to [ETC\\_RGB4](#) and expanded into memory, but there is almost no degradation in memory expansion speed at that time. However, the image quality will deteriorate considerably, so be sure to visually check whether the quality is good enough for the product.

## ETC2

ETC2 is a texture format that can be used on Android devices that support OpenGL ES 3.0 or higher, and can also be used on iOS devices with A7 chip (iPhone 5S) or later with OpenGL ES 3.0 or higher or Metal with Unity 2017.3 or higher. As with ETC1, there are not many opportunities to use it on iOS. Also, like ETC1, Crunch compression is available in Unity 2017.3 or later.

There are three texture formats that can be used for UI: [ETC2\\_RGB](#) with no 4-bit alpha for one pixel, [ETC2\\_RGBA8](#) with 8-bit alpha for one pixel, and [ETC2\\_RGBA8Crunched](#) with Crunch compression. There are also [ETC2\\_RGBA1](#), [EAC\\_R](#), [EAC\\_R\\_SIGNED](#), [EAC\\_RG](#), and [EAC\\_RG\\_SIGNED](#), but they are rarely used for UI.

Format	Number of bits per pixel	Have alpha?	Annotation
ETC2_RGB	4	No	
ETC2_RGBA8	8	Yes	
ETC2_RGBA8Crunched	8	Yes	Crunch compression of ETC2_RGBA8
ETC2_RGBA1	4	Yes	Alpha is 1 bit
EAC_R	4	No	Unsigned R only
EAC_R_SIGNED	4	No	Signed R only
EAC_RG	8	No	Unsigned RG only
EAC_RG_SIGNED	8	No	Signed RG only

Depending on the balance between image quality and file size, if you cannot use ASTC on Android, consider using [ETC2\\_RGBA8Crunched](#) for UI textures, and if you see problems with Crunch compression image quality, you may want to use [ETC2\\_RGBA8](#) instead. ETC2\_RGBA8.

## ASTC

ASTC is a texture format that can be used on newer iOS and Android devices. iOS can use OpenGL ES 3.0 or higher or Metal on devices with A8 chips (iPhone 6) or later, so there is no reason not to use it for new titles. For Android, it is available for OpenGL ES 3.2, OpenGL ES 3.1+AEP GPUs and some OpenGL ES 3.0, and most devices released after 2016 will support it.

The following table shows the status of texture format support on Google Play devices, as compiled by Google in September 2020.

<https://developer.android.com/guide/app-bundle/asset-delivery/texture-compression>

Texture format	Percentage of Google Play devices that are supported
ETC1	99%.
ETC2	87%.
ASTC	77%.
ATC	35%.
PVRTC	11%.
DXT1	0.7

As you can see, 77% of the terminals support ASTC. I hope you can use this number to persuade the decision makers somehow.

The number of bits per pixel in ASTC depends on the pixel block. The cleanest 4x4 pixel block is 8 bits per pixel, but the most compressed 12x12 pixel block is 0.89 bits per pixel, which compresses the data quite well.

Format	Number of bits per pixel	Have alpha?
ASTC_RGB_4x4	8	No
ASTC_RGB_5x5	5.12	No
ASTC_RGB_6x6	3.56	No
ASTC_RGB_8x8	2	No
ASTC_RGB_10x10	1.28	No
ASTC_RGB_12x12	0.89	No
ASTC_RGBA_4x4	8	Yes
ASTC_RGBA_5x5	5.12	Yes
ASTC_RGBA_6x6	3.56	Yes
ASTC_RGBA_8x8	2	Yes
ASTC_RGBA_10x10	1.28	Yes
ASTC_RGBA_12x12	0.89	Yes

The decision on which pixel block to use should be based on the artist's opinion, but in general, [ASTC\\_RGBA\\_4x4](#) for UI, [ASTC\\_RGB\\_8x8](#) for 3D normal maps, and [ASTC\\_RGB\\_10x10](#) for albedo should be fine.

## DXT / BC

DXT and BC are format for DirectX, and [BC7](#) is probably the most practical format for UI.

(computer) format	Number of bits per pixel	Have alpha?	Annotation
DXT1	4	No	
DXT5	8	Yes	
DXT1Crunched	4	No	Crunch compression on DXT1
DXT5Crunched	8	No	Crush compression in DXT5
BC4	4	No	R only
BC5	8	No	R and G only
BC6H	8	No	HDR
BC7	8	Yes	

[DXT5](#) is susceptible to severe color changes, so it may be difficult to use for UI-related applications where you want to display clear boundaries. On the other hand, it should be noted that [BC7](#) has the disadvantage of taking a long time to compress.

## Texture import settings

As mentioned above, the texture format of the texture file can be set from the **Import Settings** of **Inspector**, but it can also be set from the Editor script. To be more specific, you can get the **TextureImporter** using **OnPreprocessTexture()** in the class that inherits from **UnityEditor**. The following is a sample of the original texture import settings.

```
using UnityEngine;
using UnityEditor; // Note that this is an Editor script

public class MyAssetImportProcessor : AssetPostprocessor
{
    private void OnPreprocessTexture()
    {
        // The path to each asset will be passed as assetPath, so we can use it to import only specific
        // folders.
        if (assetPath.Contains("CompTextures"))
        {
            // the AssetImporter (import settings) for each asset will be passed as assetImporter.
            TextureImporter textureImporter = (TextureImporter)assetImporter;

            // Add the settings for each platform
            textureImporter.SetPlatformTextureSettings(new TextureImporterPlatformSettings
            {
                overridden = true,
                name = "Android",
                maxTextureSize = 4096,
                format = TextureImporterFormat.ETC2_RGB_A8,
                allowsAlphaSplitting = false
            });
            textureImporter.SetPlatformTextureSettings(new TextureImporterPlatformSettings
            {
                overridden = true,
                name = "iPhone",
                maxTextureSize = 4096,
                format = TextureImporterFormat.PVRTC_RGBA4,
                allowsAlphaSplitting = false
            });
        }
    }
}
```

```
    }
```

If you place this script file in an appropriate location such as the Editor folder, the **ImportSettings** will be set and can be checked in **Inspector**. If it is not set, right-click on the target folder or file and select **Reimport** to run the import script.

## RenderTexture

It is often the case that you want to display the result of rendering with a 3D camera as an image. In this case, the [RenderTexture](#) class is used.

```
public RenderTexture(int width, int height, int depth, RenderTextureFormat format= RenderTextureFormat.Default, RenderTextureReadWrite readWrite= RenderTextureReadWrite.Default);
public RenderTexture(RenderTexture textureToCopy);
public RenderTexture(RenderTextureDescriptor desc);
```

When creating a new [RenderTexture](#), pass the width, height, depth buffer bits, format, and color space conversion mode as arguments to the constructor. The width and height do not have to be a power of two or a square. [ARGB32](#) is often used for the format.

The depth buffer can be set to 0, 16, or 24 bits, but we recommend setting it to 24 unless there is a particular reason not to. 16 bit does not create a stencil buffer to be used for masking. Also, some Android devices may not support 16-bit depth. If you specify [RenderTextureReadWrite.Default](#) as the color space conversion mode, the color space conversion (gamma or linear) based on the project settings will be applied as is. If you want to specify more detailed settings, you can call the constructor passing [RenderTextureDescriptor](#) as an argument.

There are three things to keep in mind when creating a [RenderTexture](#).

1. [SystemInfo.SupportsRenderTextureFormat\(\)](#) to see if the format is supported. ARGB32 is supported on almost all platforms, but there are some devices that do not support [RenderTextureFormat.RGB565](#), which is a 16 bit texture. ARGB32 is supported on almost all platforms, but there are some devices that do not support [RenderTextureFormat.RGB565](#), which is a 16-bit texture. Please check before creating, and if it is not supported, prepare a fallback by using [RenderTextureFormat](#).
2. Immediately after new [RenderTexture](#) is created, the texture in memory is not actually created. It is created at the timing when it becomes active (when [RenderTexture.active](#) points to the [RenderTexture](#)). If you want to ensure that the texture is allocated in memory, you need to call [Create\(\)](#). Basically, it is recommended to call [Create\(\)](#) immediately after creating a new texture.
3. [RenderTexture](#) created by itself must be released by calling [Release\(\)](#), otherwise the texture in memory will not be released. It would be better to have [RenderTexture](#) as a member variable and use [OnDestroy\(\)](#) to release it.

Based on the above, the code to display the result rendered by the camera as a [RawImage](#) is described below.

```

using UnityEngine;
using UnityEngine.UI;

// Display the result of drawing from the camera as a RawImage
public class DrawRawImageFromCamera : MonoBehaviour
{
    private RenderTexture renderTexture;

    private void OnDestroy()
    {
        // RenderTexture must be released.
        if (renderTexture != null)
        {
            renderTexture. Release();
        }
    }

    public void Draw(RawImage rawImage, Camera targetCamera, int width, int height, int depth = 24, RenderTextureFormat renderTextureFormat = RenderTextureFormat.ARGB32)
    {
        if (!SystemInfo.SupportsRenderTextureFormat(renderTextureFormat))
        {
            Debug.LogErrorFormat("Unsupported format {0}, use ARGB32.", renderTextureFormat);
        }
        return;
    }

    // Create the RenderTexture with the appropriate size
    // It doesn't have to be a power of 2 size
    if (renderTexture == null)
    {
        renderTexture = new RenderTexture(width, height, depth, renderTextureFormat);
    }

    // Immediately after new, the actual render target has not been created, so read Create()
    renderTexture.Create();

    // set the camera render target to RenderTexture
    targetCamera. targetTexture = renderTexture;

    // Set the camera's render destination to RenderTexture targetCamera.
}

```

```
    rawImage.texture = renderTexture;  
}  
}
```

You may want to do some optimization, such as disabling the camera rendering once the camera has rendered to the [RenderTexture](#) to reduce the load. Note, however, that the contents of the [RenderTexture](#) can be lost (due to external factors such as screen savers). You should call [RenderTexture's IsCreated\(\)](#) method every frame to check if the content is lost, and if it is, call [Create\(\)](#) to render the camera's rendering result to [RenderTexture](#) again.

Note that the simplest way to wait for a camera to render once is to wait for one frame. If you want to wait strictly, you can set up a callback to [Camera.onPostRender](#) and determine if it is the camera you want. The following is a pseudo code to achieve this.

```
{  
    ...  
    // Set the callback and flag it after rendering  
    Camera.onPostRender += OnCameraPostRender;  
  
    yield return new WaitUntil(() => isRenderDone);  
    ...  
    // Remove the callback if it malfunctions  
    Camera.onPostRender -= OnCameraPostRender;  
}  
  
public static void OnCameraPostRender(Camera cam)  
{  
    // If cam is the desired camera.  
    isRenderDone = true;  
}
```

## Screen resolution

### Gets/Sets screen resolution

The currently running screen resolution can be obtained by `Screen.width` and `Screen.height`. The most important thing to keep in mind is to make sure that the screen resolution is not unintentionally high. The hardware resolutions of modern PCs and mobile devices are very high, and if the screen is set to full screen with the same resolution, the rendering load will be very high. `SetResolution()` to prevent this from happening, and it is highly recommended to set the resolution not to be too high.

```
public static void SetResolution(int width, int height, bool fullscreen);
public static void SetResolution(int width, int height, bool fullscreen, int preferredRefreshRate=0);
public static void SetResolution(int width, int height, FullScreenMode fullscreenMode, int preferredRefreshRate=0);
```

If the device does not support a resolution that matches `width` and `height`, the closest resolution will be selected. A list of resolutions supported in fullscreen can be obtained from `Screen.resolutions`, and it is safe to select a resolution from the list.

*Sample code to output the resolution list*

```
Resolution[] resolutions = Screen.resolutions;
foreach (var res in resolutions)
{
    Debug.LogFormat("{0} x {1} : {2}", res.width, res.height, res.refreshRate);
}
```

On Android devices, `Screen.resolutions` may return an empty array or just the current resolution, but it can (and should) be set to any resolution.

If `0` is specified for `preferredRefreshRate`, which specifies the refresh rate (this is the default behavior), the highest refresh rate will be selected. If `preferredRefreshRate` is set to something other than `0`, but that refresh rate is not supported, the highest refresh rate will be selected.

The resolution switch is done after the completion of the current frame. Basically, it is better to call it at startup. `SetResolution()` is not available on the Editor, so be sure to check the effect on the actual device.

### Getting the screen resolution in the Editor

In the Editor, `Screen.width` and `Screen.height` may not return the exact resolution of the **Game View**. When running in the Editor, the most reliable is the string returned by `UnityEditor`. `UnityStats.screenRes` returns a string like "640x1136", so you can use that.

```
int screenWidth = Screen.width;
int screenHeight = Screen.height;

#if UNITY_EDITOR
string[] editorScreenRes = UnityEditor.UnityStats.screenRes.Split('x');
if (editorScreenRes.Length >= 2)
{
    if (int.TryParse(editorScreenRes[0], out int editorScreenResWidth))
    {
        if (int.TryParse(editorScreenRes[1], out int editorScreenResHeight))
        {
            screenWidth = editorScreenResWidth;
            screenHeight = editorScreenResHeight;
        }
    }
}
#endif
```

## Chapter 4 Graphic

### Graphic Components

```
[DisallowMultipleComponent]
[RequireComponent(typeof(RectTransform))]
[ExecuteAlways]
public abstract class Graphic : UIBehaviour, ICanvasElement
```

The `Graphic` class is the base class for all visible UI components. If you want to create your own visible UI components, it is probably easiest to inherit from this class. As an example, here is a component that displays a rectangle that fills the `RectTransform` area.

```
using UnityEngine;
using UnityEngine.UI;

// Specify ExecuteAlways to display in the Scene view in Editor mode.
[ExecuteAlways]
// If the CanvasRenderer is not attached, it will not be drawn.
[RequireComponent(typeof(CanvasRenderer))]
// Display a colored rectangle filled with RectTransform
public class SimpleImage : Graphic
{
    protected override void OnPopulateMesh(VertexHelper vh)
    {
        // lower left vertex
        Vector2 vertexLeftBottom = Vector2.zero;

        // top right vertex
        Vector2 vertexRightTop = Vector2.zero;

        float width = rectTransform.rect.width;
        float height = rectTransform.rect.height;

        // Fill the lower left and upper right vertices with the RectTransform area
        vertexLeftBottom.x = (0 - rectTransform.pivot.x) * width;
        vertexLeftBottom.y = (0 - rectTransform.pivot.y) * height;
        vertexRightTop.x = (1 - rectTransform.pivot.x) * width;
```

```

vertexRightTop.y = (1 - rectTransform.pivot.y) * height;

// Clear the vertices of this mesh
vh.Clear();

// Each vertex to be added
UIVertex vert = UIVertex.simpleVert;

// Set the position and color and pass it to VertexHelper
vert.position = new Vector2(vertexLeftBottom.x, vertexLeftBottom.y);
vert.color = color;
vh.AddVert(vert);

vert.position = new Vector2(vertexLeftBottom.x, vertexRightTop.y);
vert.color = color;
vh.AddVert(vert);

vert.position = new Vector2(vertexRightTop.x, vertexRightTop.y);
vert.color = color;
vh.AddVert(vert);

vert.position = new Vector2(vertexRightTop.x, vertexLeftBottom.y);
vert.color = color;
vh.AddVert(vert);

// Set up two triangles using the four vertices passed to VertexHelper
vh.AddTriangle(0, 1, 2);
vh.AddTriangle(2, 3, 0);
}

}

```

The `VertexHelper` class is a utility class to support UI mesh generation, and the `Graphic` class holds `VertexHelper` as a static variable.

`[NonSerialized] private static readonly VertexHelper s_VertexHelper = new VertexHelper();`

The `Graphic` class uses this static `VertexHelper` to generate the vertices of its own mesh; the details of the `VertexHelper` are described below.

## Static property of Graphic

defaultGraphicMaterial

```
public static Material defaultGraphicMaterial { get; }
```

Get the material to be used if no material is specified. [GetDefaultCanvasMaterial\(\)](#) is actually cached and stored as a `static` variable.

## Properties of the Graphic

### canvas

```
public Canvas canvas { get; }
```

Get the [Canvas](#) to which this component belongs.

If the [Canvas](#) is nested, it returns the closest one.

The first time this property is accessed, [GetComponentsInParent\(\)](#) is called to cache the parent [Canvas](#) and return it. If you want to avoid the overhead of [GetComponentsInParent\(\)](#), it is better to keep the reference to the [Canvas](#) by yourself (e.g. by setting it in [Inspector](#)) instead of using this property.

### canvasRenderer

```
public CanvasRenderer canvasRenderer { get; }
```

Get the [CanvasRenderer](#) that this component is using.

If the [CanvasRenderer](#) is not [null](#), its value will be cached.

### color

```
public virtual Color color { get; set; }
```

Gets/Sets the color used as the vertex color.

If a different color is set than the previous one, [SetVerticesDirty\(\)](#) will be called to make the geometry dirty, which will later cause a Graphic rebuild.

## defaultMaterial

---

```
public virtual Material defaultMaterial { get; }
```

Get the [defaultGraphicMaterial](#) as it is.

## depth

---

```
public int depth { get; }
```

Get the order in the [Canvas](#).

The topmost element directly under the [Canvas](#) is [0](#), and its children or the elements below it are [1](#), and so on. In actuality, [CanvasRenderer.absoluteDepth](#) is returned.

If the first frame is not completed, or if the [GameObject](#) is inactive and not rendered by the [Canvas](#), you will get a value of [-1](#).

This value is used to reorder the results of the [GraphicRaycaster](#) raycast.

## mainTexture

---

```
public virtual Texture mainTexture { get; }
```

Get the main texture to be used for this [Graphic](#).

Since the [mainTexture](#) of the [Graphic](#) class only returns [Texture2D.whiteTexture](#), it is assumed to be overridden and implemented in each derived class.

## material

---

```
public virtual Material material { get; set; }
```

Gets/Sets the material.

If no material has been set, `defaultMaterial` will be returned. If a different material from the previous one is set, `SetMaterialDirty()` will be called to make the material dirty, which will result in a Graphic rebuild later.

### materialForRendering

```
public virtual Material materialForRendering { get; }
```

Gets the material that the `CanvasRenderer` will actually use for rendering.

However, if a component that implements the `IMaterialModifier` interface, such as `MaskableGraphic` or `Mask`, is attached, the converted material will be returned via `IMaterialModifier.GetModifiedMaterial()`. This implementation makes it possible to change the material used for rendering without affecting the original material.

### raycastPadding

```
public Vector4 raycastPadding { get; set; }
```

Gets/Sets the padding to expand the area of the raycast.

This property was introduced in Unity 2020.1. In `Vector4`, `x` is left, `y` is down, `z` is right, and `w` is up. Negative values make the area larger, while positive values make the area smaller.

### raycastTarget

```
public virtual bool raycastTarget { get; set; }
```

Gets/Sets whether this object is the target of ray casting (i.e., whether touch judgment is enabled).

Each time the setting is changed, `GraphicRegistry.RegisterRaycastGraphicForCanvas()` or `GraphicRegistry.UnregisterRaycastGraphicForCanvas()` is called to register/unregister the graph as a raycast target. `UnregistererRaycastGraphicForCanvas()` is called to register/unregister the canvas as a raycast target.

## rectTransform

---

```
public RectTransform rectTransform { get; }
```

Gets RectTransform.

Since [GetComponent\(\)](#) retrieves and returns the cached version, there is no overhead for the second and subsequent calls.

## Public methods of Graphic's

### CrossFadeAlpha

```
public virtual void CrossFadeAlpha(float alpha, float duration, bool ignoreTimeScale);
```

Gradually changes the color alpha value of the [CanvasRenderer](#) over time.

[CoroutineTween](#) is implemented by a dedicated tweening function inside uGUI (in [UnityEngine.UI](#) namespace.)

If [ignoreTimeScale](#) is [true](#), then [Time.unscaledDeltaTime](#), which is not affected by [Time.timeScale](#), will be used to measure the elapsed time. [Time.unscaledDeltaTime](#) will be used to measure the elapsed time. Conversely, if [ignoreTimeScale](#) is [false](#), [Time.deltaTime](#) will be used to measure the elapsed time.

### CrossFadeColor

```
public virtual void CrossFadeColor(Color targetColor, float duration, bool ignoreTimeScale, bool useAlpha);
public virtual void CrossFadeColor(Color targetColor, float duration, bool ignoreTimeScale, bool useAlpha, bool useRGB);
```

As with [CrossFadeAlpha\(\)](#), this will gradually change the color of the [CanvasRenderer](#) over time. If [useAlpha](#) is [true](#), the alpha value will change; if [useRGB](#) is [true](#), the RGB component will change. If [useAlpha](#) is [true](#), the alpha changes; if [useRGB](#) is [true](#), the RGB component changes. Of course, if neither [useAlpha](#) nor [useRGB](#) is [false](#), nothing happens.

### GetPixelAdjustedRect

```
public Rect GetPixelAdjustedRect();
```

Returns the [Rect](#) of the current [RectTransform](#) transformed to pixel perfect (i.e., coordinates rounded to the nearest integer).

However, if the `renderMode` of the `canvas` is `WorldSpace` or `pixelPerfect` is `false`, the rect itself will be returned.

### GraphicUpdateComplete

---

```
public virtual void GraphicUpdateComplete();
```

This method is called by the `CanvasUpdateRegistry` (via the `ICanvasElement` interface) when the Graphic rebuild is complete.

### LayoutComplete

---

```
public virtual void LayoutComplete();
```

This method is called by the `CanvasUpdateRegistry` (via the `ICanvasElement` interface) when the Layout rebuild is complete.

### OnCullingChanged

---

```
public virtual void OnCullingChanged();
```

This method is called when the `cull` of the `CanvasRenderer` is changed.

Registers itself with `CanvasUpdateRegistry.RegisterCanvasElementForGraphicRebuild()` if the vertex or material is dirty when `cull` becomes `false` (i.e., when it is rendered as usual).  
`RegistarCanvasElementForGraphicRebuild()`.

### OnRebuildRequested

---

```
public virtual void OnRebuildRequested();
```

Called when a Graphic rebuild is requested, such as when an asset is re-imported.

Called only in Editor Mode.

## PixelAdjustPoint

---

```
public Vector2 PixelAdjustPoint(Vector2 point);
```

Converts the local position of the pixel given in the argument to pixel perfect and returns it.

However, if the [renderMode](#) of [Canvas](#) is [WorldSpace](#) or [pixelPerfect](#) is [false](#), it will be returned without conversion.

## Raycast

---

```
public virtual bool Raycast(Vector2 sp, Camera eventCamera);
```

Returns [true](#) if the point specified by [sp](#) is a valid position for ray casting.

It is usually called from the [Raycast\(\)](#) method of the [GraphicRaycaster](#) component attached to the [Canvas](#).

Before this method is called, [GraphicRaycaster.Raycast\(\)](#) determines the raycast based on the [RectTransform rect](#), and if it is outside the [rect](#) range, [Graphic](#).

This method looks for components that implement the [ICanvasRaycastFilter](#) interface (e.g., [Image](#), [Mask](#), [RectMask2D](#)) attached to the same [GameObject](#) and returns [false](#) if any of them are valid. If [IsRaycastLocationValid\(\)](#) is [false](#), the method returns [false](#); if no component is found for which [IsRaycastLocationValid\(\)](#) returns [false](#), the method moves up to the parent level and repeats the check.

For components that do not implement the [ICanvasRaycastFilter](#) interface, such as [Text](#), [true](#) is returned. In other words, for [Text](#), [RectTransform](#)'s judgment of inside and outside of [rect](#) is directly used for raycast judgment.

## SetLayoutDirty

---

```
public virtual void SetLayoutDirty();
```

Mark the Layout as dirty.

If a callback has been set by [RegisterDirtyLayoutCallback\(\)](#), it will be called. If the layout is marked as dirty, a Layout rebuild will be performed via [CanvasUpdateRegistry](#). The load during this layout rebuild is measured as **Layout** in the UI area of the **Profiler**.

### [SetMaterialDirty](#)

```
public virtual void SetMaterialDirty();
```

Mark the material as dirty.

If a callback has been set by [RegisterDirtyMaterialCallback\(\)](#), it will be called. If the material is marked as dirty, a Graphic rebuild will be performed via [CanvasUpdateRegistry.PerformUpdate\(\)](#). The load at this time is measured as **Render** in the UI area of the **Profiler**.

### [SetVerticesDirty](#)

```
public virtual void SetVerticesDirty();
```

Mark the vertex as dirty.

If a callback has been set up with [RegisterDirtyMaterialCallback\(\)](#), it will be called. If a vertex is marked as dirty, a Graphic rebuild is performed via [CanvasUpdateRegistry.PerformUpdate\(\)](#) as if the material was dirty. The load at this time is measured as **Render** in the UI area of the Profiler.

### [SetAllDirty](#)

```
public virtual void SetAllDirty();
```

Mark Layout and Material and Vertex as dirty.

Normally, the three methods [SetLayoutDirty\(\)](#), [SetMaterialDirty\(\)](#), and [SetVerticesDirty\(\)](#) are called, but the first two are skipped if the layout and material are known to be unchanged, such as in a spritesheet animation. The first two are skipped. This skipping is done using the [m\\_SkipLayoutUpdate](#) and [m\\_SkipMaterialUpdate](#) variables.

*Defining m\_SkipLayoutUpdate and m\_SkipMaterialUpdate in Graphic*

```
[NonSerialized] protected bool m_SkipLayoutUpdate;  
[NonSerialized] protected bool m_SkipMaterialUpdate;
```

*Implementation inside the sprite property of Image*

```
public Sprite sprite  
{ ...  
...  
set  
{  
    m_SkipLayoutUpdate = m_Sprite.rect.size.Equals(value ? value.rect.size : Vector2.zero)  
    m_SkipMaterialUpdate = m_Sprite.texture == (value ? value.texture : null);  
    m_Sprite = value;  
  
    SetAllDirty();  
    ...  
}  
...  
}
```

Since these are **protected** variables, you can skip the redundant layout and material update process by setting them to **true** in the classes that inherit from [Graphic](#).

## Rebuild

---

```
public virtual void Rebuild(CanvasUpdate update);
```

Rebuild the geometry and materials in the [PreRender](#) stage of the Canvas rebuild.

If the vertex is dirty, it calls [UpdateGeometry\(\)](#), and if the material is dirty, it calls [UpdateMaterial\(\)](#).

## RegisterDirtyLayoutCallback

---

```
public void RegisterDirtyLayoutCallback(UnityAction action);
```

Set up any additional processing that you want to do after [SetLayoutDirty\(\)](#) is called.

*It is used in the [ApplyTextEllipsis](#) component of the sample code in the section "Using [TextGenerator](#) to abbreviate outliers with ellipsis (...)".*

### RegisterDirtyMaterialCallback

---

```
public void RegisterDirtyMaterialCallback(UnityAction action);
```

Set any additional processing you want to do after [SetMaterialDirty\(\)](#) is called (to mark the material as dirty).

### RegisterDirtyVerticesCallback

---

```
public void RegisterDirtyVerticesCallback(UnityAction action);
```

Set up any additional processing you want to do after [SetVerticesDirty\(\)](#) is called (to mark a vertex as dirty).

### UnregisterDirtyLayoutCallback

---

```
public void UnregisterDirtyLayoutCallback(UnityAction action);
```

Cancel the callback registered with [RegisterDirtyLayoutCallback](#).

### UnregisterDirtyMaterialCallback

---

```
public void UnregisterDirtyMaterialCallback(UnityAction action);
```

Cancel the callback registered with [RegisterDirtyMaterialCallback](#).

## UnregisterDirtyVerticesCallback

---

```
public void UnregisterDirtyVerticesCallback(UnityAction action);
```

Cancel the callback registered with [RegisterDirtyVerticesCallback](#).

## SetNativeSize

---

```
public virtual void SetNativeSize();
```

Adjust the size to a pixel-perfect size.

Since the implementation in the [Graphic](#) class is empty, it must be implemented in a child class if necessary.

When the **Set Native Size** button is pressed, the [RectTransform](#) size will be the same as the [Sprite](#) size. If the [Sprite](#)'s [pixelsPerUnit](#) is [200](#) instead of the default [100](#), then the [RectTransform](#)'s size will be half that of the [Sprite](#)'s (or more accurately, the number obtained by dividing the [CanvasScaler](#)'s [referencePixelsPerUnit](#) by the [Sprite](#)'s [pixelsPerUnit](#)). [pixelsPerUnit](#) of the [Sprite](#)).

## Protected methods of the Graphic

In this document, we do not basically explain **protected** methods, but we do explain the Graphic class, since there are many opportunities to create your own class by inheriting from it.

### OnPopulateMesh

```
protected virtual void OnPopulateMesh(VertexHelper vh);
```

Called when a UI element generates a vertex.

If you inherit from the Graphic class and want to draw with your own vertices, you should **override** this method.

If you add a vertex to the **VertexHelper** passed as an argument, the vertex will be drawn later. **OnPopulateMesh()** in practice.

*Implementation of Graphic.OnPopulateMesh()*

```
protected virtual void OnPopulateMesh(VertexHelper vh)
{
    // Get a pixel-perfect Rect
    var r = GetPixelAdjustedRect();

    // create a Vector4 representing the position of the rectangle's vertices
    var v = new Vector4(r.x, r.y, r.x + r.width, r.y + r.height);

    // Use the vertex color set in the member variable
    Color32 color32 = color;

    // delete vertices that have already been set in VertexHelper
    vh.Clear();

    // Add the vertex information of the rectangle. The third argument is UV.
    vh.AddVert(new Vector3(v.x, v.y), color32, new Vector2(0, 0));
    vh.AddVert(new Vector3(v.x, v.w), color32, new Vector2(0, 1));
    vh.AddVert(new Vector3(v.z, v.w), color32, new Vector2(1, 1));
    vh.AddVert(new Vector3(v.z, v.y), color32, new Vector2(1, 0));
```

```
// Set up two triangles using the set vertices  
vh.AddTriangle(0, 1, 2);  
vh.AddTriangle(2, 3, 0);  
}
```

## UpdateGeometry

```
protected virtual void UpdateGeometry();
```

Called in the [PreRender](#) stage of a Canvas rebuild when a vertex is dirty.

Generate vertices with [OnPopulateMesh\(\)](#), generate a mesh using those vertices, and pass the vertex data to be drawn to the [CanvasRenderer](#).

## UpdateMaterial

```
protected virtual void UpdateMaterial();
```

Called in the [PreRender](#) stage of a Canvas rebuild if the material is dirty.

Pass the material and texture you want to draw to the [CanvasRenderer](#).

## The VertexHelper class

```
public class VertexHelper : IDisposable
```

VertexHelper is a helper class that can be used to generate meshes to be passed to the CanvasRenderer. The flow of passing the mesh Graphic wants to draw to the CanvasRenderer is as follows

1. Set a vertex to VertexHelper with [OnPopulateMesh\(\)](#) and specify a triangle from that vertex.
2. Generates a mesh from the triangles set in VertexHelper.
3. Pass the generated mesh to the [CanvasRenderer](#).

VertexHelper is just a convenience class for generating meshes.

## Properties of VertexHelper

### currentIndexCount

```
public int currentIndexCount { get; }
```

Gets the number of indices set in [VertexHelper](#).

A sample is shown in [currentVertCount\(\)](#), described below.

### currentVertCount

Gets the number of vertices set in the [VertexHelper](#).

The following sample code returns [4](#) as [currentVertCount](#) (which is the number of vertices set by [AddVert\(\)](#)) and [6](#) as [currentIndexCount](#) (for two triangles).

```
public class VertexHelperSample1 : Graphic
{
    protected override void OnPopulateMesh(VertexHelper vh)
    {
        var r = GetPixelAdjustedRect();
        var v = new Vector4(r.x, r.y, r.x + r.width, r.y + r.height);
        Color32 color32 = color;

        vh.Clear();

        vh.AddVert(new Vector3(v.x, v.y), color32, new Vector2(0, 0));
        vh.AddVert(new Vector3(v.x, v.w), color32, new Vector2(0, 1));
        vh.AddVert(new Vector3(v.z, v.w), color32, new Vector2(1, 1));
        vh.AddVert(new Vector3(v.z, v.y), color32, new Vector2(1, 0));

        vh.AddTriangle(0, 1, 2);
        vh.AddTriangle(2, 3, 0);

        Debug.Log(vh.currentVertCount); // output 4
        Debug.Log(vh.currentIndexCount); // output 6
    }
}
```

$\left. \begin{array}{c} \\ \end{array} \right\}$

## Public methods of VertexHelper

### Clear

```
public void Clear();
```

Delete the already set vertex/index/color/UV data from [VertexHelper](#).

### AddVert

```
public void AddVert(UIVertex v);
public void AddVert(Vector3 position, Color32 color, Vector4 uv0);
public void AddVert(Vector3 position, Color32 color, Vector4 uv0, Vector4 uv1, Vector3 normal, Vector4 tangent);
public void AddVert(Vector3 position, Color32 color, Vector4 uv0, Vector4 uv1, Vector4 uv2, Vector4 uv3, Vector3 normal, Vector4 tangent);
```

Set one vertex to [VertexHelper](#).

Since this method only adds vertices, it is necessary to specify the vertices and call [AddTriangle\(\)](#) to generate a triangle.

### AddTriangle

```
public void AddTriangle(int idx0, int idx1, int idx2);
```

Set up one triangle in [VertexHelper](#) by specifying three vertices added by [AddVert\(\)](#).

In fact, these three indexes will be added to the list of indexes.

### AddUIVertexStream

```
public void AddUIVertexStream(List<UIVertex> verts, List<int> indices);
```

Pass a list of `UIVertex` and indexes to set the data needed for the mesh.

The sample code to draw a rectangle using `AddUIVertexStream()` instead of `AddVert()` and `AddTriangle()` is as follows.

```
public class VertexHelperSample2 : Graphic
{
    protected override void OnPopulateMesh(VertexHelper vh)
    {
        var r = GetPixelAdjustedRect();
        var v = new Vector4(r.x, r.y, r.x + r.width, r.y + r.height);
        Color32 color32 = color;

        vh.Clear();

        List<UIVertex> verts = new List<UIVertex>();
        List<int> indices = new List<int>();

        verts.Add(new UIVertex()
        {
            position = new Vector3(v.x, v.y),
            color = color32,
            uv0 = new Vector2(0, 0),
        });
        verts.Add(new UIVertex()
        {
            position = new Vector3(v.x, v.w),
            color = color32,
            uv0 = new Vector2(0, 1),
        });
        verts.Add(new UIVertex()
        {
            position = new Vector3(v.z, v.w),
            color = color32,
            uv0 = new Vector2(1, 1),
        });
        verts.Add(new UIVertex()
        {
            position = new Vector3(v.z, v.y),
            color = color32,
            uv0 = new Vector2(1, 0),
        });

        indices.Add(0);
        indices.Add(1);
        indices.Add(2);
        indices.Add(3);

        vh.AddUIVertexStream(verts);
        vh.AddUIVertexIndexStream(indices);
    }
}
```

```
});

indices.Add(0);
indices.Add(1);
indices.Add(2);

indices.Add(2);
indices.Add(3);
indices.Add(0);

vh.AddUIVertexStream(verts, indices);
}

}
```

## AddUIVertexTriangleStream

```
public void AddUIVertexTriangleStream(List<UIVertex> verts);
```

Pass a list of [UIVertices](#) that make up multiple triangles and set the data needed for the mesh.

The list of indices does not have to be passed, but instead the number of vertices passed must be a multiple of three. When drawing a rectangle, the number of vertices passed will be duplicated, which is not efficient.

The following is a sample of using [AddUIVertexStream\(\)](#) instead of [AddVert\(\)](#) and [AddTriangle\(\)](#) to draw a rectangle.

```
public class VertexHelperSample3 : Graphic
{
    protected override void OnPopulateMesh(VertexHelper vh)
    {
        var r = GetPixelAdjustedRect();
        var v = new Vector4(r.x, r.y, r.x + r.width, r.y + r.height);
        Color32 color32 = color;

        vh.Clear();

        List<UIVertex> verts = new List<UIVertex>();
```

```

// Vertex of the first triangle
verts.Add(new UIVertex()
{
    position = new Vector3(v.x, v.y),
    color = color32,
    uv0 = new Vector2(0, 0),
});
verts.Add(new UIVertex()
{
    position = new Vector3(v.x, v.w),
    color = color32,
    uv0 = new Vector2(0, 1),
});
verts.Add(new UIVertex()
{
    position = new Vector3(v.z, v.w),
    color = color32,
    uv0 = new Vector2(1, 1),
});

// Vertex of the second triangle
verts.Add(new UIVertex()
{
    position = new Vector3(v.z, v.w),
    color = color32,
    uv0 = new Vector2(1, 1),
});
verts.Add(new UIVertex()
{
    position = new Vector3(v.z, v.y),
    color = color32,
    uv0 = new Vector2(1, 0),
});
verts.Add(new UIVertex()
{
    position = new Vector3(v.x, v.y),
    color = color32,
    uv0 = new Vector2(0, 0),
});

vh.AddUIVertexTriangleStream(verts);

```

```
}
```

## AddUIVertexQuad

```
public void AddUIVertexQuad(UIVertex[] verts);
```

Pass an array of [UIVertices](#) that make up a single rectangle and set the data required for the mesh.

If you just want to draw a single rectangle, it is easy to use this method. The sample code is shown below.

```
public class VertexHelperSample4 : Graphic
{
    protected override void OnPopulateMesh(VertexHelper vh)
    {
        var r = GetPixelAdjustedRect();
        var v = new Vector4(r.x, r.y, r.x + r.width, r.y + r.height);
        Color32 color32 = color;

        vh.Clear();

        UIVertex[] verts = new UIVertex[4];

        // 4 vertices of the rectangle
        verts[0] = new UIVertex()
        {
            position = new Vector3(v.x, v.y),
            color = color32,
            uv0 = new Vector2(0, 0),
        };

        verts[1] = new UIVertex()
        {
            position = new Vector3(v.z, v.y),
            color = color32,
            uv0 = new Vector2(1, 0),
        };
    }
}
```

```
verts[2] = new UIVertex()
{
    position = new Vector3(v.z, v.w),
    color = color32,
    uv0 = new Vector2(1, 1),
};

verts[3] = new UIVertex()
{
    position = new Vector3(v.x, v.w),
    color = color32,
    uv0 = new Vector2(0, 1),
};

vh.AddUIVertexQuad(verts);
}
}
```

## Dispose

```
public void Dispose();
```

Free the allocated memory.

`VertexHelper` uses the static class `UnityEngine.UI.ListPool` to pool vertices and other objects, so you need to implement the `IDisposable` interface and explicitly free memory in `Dispose()`

## FillMesh

```
public void FillMesh(Mesh mesh);
```

Pack your own data into the mesh given as an argument.

The data that needs to be passed to `CanvasRenderer` for drawing is not `VertexHelper` but `Mesh`. By calling this method, the `VertexHelper` data is packed into a `Mesh`, and the `Mesh` is passed to the `CanvasRenderer`.

The actual usage of `DoMeshGeneration()`, which is called from `UpdateGeometry()` of the `Graphic` class, is easy to understand. `DoMeshGeneration()` generates the vertices required for this UI element, packs them into a mesh, and finally passes them to the `CanvasRenderer` for rendering.

*Graphic's UpdateGeometry() and DoMeshGeneration()*

```
/// <summary>
/// Update the geometry of this UI element and pass it to the CanvasRenderer
/// </summary>
protected virtual void UpdateGeometry()
{
    // true by default, but false for Image/RawImage/Text
    if (useLegacyMeshGeneration)
    {
        DoLegacyMeshGeneration();
    }
    else
    {
        // So basically, just look at this
        DoMeshGeneration();
    }
}

private void DoMeshGeneration()
{
    // generate a vertex only if it is in a state that should be drawn
    if (rectTransform != null && rectTransform.rect.width >= 0 && rectTransform.rect.height >= 0)
    {
        OnPopulateMesh(s_VertexHelper);
    }
    else
    {
        // If it's an abnormal state, clear the vertex and don't draw anything
        s_VertexHelper.Clear();
    }

    // Borrow the list of components from the pool
    // (Borrowing from the pool is more efficient, since creating a new list is heavy)
    var components = ListPool<Component>.Get();
```

```

// Find components that implement IMeshModifier (e.g. Shadow) if they are attached
GetComponents(typeof(IMeshModifier), components);

// Apply shadow, etc.
for (var i = 0; i < components. Count; i++)
    ((IMeshModifier)components[i]).ModifyMesh(s_VertexHelper);

// Release the borrowed list.
ListPool<Component>.Release(components);

// The generated vertices are stored in VertexHelper, so we fill the mesh with them
s_VertexHelper.FillMesh(workerMesh);

FillMesh(workerMesh); // Pass the mesh we want to draw to CanvasRenderer. canvasRenderer
.
SetMesh(workerMesh);
}

```

## GetUIVertexStream

---

```
public void GetUIVertexStream(List<UIVertex> stream);
```

Get the [UIVertex](#) that has already been set.

The [Shadow](#) component [ModifyMesh\(\)](#) has a very straightforward use, so let's take a look at the code.

```

public class Shadow : BaseMeshEffect
{
    ...
    ...
    public override void ModifyMesh(VertexHelper vh)
    {
        if (!IsActive())
            return;

        // Borrow a list of UIVertexes from the pool (new would be too heavy)
        var output = ListPool<UIVertex>.Get();
    }
}
```

```

// get the data already set in the VertexHelper
vh.GetUIVertexStream(output);

// add the vertices for the shadow
ApplyShadow(output, effectColor, 0, output.Count, effectDistance.x, effectDistance.y);

// erase the data already set in VertexHelper
vh.Clear();

// set VertexHelper to the data with the shadows added
vh.AddUIVertexTriangleStream(output);

// release the borrowed list
ListPool<UIVertex>.Release(output);
}
...

```

## PopulateUIVertex

---

```
public void PopulateUIVertex(ref UIVertex vertex, int i);
```

Copy the already set i-th **UIVertex** to the vertex and return it.

It is not very useful, but the example of **PositionAsUV1** may be helpful.

```

public class PositionAsUV1 : BaseMeshEffect
{
    ...
    public override void ModifyMesh(VertexHelper vh)
    {
        // a temporary instance to hold the vertex data
        UIVertex vert = new UIVertex();
        for (int i = 0; i < vh.currentVertCount; i++)
        {
            // get the i-th vertex
            vh.PopulateUIVertex(ref vert, i);

            // Populate UV1 with the vertex positions
        }
    }
}
```

```
    vert.uv1 = new Vector2(vert.position.x, vert.position.y);

    // set the i-th vertex
    vh.SetUIVertex(vert, i);
}

}
```

## SetUIVertex

```
public void SetUIVertex(UIVertex vertex, int i);
```

Set the specified [UIVertex](#) to the i-th [UIVertex](#).

Please refer to the example of [PositionAsUV1](#) above.

## MaskableGraphic component

```
public abstract class MaskableGraphic : Graphic, IClippable, IMaskable, IMaterialModifier;
```

The [MaskableGraphic](#) component is a [Graphic](#) component that can be the target of a mask; [Image](#), [RawImage](#), [Text](#), and [TextMesh Pro's TextMeshProUGUI](#) inherit from [MaskableGraphic](#).

The [IClippable](#) interface implemented by this class indicates the object to be clipped by the object that implements the [IClipper](#) interface. The only object that implements the [IClipper](#) interface is [RectMask2D](#), that is, [MaskableGraphic](#) is subject to clipping by [RectMask2D](#).

The [IMaskable](#) interface implemented by this class indicates that it is an object to be masked by a stencil mask.

And the [IMaterialModifier](#) is an interface to change the material used for rendering without affecting the original material. This is necessary for stencil masks with [Mask](#).

## MaskableGraphic properties

### isMaskingGraphic

```
public bool isMaskingGraphic { get; set; }
```

Gets/Sets whether the [Mask](#) component attached to the same [GameObject](#) is enabled or not.

It was introduced in Unity 2020.2. With the implementation of this method, it is no longer necessary to get the [mask](#) component with [GetCompolement<Mask>\(\)](#) and check for [enabled](#).

If you want to change the value of this property, be sure to call [MaskUtilities.NotifyStencilStateChanged\(this\)](#) when you change it.

### maskable

```
public bool maskable { get; set; }
```

Gets/Sets whether this component is maskable or not.

This setting affects both the stencil mask ([Mask](#)) and the clipping ([RectMask2D](#)). Changing this property will cause the material to become dirty, resulting in a Graphic rebuild.

### onCullStateChanged

```
public MaskableGraphic.CullStateChangedEvent onCullStateChanged { get; set; }
```

Get/Sets the callback when the culling status changes (i.e., when it is either all invisible or partially visible due to [RectMask2D](#)). The sample code is shown below.

```
using UnityEngine;
using UnityEngine.UI;

// Callback received when RectMask2D makes all/partially visible.
```

```

[RequireComponent(typeof(MaskableGraphic))]
public class OnCullStateChangedSample : MonoBehaviour
{
    private MaskableGraphic maskableGraphic;

    private void Start()
    {
        maskableGraphic = GetComponent<MaskableGraphic>();
        maskableGraphic.onCullStateChanged.AddListener(OnCullStateChanged);
    }

    private void OnDestroy()
    {
        maskableGraphic.onCullStateChanged.removeListener(OnCullStateChanged);
    }

    public void OnCullStateChanged(bool culled)
    {
        if (culled)
        {
            Debug.LogFormat("All invisible due to RectMask2D");
        }
        else
        {
            Debug.LogFormat("Even part of it is now visible");
        }
    }
}

```

This will be useful if you want to stop some process when you are clipped and cannot see everything.

## MaskableGraphic's public method

### cull

```
public virtual void Cull(Rect clipRect, bool validRect);
```

If `validRect` is `false`, it is clipped.

This method is an implementation of the [IClippable](#) interface.

### GetModifiedMaterial

```
public virtual Material GetModifiedMaterial(Material baseMaterial);
```

Gets the material that the [CanvasRenderer](#) will actually use for rendering.

This method is an implementation of the [IMaterialModifier](#) interface.

This method is used to change the material used for rendering without affecting the original material. The resulting material is returned after the original material has been affected by the mask stencil.

### RecalculateClipping

```
public virtual void RecalculateClipping();
```

Recalculate the clipping area.

This method is an implementation of the [IClippable](#) interface.

### RecalculateMasking

```
public virtual void RecalculateMasking();
```

Recalculate the stencil mask.

This method is a method that implements the [IMaskable](#) interface.

Calling this method will make the material dirty and cause a Graphic rebuild.

## SetClipRect

---

```
public virtual void SetClipRect(Rect clipRect, bool validRect);
```

Set the clipping area.

This method is an implementation of the [IClippable](#) interface.

If `validRect` is `true`, the `clipRect` is set as the clipping region; if `validRect` is `false`, the clipping itself is disabled.

## SetClipSoftness

---

```
public virtual void SetClipSoftness(Vector2 clipSoftness);
```

Set the range of the soft mask (edge blur function) introduced in Unity 2020.1.

`canvasRenderer.clippingSoftness` is set to this value, which is eventually passed to the UI shaders `_MaskSoftnessX` and `_MaskSoftnessY`.

## Create a transparent layer to absorb touch events.

We often find ourselves in a situation where we do not want touch events to occur in some areas. A simple solution would be to put a transparent [RawImage](#) or [Image](#) (with `raycastTarget` set to `true`) on top to absorb touch events, but this would result in unnecessary drawing.

In order to solve this problem, there is a way to extend [MaskableGraphic](#), clearing the vertex data and keeping only the touch judgment alive. The code is shown below.

```
using UnityEngine.UI;
#if UNITY_EDITOR
using UnityEditor;
#endif

// Transparent layer for absorbing touch
public class TransparentLayer : MaskableGraphic
{
    protected override void OnPopulateMesh(VertexHelper toFill)
    {
        toFill.Clear();
    }
}

// Do not display anything in the Inspector
#if UNITY_EDITOR
[CustomEditor(typeof(TransparentLayer))]
public class TransparentLayerEditor : Editor
{
    public override void OnInspectorGUI()
    {
    }
}
#endif
```

You can adjust the position and size of [RectTransform](#) of this [TransparentLayer](#) component to cover the area where you don't want the touch detection to occur.

## Chapter 5 Images and Effects

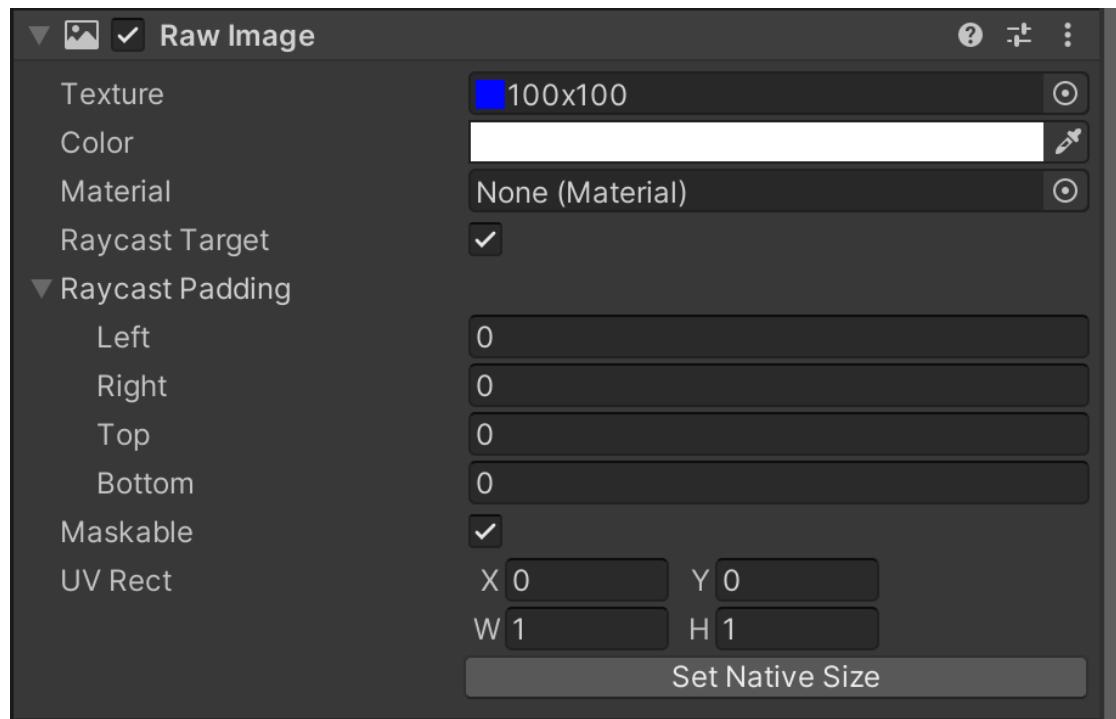
### Using RawImage vs. Image

There are two types of components for displaying images: [RawImage](#) and [Image](#). This section describes the characteristics of these and how to use them.

- [RawImage](#) Features
  - It is a prerequisite component for displaying [textures](#).
  - It is possible to change the numerical values of UV coordinates from **Inspector** and scripts.
  - Suitable for displaying unatlashed single pictures and [RenderTexture](#).
  - The official documentation says that there will be more extra draw calls, but that's not true (they are batched properly).
- [Image](#) Features
  - This component is a prerequisite for displaying [Sprite](#).
  - The UV coordinates of [Sprite](#) will be used (UV coordinate changes are not expected).
  - Circular and other displays are also possible.
  - It is suitable for displaying atlasized images.

`Create()`, which is called to create a [Sprite](#), is CPU- and memory-intensive. `Create()`, which is called to create the [Sprite](#), is CPU- and memory-intensive. Therefore, [Sprite.Create\(\)](#) calls at runtime should be avoided as much as possible. `Create()` calls at runtime should be avoided as much as possible. For example, if a large number of [Images](#) are generated at runtime, this will result in a large number of [Sprite](#). In this case, it is better to use [RawImage](#).

## RawImage component



```
[RequireComponent(typeof(CanvasRenderer))]
[AddComponentMenu("UI/Raw Image", 12)]
public class RawImage : MaskableGraphic, ICanvasElement, IClippable, IMaskable, IMaterialModifier
```

RawImage is a component for displaying [textures](#). It is suitable for displaying a simple single image or a [RenderTexture](#).

## Properties of RawImage

### mainTexture

```
public override Texture mainTexture { get; }
```

Get the main texture to be used in this RawImage.

If a texture is set, return that texture; if not, return the [mainTexture](#) of the [material](#); if not, return the default white texture ([Texture2D.whiteTexture](#)).

### texture

```
public Texture texture { get; set; }
```

Gets/Sets the main texture to be used in this [RawImage](#).

The difference with [mainTexture](#) is that null is returned if nothing is set. Also, if a different texture is set, [SetVerticesDirty\(\)](#) and [SetMaterialDirty\(\)](#) will be called to cause a Graphic rebuild.

### uvRect

```
public Rect uvRect { get; set; }
```

Gets/Sets UV coordinates.

By default, [\(x, y, width, height\)](#) is [\(0, 0, 1, 1\)](#). If a different value is set, [SetVerticesDirty\(\)](#) will be called and a Graphic rebuild will occur.

## Public methods of RawImage

### SetNativeSize

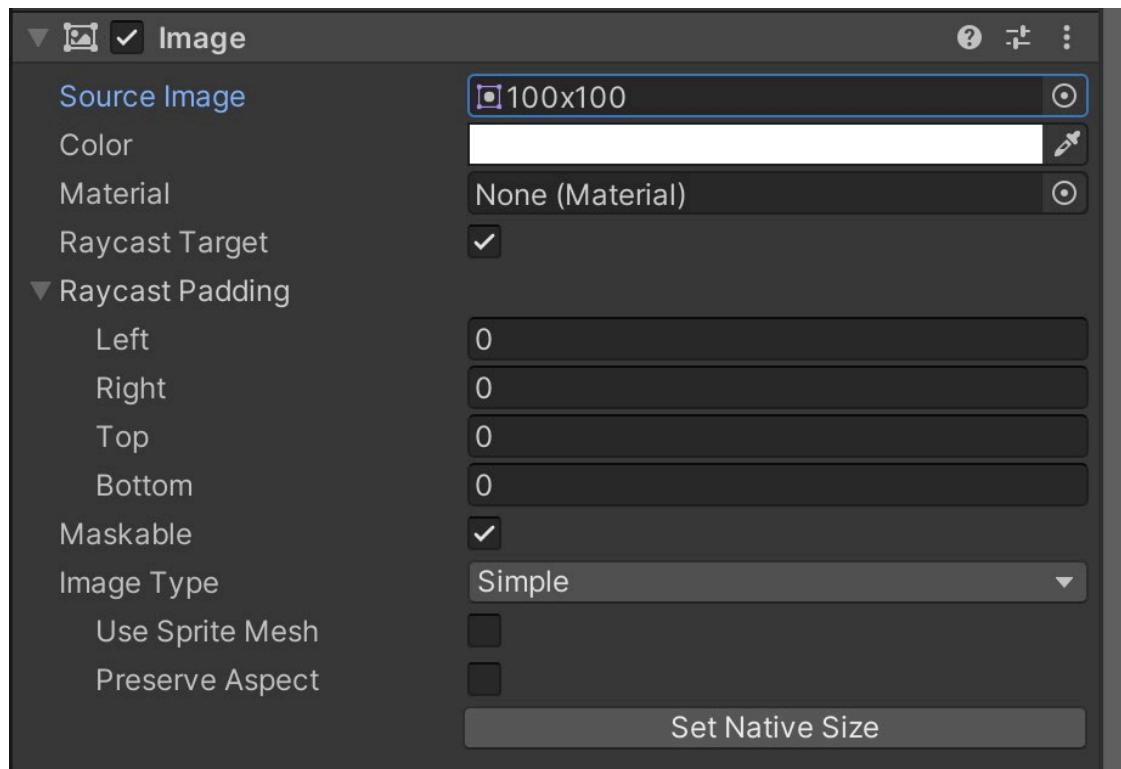
```
public override void SetNativeSize();
```

Adjust the size to a pixel-perfect size.

Since the implementation in the [Graphic](#) class was empty, the actual implementation is done here.

The result of multiplying the width and height of the [mainTexture](#) by the width and height of the [uvRect](#), respectively, is assigned to [rectTransform.sizeDelta](#) as an [int](#) value.

## Image component



```
[RequireComponent(typeof(CanvasRenderer))]
[AddComponentMenu("UI/Image", 11)]
public class Image : MaskableGraphic, ISerializationCallbackReceiver, ILayoutElement, ICanvasRaycastFilter
```

As mentioned above, **Image** is a prerequisite component for displaying a **Sprite**. It is suitable for displaying atlasized images.

## Static variables of Image

s\_ETC1DefaultUI

**protected static** Material s\_ETC1DefaultUI;

Returns the default material dedicated to the ETC1 format used in Android.

[GetETC1SupportedCanvasMaterial\(\)](#), which is actually a cache of Canvas.

## Properties of Image

### sprite

```
public Sprite sprite { get; set; }
```

Gets/Sets the sprite to render this [Image](#).

If a different [Sprite](#) is set than the previous one, [SetAllDirty\(\)](#) will be called and a Graphic rebuild will occur.

### overrideSprite

```
public Sprite overrideSprite { get; set; }
```

Gets/Sets a sprite to override the one used for rendering.

If the [overrideSprite](#) is [null](#), then a normal [sprite](#) will be used for rendering; otherwise, the [overrideSprite](#) will be used for rendering. The [Sprite](#) used for rendering is called the "active [Sprite](#)".

The [overrideSprite](#) is used when you want to change the sprite that is temporarily displayed by a [Button](#).

*Sample code for overrideSprite*

```
[RequireComponent(typeof(Image))]
public class OverrideSpriteSample : MonoBehaviour
{
    public Sprite baseSprite;
    public Sprite overrideSprite;

    private Image image;

    public void Start()
    {
        image = GetComponent<Image>();
```

```

        image.sprite = baseSprite;
    }

// Temporarily change the sprite of image
public void DoOverrideSprite()
{
    image.overrideSprite = overrideSprite;
}

// Undo the temporarily changed sprite
public void UndoOverrideSprite()
{
    image.overrideSprite = null;
}

public void Update()
{
    // set/unset the override sprite every 30 frames
    if (Time.frameCount % 60 == 0)
    {
        DoOverrideSprite();
    }
    else if (Time.frameCount % 60 == 30)
    {
        UndoOverrideSprite();
    }
}
}

```

If you want to temporarily replace a `sprite` directly without using `overrideSprite`, the external component needs to keep the original `sprite`. One of the purposes of `overrideSprite` is to avoid this hassle (and the possibility of extra memory allocation).

However, since Unity 2019.1, optimizations have been implemented such that when a `sprite` is changed, the layout rebuild is skipped if the size has not changed, and the material update process is skipped if the texture has not changed. On the other hand, there is no such processing in `overrideSprite`, so if you are replacing sprites of the same size or within the same texture, it is actually better to change the `sprite` directly.

## hasBorder

---

```
public bool hasBorder { get; }
```

Returns `true` if the size of the active `Sprite's border` is greater than `0`.

As we will see later, if the `type` of this `Image` component is `Sliced` or `Tiled`, this property must be `true` to get the desired result.

## type

---

```
public Image.Type type { get; set; }
```

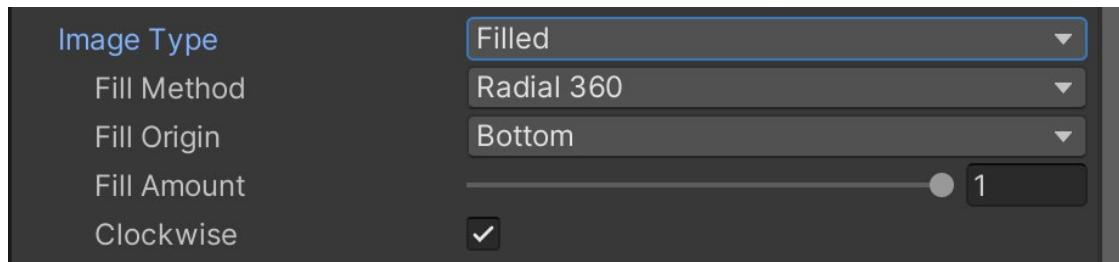
Gets/Sets the display method of the image. The following four types are defined.

- **Simple:** Displays the entire image with 4 vertices. The display size is determined by the size of `RectTransform`.
- **Sliced:** Display the image in 9 slices. 9 slices allow you to stretch only the center part of the image while keeping the pixels in the corners. This may be useful for dialog backgrounds and button images. In order to set the `border`, start the **Sprite Editor** from the *Import Settings* of the texture and set it. The number of rectangles used for drawing is 9, and the number of vertices, including duplicates, is 36.
- **Tiled:** Repeats the center of the image, whereas `Sliced` stretches the center. As with `Sliced`, the `border` of `Image.sprite` is used to determine the center. However, the more repetitions there are, the larger the number of rectangles (and vertices) that are divided in the mesh. Since the number of rectangles in the mesh is limited to `16250` (actually, the number of vertices is checked to make sure it does not exceed `65000`), it is not possible to repeat more than that. (In fact, the number of vertices is checked to make sure it does not exceed `65000`.) If you try to iterate more than that, an error log will be printed. If you do not need the four corners, you can set the `wrapMode` of a `Sprite texture` that does not have `borders` and is not packed to `TextureWrapMode`. Repeat is possible without adding geometry.
- **Filled:** Displays only a portion of the image. There are five types of display methods: `Horizontal`, `Vertical`, `Radial90`, `Radial180`, and `Radial360`, the details of which are explained below. In `Horizontal` and `Vertical`, only a part of the mesh is drawn by moving the vertices, and in `Radial90`, `Radial180`, and `Radial360`, only a part of the mesh is drawn by dividing the mesh into multiple rectangles and adjusting the vertex positions. In `Radial90`, `Radial180`, and `Radial360`, the mesh is divided into multiple rectangles and the vertex positions are adjusted to draw only a portion of the mesh.

## fillMethod

```
public Image.FillMethod fillMethod { get; set; }
```

Gets/Sets the display method when [Type](#) is Filled.



In both cases, the direction of filling depends on the [FillOrigin](#) (and [Clockwise](#)) values.

- If [fillMethod](#) is [Horizontal](#), the display area will be filled horizontally.
  - If [FillOrigin](#) is [Left](#), the closer [FillAmount](#) is to 1, the more of the right side will be displayed.
  - If [FillOrigin](#) is [Right](#), the closer [FillAmount](#) is to 1, the more of the right side will be displayed.
- If [fillMethod](#) is [Vertical](#), the display area will be filled vertically.
  - If [FillOrigin](#) is [Bottom](#), the closer [FillAmount](#) is to 1, the more the upper side will be displayed.
  - If [FillOrigin](#) is [Top](#), the closer [FillAmount](#) is to 1, the more the lower side is displayed.

- If `fillMethod` is `Radial90`, the display area will be filled with a 90-degree fan-shaped area.
  - If `FillOrigin` is `BottomLeft`, the closer `FillAmount` is to `1`, the more it will be displayed from bottom-right to top-left.
  - If `FillOrigin` is `TopLeft`, the closer `FillAmount` is to `1`, the more it will be displayed from lower left to upper right.
  - If `FillOrigin` is `TopRight`, the closer `FillAmount` is to `1`, the more it will be displayed from top left to bottom right.
  - If `FillOrigin` is `BottomRight`, the closer `FillAmount` is to `1`, the more it will be displayed from top-right to bottom-left.
- If `fillMethod` is `Radial180`, the display area will be filled with a 180-degree fan-shaped area.
  - If `FillOrigin` is `Bottom`
    - If `Clockwise` is `true`, the closer `FillAmount` is to `1`, the more it will be displayed from lower right to lower left, centered at the bottom center.
    - If `Clockwise` is `false`, the closer `FillAmount` is to `1`, the more it will be displayed from lower left to lower right, centered at the bottom center.
  - If `FillOrigin` is `Left`
    - If `Clockwise` is `true`, the closer `FillAmount` is to `1`, the more the image will be displayed from the lower left to the upper left, centered on the left center.
    - If `Clockwise` is `false`, the closer `FillAmount` is to `1`, the more it will be displayed from top-left to bottom-left, centered on the left center.
  - If `FillOrigin` is `Top`
    - If `Clockwise` is `true`, the closer the `FillAmount` is to `1`, the more it will be displayed from top left to top right, centered at the top center.
    - If `Clockwise` is `false`, the closer the `FillAmount` is to `1`, the more it will be displayed from top right to top left, centered on the top center.
  - If `FillOrigin` is `Right`
    - If `Clockwise` is `true`, the closer the `FillAmount` is to `1`, the more it will be displayed from the upper right to the lower right, centered on the right center.
    - If `Clockwise` is `false`, the closer the `FillAmount` is to `1`, the more it will be displayed from the bottom right to the top right, centered at the center right.

- If `fillMethod` is `Radial360`, the display area will be filled with a 360-degree fan-shaped area.
  - If `Clockwise` is `true`, the closer `FillAmount` is to `1`, the more the display area will increase clockwise from the bottom center.
  - If `Clockwise` is `false`, the closer `FillAmount` is to `1`, the more the display area will increase counterclockwise from the bottom center.

## fillAmount

---

```
public float fillAmount { get; set; }
```

Gets/Sets the display amount when `type` is set to `Image.Type.Filled`.

If it's `0`, it won't be shown, and if it's `1`, all of them will be shown.

## fillCenter

---

```
public bool fillCenter { get; set; }
```

Gets/Sets whether or not to draw the middle part of the image when it is `Tiled` or `Sliced`.

To enable this, you need to set a border on the `sprite` in the **Sprite Editor**, etc. If you use `fillCenter` properly, you can save the fill rate.

## fillClockwise

---

```
public bool fillClockwise { get; set; }
```

Gets/Sets whether the display direction should be clockwise or not when `type` is `Type.Filled` and `Image.fillMethod` is `Radial90`, `Radial180`, or `Radial360`.

## fillOrigin

---

```
public int fillOrigin { get; set; }
```

Gets/Sets the start position of the display direction when `type` is `Type.Filled`.

#### useSpriteMesh

```
public bool useSpriteMesh { get; set; }
```

Gets/Sets whether to draw using the mesh generated by the `TextureImporter` or a simple rectangular mesh, valid only if `type` is `Simple`.

The default value is `false`, which means that a simple rectangular mesh will be used. The default value is `false`, which means that a simple rectangular mesh will be used. If this property is set to `true`, the mesh will be drawn using the mesh generated without transparent parts if the `MeshType` is set to `Tight` in the `TextureImporter`. If the `MeshType` is set to `FullRect` in the `TextureImporter`, the generated mesh will always be a simple rectangle, and will be drawn with a simple rectangle regardless of this property.

#### preserveAspect

```
public bool preserveAspect { get; set; }
```

Gets/Sets whether the aspect ratio of the `sprite` should be used as is, and is only available when `type` is `Simple`.

The default value is `false`, which means that the image will be stretched according to the size of `RectTransform`. If this property is set to `true`, the display size will be determined based on the smaller of the `width` and `height` of the `RectTransform`.

#### flexibleHeight

```
public virtual float flexibleHeight { get; }
```

Get the flexible height used for Auto Layout.

Always returns `-1`. The details are explained in *Chapter 9 Auto Layout*.

## flexibleWidth

---

```
public virtual float flexibleWidth { get; }
```

Get the flexible width used for Auto Layout.

Always returns -1. The details are explained in *Chapter 9 Auto Layout*.

## layoutPriority

---

```
public virtual int layoutPriority { get; }
```

Get the priority level used for Auto Layout.

Always returns 0. The details are explained in *Chapter 9 Auto Layout*.

## mainTexture

---

```
public override Texture mainTexture { get; }
```

Gets the texture to be used for rendering this [Image](#).

If the active [Sprite](#) is not [null](#), it returns the [texture](#) of that [Sprite](#); if the [Sprite](#) is [null](#), it returns the [mainTexture](#) of the [material](#). If neither [Sprite](#) nor [Material](#) is set, the default white texture ([Texture2D.whiteTexture](#)) is returned.

## material

---

```
public override Material material { get; set; }
```

Gets/Sets the texture to be used for rendering this [Image](#).

If no material is set, [defaultMaterial](#) (or [defaultETC1GraphicMaterial](#) if [Sprite](#) for ETC1 is used) will be returned.

## minHeight

---

```
public virtual float minHeight { get; }
```

Get the minimum height to be used for Auto Layout.

Always returns [0](#). The details are explained in *Chapter 9 Auto Layout*.

## minWidth

---

```
public virtual float minWidth { get; }
```

Get the minimum width to be used for Auto Layout.

Always returns [0](#). The details are explained in *Chapter 9 Auto Layout*.

## pixelsPerUnit

---

```
public float pixelsPerUnit { get; }
```

Get the number of pixels per unit.

The value obtained by dividing the [pixelsPerUnit](#) of the currently active [Sprite](#) (default value is [100](#)) by the [referencePixelsPerUnit](#) of [Canvas](#) will be returned. Normally, [1](#) is returned.

It is used to calculate the size of [RectTransform](#) when [SetNativeSize\(\)](#) is pressed, and also used to calculate [preferredWidth](#) and [preferredWidth](#).

## pixelsPerUnitMultiplier

---

```
public float pixelsPerUnitMultiplier { get; set; }
```

Get the factor to multiply [pixelsPerUnit](#) by.

If `type` is `Sliced` or `Tiled`, the size of the perimeter of the slice is `1.0f / pixelsPerUnitMultiplier`, and if `Tiled`, the number of iterations is multiplied by `pixelsPerUnitMultiplier`. The default value is `1`, and the minimum value is `0.01f`.

### preferredHeight

```
public virtual float preferredHeight { get; }
```

If an active `Sprite` exists, its height (divided by `pixelsPerUnit`) is returned.

If `type` is `Sliced` or `Tiled`, the minimum height (divided by `pixelsPerUnit`) is returned. If there is no active `Sprite`, `0` is returned. Details are explained in *Chapter 9 Auto Layout*.

### preferredWidth

```
public virtual float preferredWidth { get; }
```

If an active `Sprite` exists, its width (divided by `pixelsPerUnit`) is returned.

If `type` is `Sliced` or `Tiled`, the minimum width (divided by `pixelsPerUnit`) is returned. If there is no active `Sprite`, `0` is returned. Details are explained in *Chapter 9 Auto Layout*.

### AlphaHitTestMinimumThreshold

```
public float alphaHitTestMinimumThreshold { get; set; }
```

Gets/Sets the minimum alpha value required for a raycast hit.

The default value is `0`, which means that transparent areas will also be raycast hit.

If this property is set to `1`, pixels with an alpha value of `0` will not be raycast hit. The alpha value checked here is the `Sprite`'s alpha value, and the alpha value of the `color` property will be ignored.

Note that to set this value to a non-zero value, you need to enable `Read/Write enabled` in the `Texture's Import Settings` and disable atlasing. `Read/Write enabled` doubles the amount of texture memory used, which may be difficult to use in real products. The solution to this problem is

described in the section "*Preventing Images from Responding to Touches in Transparent Areas*" in this chapter.

## defaultETC1GraphicMaterial

---

```
public static Material defaultETC1GraphicMaterial { get; }
```

Returns the default material dedicated to the ETC1 format used in Android.

## Public methods of Image

### CalculateLayoutInputHorizontal

```
public virtual void CalculateLayoutInputHorizontal();
```

This method is an implementation of [CalculateLayoutInputHorizontal\(\)](#) of the [ILayoutElement](#) interface, but it is empty.

After this method is called, the properties of the horizontal input of the layout will return the latest values. Also, at the time this method is called, the child always has (is supposed to have) the latest horizontal input of the layout.

### CalculateLayoutInputVertical

```
public virtual void CalculateLayoutInputVertical();
```

This method is an implementation of [CalculateLayoutInputVertical\(\)](#) of the [ILayoutElement](#) interface, but it is empty.

After this method is called, the property of the vertical input of the layout will return the latest value. Also, at the time this method is called, the child should always have the latest vertical input of the layout.

### DisableSpriteOptimizations

```
public void DisableSpriteOptimizations();
```

Turn off the flag to skip unnecessary Layout rebuilds and material updates.

This method was introduced for performance optimization in Unity 2019.1, but we don't usually call it.

## IsRaycastLocationValid

```
public virtual bool IsRaycastLocationValid(Vector2 screenPoint, Camera eventCamera);
```

Make a raycast decision.

Returns `true` if the alpha value of the image at the position of the `screenPoint` given in the argument is greater than or equal to `alphaHitTestMinimumThreshold`.

As mentioned above, if you set `alphaHitTestMinimumThreshold` to a value greater than `0`, enable `Read/Write enabled` in the `Texture` import settings, and disable atlasing, `Texture.GetPixelBilinear()` will be called when judging by alpha.

The actual flow of the ray casting decision process is as follows.

1. Returns `true` if `alphaHitTestMinimumThreshold` is less than or equal to `0`.
2. If `alphaHitTestMinimumThreshold` is greater than `1`, `false` is returned.
3. If the active `Sprite` is `null`, `true` is returned.
4. Returns `false` if the `screenPoint` is not in the plane of `RectTransform` in world space (it does not determine if it is in the `RectTransform` rectangle).
5. `GetPixelBilinear()` of the active `Sprite` to obtain the alpha value of the image at the `screenPoint` position. `GetPixelBilinear()` of the active `Sprite` to get the alpha value of the image at the position of the `screenPoint`. If the alpha value is greater than or equal to `alphaHitTestMinimumThreshold`, `true` is returned; otherwise, `false` is returned.
6. `GetPixelBilinear()` throws an exception if the texture's pixels cannot be read, or if the active `Sprite` is packed, and this method returns `true`.

## OnAfterDeserialize

```
public virtual void OnAfterDeserialize();
```

Called before this object is deserialized by Unity.

This method is an implementation of the `ISerializationCallbackReceiver` interface.

If `fillOrigin` is not appropriate, change it to `0`, and change `fillAmount` to be in the range of `0` to `1`. Note that Unity's serialization process runs in a separate thread from the main thread, so be careful when implementing this callback not to destroy data by rewriting it from multiple threads.

## OnBeforeSerialize

---

```
public virtual void OnBeforeSerialize();
```

It is called after this object has been serialized by Unity.

This method is an implementation of the [ISerializationCallbackReceiver](#) interface.

The implementation in the [Image](#) component is empty.

## SetNativeSize

---

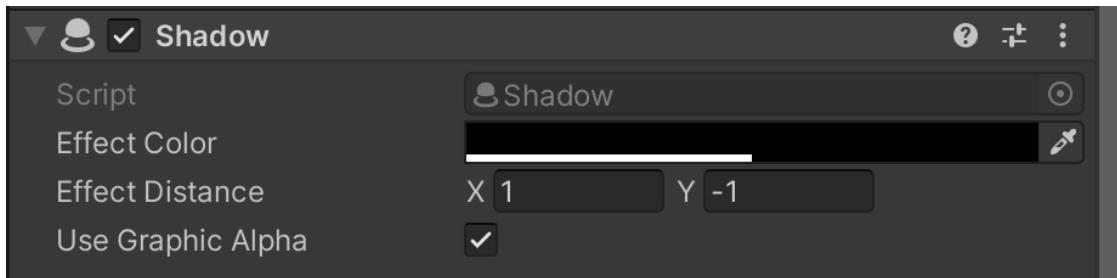
```
public override void SetNativeSize();
```

Adjust the [RectTransform](#) size to the size of the active [Sprite](#) so that it is pixel perfect.

The **Set Native Size** button appears in the **Inspector** only when [type](#) is [Simple](#), and this method is called when the button is pressed.

Note, however, that if you call this method when the [anchor](#) is **stretch** (i.e., [anchorMin = \(0, 0\)](#), [anchorMax = \(1, 1\)](#)), the **stretch** will be removed and the [anchor](#) will be based on the lower left corner ([anchorMin = \(0, 0\)](#), [anchorMax = \(0, 0\)](#)). ([anchorMin = \(0, 0\)](#), [anchorMax = \(0, 0\)](#)).

## Shadow component



```
[AddComponentMenu("UI/Effects/Shadow", 14)]  
public class Shadow : BaseMeshEffect
```

The [Shadow](#) component is used to add shadows to [Graphic](#) components such as [Image](#) and [Text](#), and inherits from the [BaseMeshEffect](#) class.

Drawing a shadow is done in the following way.

1. [ApplyShadow\(\)](#) is called at the time of the Graphic rebuild when the geometry is dirty.
2. Duplicate the vertices of the original [Graphic](#) from [VertexHelper](#).
3. Add the duplicated vertex to the back of the [VertexHelper](#) buffer.
4. Change the color and position of the vertices in the front half of the buffer as the shadow vertices. This will cause the shadow vertices to be drawn first.
5. Generate mesh vertices from [VertexHelper](#) and pass them to [CanvasRenderer](#).

In this way, a single mesh, including shadows, is rendered efficiently.

## Properties of Shadow

### effectColor

```
public Color effectColor { get; set; }
```

Gets/Sets the color of the shadow.

Multiply the original [Graphic](#) color by this color to get the shadow color. The default value is [\(0, 0, 0, 0.5f\)](#), which makes the shadow translucent black, but if you set it to [\(1, 1, 1, 0.5f\)](#), you will see that the shadow becomes translucent while keeping the original texture color.

### effectDistance

```
public Vector2 effectDistance { get; set; }
```

Gets/Sets the distance (in pixels) between the original [Graphic](#) and the shadow.

The higher the value, the more the shadow will be displayed in the upper right corner. The default value is [\(1, -1\)](#), so the shadow is displayed in the lower right corner.

### useGraphicAlpha

```
public bool useGraphicAlpha { get; set; }
```

Gets/Sets whether or not to use the original [Graphic](#)'s alpha value.

The default value is [true](#).

If this property is [true](#), the alpha value is the original [Graphic](#)'s (vertex color's) alpha value multiplied by the [effectColor](#)'s alpha value.

## Public Methods of Shadow

### ModifyMesh

```
public override void ModifyMesh(VertexHelper vh);
```

Called when rebuilding a mesh, for example via Graphic.Rebuild().

In the case of Shadow, ApplyShadow() is called from here. The actual code is shown below.

```
public class Shadow : BaseMeshEffect
{
    ...
    ...
    public override void ModifyMesh(VertexHelper vh)
    {
        if (!IsActive())
            return;

        // Borrow a list of UIVertexes from the pool (new would be too heavy)
        var output = ListPool<UIVertex>.Get();

        // Get the data already set in the VertexHelper
        vh.GetUIVertexStream(output);

        // Add the vertices for the shadow
        ApplyShadow(output, effectColor, 0, output.Count, effectDistance.x, effectDistance.y);

        // Ease the data already set in VertexHelper
        vh.Clear();

        // set VertexHelper to the data with the shadows added
        vh.AddUIVertexTriangleStream(output);

        // Release the borrowed list
        ListPool<UIVertex>.Release(output);
    }
    ...
}
```

## Protected method of Shadow

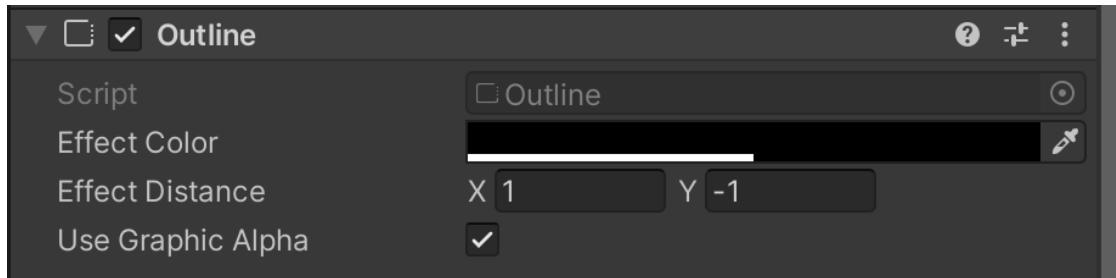
### ApplyShadow

```
protected void ApplyShadow(List<UIVertex> verts, Color32 color, int start, int end, float x, float y);
```

The process of inserting shadow vertices into the [VertexHelper](#) of the original mesh is performed.

The actual process is as already explained.

## Outline Components



```
[AddComponentMenu("UI/Effects/Outline", 15)]
public class Outline : Shadow, IMeshModifier
```

The [Outline](#) component is used to add outlines to [Graphic](#) components such as [Image](#) and [Text](#).

To put it simply, the outline is drawn by drawing [Shadows](#) up, down, left, and right. In other words, by specifying [\(x, y\)](#), [\(-x, -y\)](#), [\(-x, y\)](#), and [\(x, -y\)](#) as the [effectDistance](#) of the [Shadow](#), four shadows are drawn to form the outline.

The actual [Outline](#) code is shown below.

```
/// <summary>
/// Add an outline to the Graphic using the IVertexModifier feature
/// </summary>
public class Outline : Shadow
{
    protected Outline()
    {}

    public override void ModifyMesh(VertexHelper vh)
    {
        if (!IsActive())
            return;

        var verts = ListPool<UIVertex>.Get();
        vh.GetUIVertexStream(verts);

        var neededCpacity = verts.Count * 5;
```

```

if (verts. Capacity < neededCapacity)
    Capacity = neededCapacity;

    var start = 0;
    var end = verts.Count;
    ApplyShadowZeroAlloc(verts, effectColor, start, verts.Count, effectDistance.x, effectDistance.y);

    start = end;
    end = verts.Count;;
    ApplyShadowZeroAlloc(verts, effectColor, start, verts.Count, effectDistance.x, -effectDistance.y);

    start = end;
    end = verts.Count;;
    ApplyShadowZeroAlloc(verts, effectColor, start, verts.Count, -effectDistance.x, effectDistance.y);

    start = end;
    end = verts.Count;;
    ApplyShadowZeroAlloc(verts, effectColor, start, verts.Count, -effectDistance.x, -effectDistance.y);

    vh.Clear();
    vh.AddUIVertexTriangleStream(verts);
    ListPool<UIVertex>.Release(verts);
}
}

```

This is a very simple method, but it does not display the outlines very smoothly. The reason is that simply shifting the outline up, down, left, and right one time at a time does not display it smoothly. A possible fix is to draw the shadow an arbitrary number of times instead of four times. The sample code is shown below.

```

public class CircleOutline : Shadow
{
    // Number of times to draw. Defaults to 4, the same as Outline.
    [SerializeField]
    private int m_EdgeCount = 4;

```

```

public override void ModifyMesh(VertexHelper vh)
{
    if (!IsActive())
    {
        return;
    }

    var verts = new List<UIVertex>();
    vh.GetUIVertexStream(verts);

    // allocate vertices for the original Graphic + number of draws
    int neededCapacity = verts.Count * (1 + m_EdgeCount);
    if (verts.Capacity < neededCapacity)
    {
        verts.Capacity = neededCapacity;
    }

    int start = 0;
    int end = verts.Count;

    // radius is taken from x
    float radius = Mathf.Abs(effectDistance.x);

    // draw the shadow in a circle
    for (int i = 0; i < m_EdgeCount; i++)
    {
        // Find the angle of the point on the circumference
        float theta = (float) (i * Mathf.PI * 2) / m_EdgeCount;

        // it will look better if it is tilted instead of straight
        theta += (Mathf.PI / m_EdgeCount);

        // get the position corresponding to m_EffectDistance
        float x = radius * Mathf.Cos(theta);
        float y = radius * Mathf.Sin(theta);

        ApplyShadowZeroAlloc(verts, effectColor, start, verts.Count, x, y);
        start = end;
        end = verts.Count;
    }

    vh.Clear();
}

```

```
    vh.AddUIVertexTriangleStream(verts);
}
}
```

If you set `m_EdgeCount` to 8 or so, you can get a pretty nice contour line.

## Public methods of Outline

```
public override void ModifyMesh(VertexHelper vh)
```

Called when rebuilding a mesh, for example via [Graphic.Rebuild\(\)](#).

## Draw the outline with a shader instead of an Outline

The above is a sample code for [CircleOutline](#), a component that draws shadows for an arbitrary number of times, but it consumes the fill rate due to overdraw, even if the draw call is once. However, even though the draw call is only once, it consumes the fill rate due to the overdraw. Therefore, an alternative method is to draw the outline using shaders. The shader for drawing the outline is shown below.

*Outline shaders based on the UI default shaders in Unity 2019.4 and earlier*

```
Shader "UI/LegacyOutline"
{
    Properties
    {
        [PerRendererData] _MainTex("Sprite Texture", 2D) = "white" {}
        _Color("Tint", Color) = (1,1,1,1)

        // Color of the outline
        _EffectColor("Effect Color", Color) = (0, 0, 0, 0.5)

        // Distance of the outline
        _EffectDistance("Effect Distance", Float) = 1

        _StencilComp("Stencil Comparison", Float) = 8
        _Stencil("Stencil ID", Float) = 0
        _StencilOp("Stencil Operation", Float) = 0
        _StencilWriteMask("Stencil Write Mask", Float) = 255
        _StencilReadMask("Stencil Read Mask", Float) = 255

        _ColorMask("Color Mask", Float) = 15

        [Toggle(UNITY_UI_ALPHA_CLIP)] _UseUIAlphaClip("Use Alpha Clip", Float) = 0
    }

    SubShader
    {
        Tags
        {
            "Queue" = "Transparent"
            "IgnoreProjector" = "True"
```

```

    "RenderType" = "Transparent"
    "PreviewType" = "Plane"
    "CanUseSpriteAtlas" = "True"
}

Stencil
{
    Ref[_Stencil]
    Comp[_StencilComp]
    Pass[_StencilOp]
    ReadMask[_StencilReadMask]
    WriteMask[_StencilWriteMask]
}

Cull Off
Lighting Off
ZWrite Off
ZTest[unity_GUIZTestMode]
Blend SrcAlpha OneMinusSrcAlpha
ColorMask[_ColorMask]

Pass
{
    Name "Default"
CGPROGRAM
#pragma vertex vert
#pragma fragment frag
#pragma target 2.0

#include "UnityCG.cginc"
#include "UnityUI.cginc"

#pragma multi_compile_local _ UNITY_UI_CLIP_RECT
#pragma multi_compile_local _ UNITY_UI_ALPHACLIP

struct appdata_t
{
    float4 vertex : POSITION;
    float4 color : COLOR;
    float2 texcoord : TEXCOORD0;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};

```

```

struct v2f
{
    float4 vertex : SV_POSITION;
    fixed4 color : COLOR;
    float2 texcoord : TEXCOORD0;
    float4 worldPosition : TEXCOORD1;
    half4 mask : TEXCOORD2;
    UNITY_VERTEX_OUTPUT_STEREO
};

sampler2D _MainTex;
fixed4 _Color;
fixed4 _TextureSampleAdd;
float4 _ClipRect;
float4 _MainTex_ST;
float _UIMaskSoftnessX;
float _UIMaskSoftnessY;

// Receive color and distance from properties
half4 _EffectColor;
half _EffectDistance;

// Texture size
// x : 1 / width
// y : 1 / height
// z : width
// w : height
float4 _MainTex_TexelSize;

// Keep the vertex shader
v2f vert(appdata_t v)
{
    v2f OUT;
    UNITY_SETUP_INSTANCE_ID(v);
    UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(OUT);
    float4 vPosition = UnityObjectToClipPos(v.vertex);
    OUT.worldPosition = v.vertex;
    OUT.vertex = vPosition;

    float2 pixelSize = vPosition.w;
    pixelSize /= float2(1, 1) * abs(mul((float2x2)UNITY_MATRIX_P, _ScreenParams.xy))
}

```

```

);

float4 clampedRect = clamp(_ClipRect, -2e10, 2e10);
float2 maskUV = (v.vertex.xy - clampedRect.xy) / (clampedRect.zw - clampedRect.xy)
;
OUT.texcoord = TRANSFORM_TEX(v.texcoord.xy, _MainTex);
OUT.mask = half4(v.vertex.xy * 2 - clampedRect.xy - clampedRect.zw, 0.25 / (0.25 *
half2(_UIMaskSoftnessX, _UIMaskSoftnessY) + abs(pixelSize.xy )));

OUT.color = v.color * _Color;
return OUT;
}

// Handle outlines in the fragment shader
half4 frag_outline(v2f IN, half4 color)
{
    // Original alpha
    half original_a = color.a;

    // If original alpha is 1, use original color, if original alpha is 0, use outline color once
    color = lerp(_EffectColor, color, original_a);

    float delta_x = _MainTex_TexelSize.x * _EffectDistance;
    float delta_y = _MainTex_TexelSize.y * _EffectDistance;

    // Get the color of the top, bottom, left and right pixels
    half4 color_top = tex2D(_MainTex, IN.texcoord + float2(0, delta_y));
    half4 color_bottom = tex2D(_MainTex, IN.texcoord - float2(0, delta_y));
    half4 color_left = tex2D(_MainTex, IN.texcoord - float2(delta_x, 0));
    half4 color_right = tex2D(_MainTex, IN.texcoord + float2(delta_x, 0));

    // Distance between diagonal points is  $\sqrt{2} / 2$  times
    delta_x *= 0.707;
    delta_y *= 0.707;

    // Get the colors of the top-left, bottom-left, top-right, and bottom-right pixels
    half4 color_left_top = tex2D(_MainTex, IN.texcoord - float2(delta_x, -delta_y));
    half4 color_left_bottom = tex2D(_MainTex, IN.texcoord - float2(delta_x, delta_y));
    half4 color_right_top = tex2D(_MainTex, IN.texcoord + float2(delta_x, delta_y));
    half4 color_right_bottom = tex2D(_MainTex, IN.texcoord + float2(delta_x, -delta_y));

    // Use the alpha of the pixel with the highest alpha among the original and surrounding
}

```

```

pixels
    color.a = max(original_a, max(max(max(color_top.a, color_bottom.a), max(color_left.a
    , color_right.a)),
        max(max(color_left_top.a, color_left_bottom.a), max(color_right_top.a, color_right_
bottom.a))));

    return color;
}

fixed4 frag(v2f IN) : SV_Target
{
    half4 color = IN.color * (tex2D(_MainTex, IN.texcoord) + _TextureSampleAdd);

    #ifdef UNITY_UI_CLIP_RECT
    half2 m = saturate({_ClipRect.zw - _ClipRect.xy - abs(IN.mask.xy)} * IN.mask.zw);
    color.a *= m.x * m.y;
    #endif

    // Apply the outline
    color = frag_outline(IN, color);

#ifdef UNITY_UI_ALPHA_CLIP
    clip(color.a - 0.001);
#endif
    color.rgb *= color.a;

    return color;
}
ENDCG
}
}
}

```

*Outline shaders based on the UI default shaders in Unity 2020.1 and later*

```

Shader "UI/Outline"
{
    Properties
    {
        [PerRendererData] _MainTex("Sprite Texture", 2D) = "white" {}
    }
}
```

```

_Color("Tint", Color) = (1,1,1,1)

// Color of the outline
_EffectColor("Effect Color", Color) = (0, 0, 0, 0.5)

// Distance of the outline
_EffectDistance("Effect Distance", Float) = 1

_StencilComp("Stencil Comparison", Float) = 8
_Stencil("Stencil ID", Float) = 0
_StencilOp("Stencil Operation", Float) = 0
_StencilWriteMask("Stencil Write Mask", Float) = 255
_StencilReadMask("Stencil Read Mask", Float) = 255

_ColorMask("Color Mask", Float) = 15

[Toggle(UNITY_UI_ALPHA_CLIP)] _UseUIAlphaClip("Use Alpha Clip", Float) = 0
}

SubShader
{
    Tags
    {
        "Queue" = "Transparent"
        "IgnoreProjector" = "True"
        "RenderType" = "Transparent"
        "PreviewType" = "Plane"
        "CanUseSpriteAtlas" = "True"
    }

    Stencil
    {
        Ref[_Stencil]
        Comp[_StencilComp]
        Pass[_StencilOp]
        ReadMask[_StencilReadMask]
        WriteMask[_StencilWriteMask]
    }

    Cull Off
    Lighting Off
    ZWrite Off
}

```

```

ZTest[unity_GUIZTestMode]
Blend One OneMinusSrcAlpha
ColorMask[_ColorMask]

Pass
{
    Name "Default"
    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag
    #pragma target 2.0

    #include "UnityCG.cginc"
    #include "UnityUI.cginc"

    #pragma multi_compile_local _ UNITY_UI_CLIP_RECT
    #pragma multi_compile_local _ UNITY_UI_ALPHACLIP

    struct appdata_t
    {
        float4 vertex : POSITION;
        float4 color : COLOR;
        float2 texcoord : TEXCOORD0;
        UNITY_VERTEX_INPUT_INSTANCE_ID
    };

    struct v2f
    {
        float4 vertex : SV_POSITION;
        fixed4 color : COLOR;
        float2 texcoord : TEXCOORD0;
        float4 worldPosition : TEXCOORD1;
        half4 mask : TEXCOORD2;
        UNITY_VERTEX_OUTPUT_STEREO
    };

    sampler2D _MainTex;
    fixed4 _Color;
    fixed4 _TextureSampleAdd;
    float4 _ClipRect;
    float4 _MainTex_ST;
    float _MaskSoftnessX;
}

```

```

float _MaskSoftnessY;

// Receive color and distance from properties
half4 _EffectColor;
half _EffectDistance;

// Texture size
// x : 1 / width
// y : 1 / height
// z : width
// w : height
float4 _MainTex_TexelSize;

// Keep the vertex shader
v2f vert(appdata_t v)
{
    v2f OUT;
    UNITY_SETUP_INSTANCE_ID(v);
    UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(OUT);
    float4 vPosition = UnityObjectToClipPos(v.vertex);
    OUT.worldPosition = v.vertex;
    OUT.vertex = vPosition;

    float2 pixelSize = vPosition.w;
    pixelSize /= float2(1, 1) * abs(mul((float2x2)UNITY_MATRIX_P, _ScreenParams.xy));
}

float4 clampedRect = clamp(_ClipRect, -2e10, 2e10);
float2 maskUV = (v.vertex.xy - clampedRect.xy) / (clampedRect.zw - clampedRect.xy);
;
    OUT.texcoord = float4(v.texcoord.x, v.texcoord.y, maskUV.x, maskUV.y);
    OUT.mask = half4(v.vertex.xy * 2 - clampedRect.xy - clampedRect.zw, 0.25 / (0.25 *
half2(_MaskSoftnessX, _MaskSoftnessY) + abs(pixelSize.xy)));
    OUT.color = v.color * _Color;
    return OUT;
}

// Handle outlines in the fragment shader
half4 frag_outline(v2f IN, half4 color)
{
    // original alpha

```

```

half original_a = color.a;

// If original alpha is 1, use original color, if original alpha is 0, use outline color once
color = lerp(_EffectColor, color, original_a);

float delta_x = _MainTex_TexelSize.x * _EffectDistance;
float delta_y = _MainTex_TexelSize.y * _EffectDistance;

// Get the color of the top, bottom, left and right pixels
half4 color_top = tex2D(_MainTex, IN.texcoord + float2(0, delta_y));
half4 color_bottom = tex2D(_MainTex, IN.texcoord - float2(0, delta_y));
half4 color_left = tex2D(_MainTex, IN.texcoord - float2(delta_x, 0));
half4 color_right = tex2D(_MainTex, IN.texcoord + float2(delta_x, 0));

// Distance between diagonal points is  $\sqrt{2} / 2$  times
delta_x *= 0.707;
delta_y *= 0.707;

// Get the colors of the top-left, bottom-left, top-right, and bottom-right pixels
half4 color_left_top = tex2D(_MainTex, IN.texcoord - float2(delta_x, -delta_y));
half4 color_left_bottom = tex2D(_MainTex, IN.texcoord - float2(delta_x, delta_y));
half4 color_right_top = tex2D(_MainTex, IN.texcoord + float2(delta_x, delta_y));
half4 color_right_bottom = tex2D(_MainTex, IN.texcoord + float2(delta_x, -delta_y));

// Use the alpha of the pixel with the highest alpha among the original and surrounding
pixels
color.a = max(original_a, max(max(max(color_top.a, color_bottom.a), max(color_left.a
, color_right.a)),
max(max(color_left_top.a, color_left_bottom.a), max(color_right_top.a, color_right_
bottom.a))));

return color;
}

fixed4 frag(v2f IN) : SV_Target
{
    half4 color = (tex2D(_MainTex, IN.texcoord) + _TextureSampleAdd) * IN.color;

#define UNITY_UI_CLIP_RECT
half2 m = saturate((_ClipRect.zw - _ClipRect.xy - abs(IN.mask.xy)) * IN.mask.zw);
color.a *= m.x * m.y;
#endif
}

```

```
// Apply the outline
color = frag_outline(IN, color);

#ifndef UNITY_UI_ALPHA_CLIP
clip(color.a - 0.001);
#endif

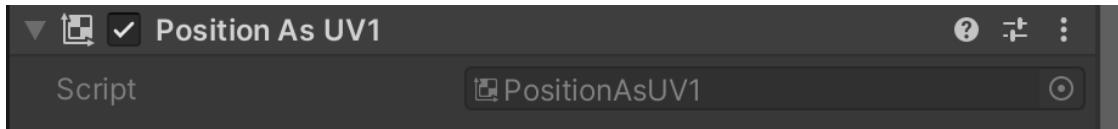
color.rgb *= color.a;

return color;
}

ENDCG
}

}
```

## PositionAsUV1component



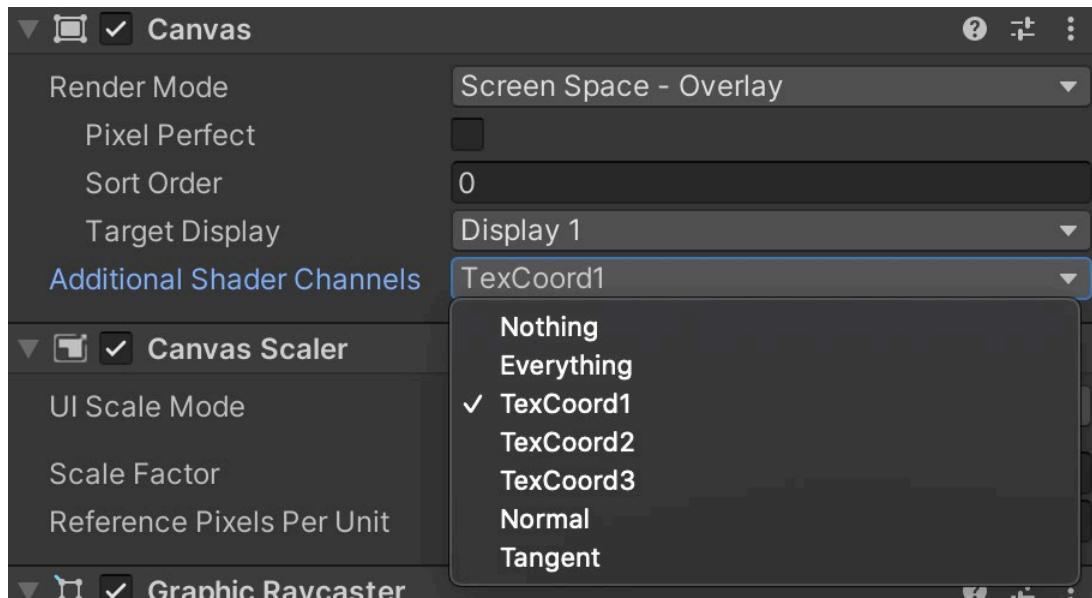
```
[AddComponentMenu("UI/Effects/Position As UV1", 16)]
public class PositionAsUV1 : BaseMeshEffect
```

The `PositionAsUV1` component terminal is a component that sets the vertex position to `uv1` of the mesh vertex.

The following is a sample code that prepares the x-coordinate and y-coordinate of a vertex in `uv1`.

```
public override void ModifyMesh(VertexHelper vh)
{
    UIVertex vert = new UIVertex();
    for (int i = 0; i < vh.currentVertCount; i++)
    {
        vh.PopulateUIVertex(ref vert, i);
        vert.uv1 = new Vector2(vert.position.x, vert.position.y);
        vh.SetUIVertex(vert, i);
    }
}
```

In order to use `uv1`, the `TexCoord1` bit of `additionalShaderChannels` in `Canvas` must be enabled.



By passing the `uv1` received by the vertex shader to the fragment shader, we can change the color of the pixels according to the position of the vertex in the object.

*Shaders using UV1, based on UI default shaders in Unity 2019.4 and earlier*

```
Shader "UI/LegacyUseUV1"
{
    Properties
    {
        [PerRendererData] _MainTex("Sprite Texture", 2D) = "white" {}
        _Color("Tint", Color) = (1,1,1,1)

        _StencilComp("Stencil Comparison", Float) = 8
        _Stencil("Stencil ID", Float) = 0
        _StencilOp("Stencil Operation", Float) = 0
        _StencilWriteMask("Stencil Write Mask", Float) = 255
        _StencilReadMask("Stencil Read Mask", Float) = 255

        _ColorMask("Color Mask", Float) = 15

        [Toggle(UNITY_UI_ALPHA_CLIP)] _UseUIAlphaClip("Use Alpha Clip", Float) = 0
    }
}
```

```

SubShader
{
    Tags
    {
        "Queue" = "Transparent"
        "IgnoreProjector" = "True"
        "RenderType" = "Transparent"
        "PreviewType" = "Plane"
        "CanUseSpriteAtlas" = "True"
    }

    Stencil
    {
        Ref[_Stencil]
        Comp[_StencilComp]
        Pass[_StencilOp]
        ReadMask[_StencilReadMask]
        WriteMask[_StencilWriteMask]
    }

    Cull Off
    Lighting Off
    ZWrite Off
    ZTest[unity_GUIZTestMode]
    Blend SrcAlpha OneMinusSrcAlpha
    ColorMask[_ColorMask]

    Pass
    {
        Name "Default"
        CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma target 2.0

            #include "UnityCG.cginc"
            #include "UnityUI.cginc"

            #pragma multi_compile_local _ UNITY_UI_CLIP_RECT
            #pragma multi_compile_local _ UNITY_UI_ALPHACLIP

```

```

struct appdata_t
{
    float4 vertex : POSITION;
    float4 color : COLOR;
    float2 texcoord : TEXCOORD0;

    // Position of the vertex in the model space passed as UV1 in PositionAsUV1
    float4 positionInObject : TEXCOORD1;

    UNITY_VERTEX_INPUT_INSTANCE_ID
};

struct v2f
{
    float4 vertex : SV_POSITION;
    fixed4 color : COLOR;
    float2 texcoord : TEXCOORD0;
    float4 worldPosition : TEXCOORD1;
    half4 mask : TEXCOORD2;

    // position of the pixel in the model space
    float4 positionInObject : TEXCOORD3;

    UNITY_VERTEX_OUTPUT_STEREO
};

sampler2D _MainTex;
fixed4 _Color;
fixed4 _TextureSampleAdd;
float4 _ClipRect;
float4 _MainTex_ST;
float _UIMaskSoftnessX;
float _UIMaskSoftnessY;

v2f vert(appdata_t v)
{
    v2f OUT;
    UNITY_SETUP_INSTANCE_ID(v);
    UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(OUT);
    float4 vPosition = UnityObjectToClipPos(v.vertex);
    OUT.worldPosition = v.vertex;
    OUT.vertex = vPosition;
}

```

```

        float2 pixelSize = vPosition.w;
        pixelSize /= float2(1, 1) * abs(mul((float2x2)UNITY_MATRIX_P, _ScreenParams.xy))
    );

        float4 clampedRect = clamp(_ClipRect, -2e10, 2e10);
        float2 maskUV = (v.vertex.xy - clampedRect.xy) / (clampedRect.zw - clampedRect.xy)
    ;
        OUT.texcoord = TRANSFORM_TEX(v.texcoord.xy, _MainTex);
        OUT.mask = half4(v.vertex.xy * 2 - clampedRect.xy - clampedRect.zw, 0.25 / (0.25 *
half2(_UIMaskSoftnessX, _UIMaskSoftnessY) + abs(pixelSize.xy )));

        // Pass the position of the vertex in the model space
        OUT.positionInObject = v.positionInObject;

        OUT.color = v.color * _Color;
        return OUT;
    }

fixed4 frag(v2f IN) : SV_Target
{
    half4 color = IN.color * (tex2D(_MainTex, IN.texcoord) + _TextureSampleAdd);

    #ifdef UNITY_UI_CLIP_RECT
    half2 m = saturate((_ClipRect.zw - _ClipRect.xy - abs(IN.mask.xy)) * IN.mask.zw);
    color.a *= m.x * m.y;
    #endif

    #ifdef UNITY_UI_ALPHA_CLIP
    clip(color.a - 0.001);
    #endif

    // Make the alpha value 0 in a circle from the center
    color.a = 1 - clamp(length(IN.positionInObject) / 100, 0, 1);

    return color;
}
ENDCG
}
}
}

```

*Shaders using UV1, based on UI default shaders in Unity 2020.1 and later*

```
Shader "UI/UseUV1"
{
    Properties
    {
        [PerRendererData] _MainTex("Sprite Texture", 2D) = "white" {}
        _Color("Tint", Color) = (1,1,1,1)

        _StencilComp("Stencil Comparison", Float) = 8
        _Stencil("Stencil ID", Float) = 0
        _StencilOp("Stencil Operation", Float) = 0
        _StencilWriteMask("Stencil Write Mask", Float) = 255
        _StencilReadMask("Stencil Read Mask", Float) = 255

        _ColorMask("Color Mask", Float) = 15

        [Toggle(UNITY_UI_ALPHA_CLIP)] _UseUIAlphaClip("Use Alpha Clip", Float) = 0
    }

    SubShader
    {
        Tags
        {
            "Queue" = "Transparent"
            "IgnoreProjector" = "True"
            "RenderType" = "Transparent"
            "PreviewType" = "Plane"
            "CanUseSpriteAtlas" = "True"
        }

        Stencil
        {
            Ref[_Stencil]
            Comp[_StencilComp]
            Pass[_StencilOp]
            ReadMask[_StencilReadMask]
            WriteMask[_StencilWriteMask]
        }
    }

    Cull Off
```

```

Lighting Off
ZWrite Off
ZTest[unity_GUIZTestMode]
Blend SrcAlpha OneMinusSrcAlpha
ColorMask[_ColorMask]

Pass
{
    Name "Default"
    CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #pragma target 2.0

        #include "UnityCG.cginc"
        #include "UnityUI.cginc"

        #pragma multi_compile_local _ UNITY_UI_CLIP_RECT
        #pragma multi_compile_local _ UNITY_UI_ALPHACLIP

        struct appdata_t
        {
            float4 vertex : POSITION;
            float4 color : COLOR;
            float2 texcoord : TEXCOORD0;

            // position of the vertex in the model space passed as UV1 in PositionAsUV1
            float4 positionInObject : TEXCOORD1;

            UNITY_VERTEX_INPUT_INSTANCE_ID
        };

        struct v2f
        {
            float4 vertex : SV_POSITION;
            fixed4 color : COLOR;
            float2 texcoord : TEXCOORD0;
            float4 worldPosition : TEXCOORD1;
            half4 mask : TEXCOORD2;

            // position of the pixel in the model space
            float4 positionInObject : TEXCOORD3;
        };
    
```

```

    UNITY_VERTEX_OUTPUT_STEREO
};

sampler2D _MainTex;
fixed4 _Color;
fixed4 _TextureSampleAdd;
float4 _ClipRect;
float4 _MainTex_ST;
float _UIMaskSoftnessX;
float _UIMaskSoftnessY;

v2f vert(appdata_t v)
{
    v2f OUT;
    UNITY_SETUP_INSTANCE_ID(v);
    UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(OUT);
    float4 vPosition = UnityObjectToClipPos(v.vertex);
    OUT.worldPosition = v.vertex;
    OUT.vertex = vPosition;

    float2 pixelSize = vPosition.w;
    pixelSize /= float2(1, 1) * abs(mul((float2x2)UNITY_MATRIX_P, _ScreenParams.xy));
);

    float4 clampedRect = clamp(_ClipRect, -2e10, 2e10);
    float2 maskUV = (v.vertex.xy - clampedRect.xy) / (clampedRect.zw - clampedRect.xy);
;
    OUT.texcoord = TRANSFORM_TEX(v.texcoord.xy, _MainTex);
    OUT.mask = half4(v.vertex.xy * 2 - clampedRect.xy - clampedRect.zw, 0.25 / (0.25 *
half2(_UIMaskSoftnessX, _UIMaskSoftnessY) + abs(pixelSize.xy )));

    // Pass the position of the vertex in the model space
    OUT.positionInObject = v.positionInObject;

    OUT.color = v.color * _Color;
    return OUT;
}

fixed4 frag(v2f IN) : SV_Target
{
    half4 color = IN.color * (tex2D(_MainTex, IN.texcoord) + _TextureSampleAdd);

```

```

#define UNITY_UI_CLIP_RECT
half2 m = saturate({_ClipRect.zw - _ClipRect.xy - abs(IN.mask.xy)) * IN.mask.zw);
color.a *= m.x * m.y;
#endif

#define UNITY_UI_ALPHA_CLIP
clip(color.a - 0.001);
#endif

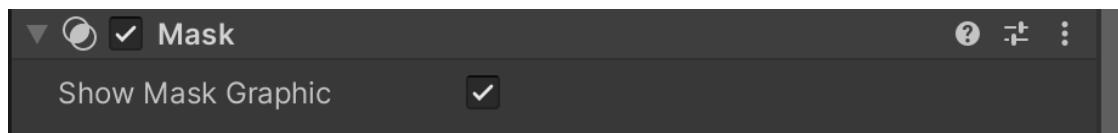
// make the alpha value 0 in a circle from the center
color.a = 1 - clamp(length(IN.positionInObject) / 100, 0, 1);

return color;
}
ENDCG
}
}
}

```

Although it may be rare to use the [PositionAsUV1](#) component itself, you may want to refer to this component for how to put arbitrary data into `uv1`, etc.

## Mask component



```
[AddComponentMenu("UI/Mask", 13)]
[ExecuteAlways]
[RequireComponent(typeof(RectTransform))]
[DisallowMultipleComponent]
public class Mask : UIBehaviour, ICanvasRaycastFilter, IMaterialModifier
```

The [Mask](#) component is a component that masks the underlying UI element using a stencil test. This allows the image to be hollowed out and displayed in an arbitrary way.

If you simply want to mask with a rectangle, it may be better to replace it with [RectMask2D](#) component for better performance. Performance will vary depending on the platform and UI configuration, so be sure to measure and select the right one.

In the following, we will discuss the basics of stencil testing and its behavior.

## Stencil test

The stencil test is a process that is performed for each fragment after the fragment shader. There is (usually) an 8-bit buffer for each fragment, which is called the stencil buffer.

During the stencil test, a comparison test between the stencil value of the currently processed fragment and the value of the stencil buffer is performed, and if the test fails, the fragment is discarded. For example, if the stencil value of the currently processed fragment is **1**, the value of the stencil buffer corresponding to the current fragment is **0**, and the comparison function is **Equal**, the stencil test will fail and the fragment will be discarded.

### Note

Normally, the natural flow is to run the stencil tests after the fragment shader (in the OpenGL rendering pipeline, the stencil tests are supposed to be run after the fragment shader), but if the results of the stencil tests are determined before running the fragment shader, then running the stencil tests first can improve performance. ), but if the results of the stencil tests are known before the fragment shaders are run, then the stencil tests can be run first to improve performance. Whether or not this is done depends on the GPU and driver.

A stencil is a piece of stationery with holes in it to draw the same shape.

The behavior of the stencil test is described in the **Stencil** section in the **SubShader** or **Pass** section of the shader. If no stencil tests are performed, the **Stencil** section will look like this

```
SubShader
{
    ...
    Stencil
    {
        // Reference value for stencil test
        Ref 0

        // Comparison function always succeeds stencil test (do not destroy)
        Comp Always

        // Behavior on successful stencil test is Keep (do nothing)
        Pass Keep

        // Bit mask when reading buffer. 0xFF means to use buffer contents as is.
    }
}
```

```
ReadMask 255
```

```
// Bit mask for buffer write. 0xFF means that buffer contents are used as is.  
WriteMask 255  
... }  
...
```

In addition, the default UI shader allows you to specify stencil test parameters such as `Ref`, `Comp`, `ReadMask`, and `WriteMask` (specified immediately above) via properties.

```
Shader "UI/Default"  
{  
    Properties  
    { ...  
        // Stencil Compare Function  
        // Defined in UnityEngine.Rendering.CompareFunction  
        // https://docs.unity3d.com/ScriptReference/Rendering.CompareFunction.html  
        // 8 is Always, which means the stencil test always succeeds  
        _StencilComp ("Stencil Comparison", Float) = 8  
  
        // Stencil test reference value (0-255)  
        _Stencil ("Stencil ID", Float) = 0  
  
        // Behavior on successful stencil test  
        // Defined in UnityEngine.Rendering.StencilOp  
        // https://docs.unity3d.com/ja/current/ScriptReference/Rendering.StencilOp.html  
        // 0 is Keep, no changes will be made  
        _StencilOp ("Stencil Operation", Float) = 0  
  
        // Mask that specifies the bit to write the reference value to the buffer after the stencil test is  
        // performed.  
        // 0xFF, so reference value and buffer contents are compared as is  
        _StencilWriteMask ("Stencil Write Mask", Float) = 255  
  
        // OR mask to apply to both the base value and the buffer contents before stencil testing  
        // 0xFF, so both the base value and buffer contents are compared as is  
        _StencilReadMask ("Stencil Read Mask", Float) = 255  
        ...  
    }  
}
```

```

SubShader
{
...
// actually set the stencil settings specified by the property
// https://docs.unity3d.com/ja/current/Manual/SL-Stencil.html
Stencil
{
...
// Base value for stencil test
Ref [_Stencil]

// Comparison function
Comp [_StencilComp]

// Behavior on successful stencil test
Pass [_StencilOp]

// Bit mask for buffer read
ReadMask [_StencilReadMask]

// Bitmask for buffer write
WriteMask [_StencilWriteMask] }
}

...

```

The stencil comparison function is defined as `CompareFunctionUnityEngine.Rendering`, and the behavior on a successful stencil test is defined as `UnityEngine.Rendering`.

```

namespace UnityEngine.Rendering
{
public enum CompareFunction
{
// Stencil test is disabled.
Disabled = 0,

// stencil test always fails
Never = 1,

// Stencil test succeeds if new value is less than the value read from buffer
Less = 2,
}

```

```

// Stencil test succeeds if new value is equal to value read from buffer
Equal = 3,

// Stencil test succeeds if new value is less than or equal to value read from buffer
LessEqual = 4,

// Stencil test succeeds if the new value is greater than the value read from the buffer
Greater = 5,

// Stencil test succeeds if new value is different from value read from buffer
NotEqual = 6,

// Stencil test succeeds if new value is greater than or equal to value read from buffer
GreaterEqual = 7,

// Stencil test always succeeds
Always = 8
}

}

namespace UnityEngine.Rendering
{
    public enum StencilOp
    {
        // Keep the current value.
        Keep = 0,

        // Set the stencil buffer value to 0
        Zero = 1,

        // Replace the stencil buffer value with the base value
        Replace = 2,

        // Increment the value of the stencil buffer.
        // The maximum value is the value that can be represented as unsigned (255 if the stencil buffer is 8-bit).
        IncrementSaturate = 3,

        // Decrement the value of the stencil buffer.
        // The minimum value is 0.
        DecrementSaturate = 4,
    }
}

```

```

// Invert the current stencil buffer value by a bit.
Invert = 5,

// Increments the value of the stencil buffer.
// Increment the stencil buffer value, or 0 if the maximum value has been incremented.
IncrementWrap = 6,

// Decrement the value of the stencil buffer.
// If decremented 0, set to maximum value.
DecrementWrap = 7
}
}

```

By setting the value of the property externally, the behavior of the [Mask](#) component and its children's shaders is changed to achieve the mask. the value of the shader property for the Mask component and its children's stencil test is created inside the [IMaterialModifier](#) implementation [StencilMaterial](#) class inside [GetModifiedMaterial\(\)](#).

The material used by the [Mask](#) component can be obtained from [Mask.GetModifiedMaterial\(\)](#). The following code is an excerpted and simplified version of [Mask](#).

*Packages/com.unity.ugui/Runtime/UI/Core/Mask.cs*

```

public virtual Material GetModifiedMaterial(Material baseMaterial)
{
    // Calculate the base value of the stencil based on the depth in the Hierarchy
    var stencilDepth = MaskUtilities.GetStencilDepth(transform, rootSortCanvas);

    // The base values are 0x1, 0x3, 0x7, ..., starting with the mask with the shallowest depth in the
    // Hierarchy. from Mask.
    int desiredStencilBit = 1 << stencilDepth;

    if (desiredStencilBit == 1) // shallowest Mask
    {
        // Create a material to perform the stencil test.
        // Shader properties are.
        // Base value: desiredStencilBit calculated above
        // On success: Always
        // Comparison function: Equal
        // ColorMask: if the Mask component's showMaskGraphic is true, then 0xFF to show the ma
    }
}

```

```

sk image; if false, then 0 to not show the image.
    // ReadMask: omitted, 0xFF
    // WriteMask: omitted, 0xFF
    var maskMaterial = StencilMaterial.Add(baseMaterial, 1, StencilOp.Replace, CompareFunction.Always, m_ShowMaskGraphic ? ColorWriteMask.All : 0);

    // Calling StencilMaterial.Add() adds to the static List, which is unnecessary, so remove it
    StencilMaterial.Remove(m_MaskMaterial);

    // use this material
    m_MaskMaterial = maskMaterial;
}

else // the mask at the second and subsequent depths
{
    // Create a material for the stencil test.
    // Shader properties are.
    // Base value: desiredStencilBit calculated above
    // On success: Replace
    // Comparison function: Equal
    // ColorMask: 0xF if the Mask component's showMaskGraphic is true (= all RGBA will be output); false: 0 since no image will be shown.
    // ReadMask: bitmask that can only drop its own reference value
    // WriteMask: Logical OR of the bits that are only your reference value and the other bits
    var maskMaterial2 = StencilMaterial.Add(baseMaterial, desiredStencilBit | (desiredStencilBit - 1), StencilOp.Replace, CompareFunction.Equal, m_ShowMaskGraphic ? ColorWriteMask.All : 0, desiredStencilBit - 1, desiredStencilBit | (desiredStencilBit - 1));

    // Calling StencilMaterial.Add() adds to the static List, but this is not necessary, so remove it
    StencilMaterial.Remove(m_MaskMaterial);

    // Use this material
    m_MaskMaterial = maskMaterial;
}

```

On the other hand, the material used by the Mask component's children (and grandchildren, great-grandchildren, etc.) can be obtained from [MaskableGraphic](#).

*Packages/com.unity.ugui/Runtime/UI/Core/MaskableGraphic.cs*

```

// Return the material to use for rendering
public virtual Material GetModifiedMaterial(Material baseMaterial)
{
    ...
    // Calculate the base value of the stencil based on its depth in the Hierarchy
    m_SignedDepthValue = maskable ? MaskUtilities.GetStencilDepth(transform, rootCanvas) : 0;

    ...
    // Create a material to perform the stencil test.
    // Shader properties are.
    // Base value: the depth calculated above
    // On success: Keep
    // Comparison function: Equal
    // ColorMask: 0xF (= output all RGBA)
    // ReadMask: bits from which the reference value can be read
    // WriteMask: 0
    var maskMat = StencilMaterial.Add(toUse, (1 << m_SignedDepthValue) - 1, StencilOp.Keep, Comp
areFunction.Equal, ColorWriteMask.All, (1 <&lt; m_SignedDepthValue) - 1, 0);

```

To simulate the immediate shader code, the [Stencil](#) section of the [Mask](#) component in the shallowest Hierarchy looks like this

```

SubShader
{
    ...
    Stencil
    {
        // Stencil test reference value
        Ref 1

        // Comparison function always succeeds stencil test (do not destroy)
        Comp Always

        // Behavior on successful stencil test is Replace (replace with base value)
        Pass Replace

        // Bit mask when reading buffer, 0xFF, so use buffer contents as is.
        ReadMask 255

        // Bit mask for buffer write, 0xFF, so buffer contents are used as is.
    }
}

```

```
    WriteMask 255  
    ... }  
    ...
```

On the other hand, the [Stencil](#) section of a UI element that is a child of [Mask](#) will look like this

```
SubShader  
{  
    ...  
    Stencil  
    {  
        // Stencil test reference value  
        Ref 1  
  
        // Comparison function succeeds if the values match  
        Comp Equal  
  
        // Stencil test succeeds, behavior is Keep (do nothing)  
        Pass Keep  
  
        // Bit mask for buffer read, 1 so only the lower 1 bit of the buffer content is used.  
        ReadMask 1  
  
        // Bitmask for buffer write, 0, so no buffer contents are used.  
        WriteMask 0  
    ... }  
    ...
```

Let's look at the actual stencil test using the Mask component. uGUI processes the stencils from the shallow end of the hierarchy, so the stencil test of the [Mask](#) component is performed as follows.

1. The fragment shader will be executed.
2. Read the value of the stencil buffer corresponding to this fragment as it is ([ReadMask](#) is **0xFF**).
3. Stencil tests are always successful ([Always](#)).
4. Since the stencil test was successful, the **0** (default value) in the stencil buffer is replaced with **1 (Replace)**.
5. The replaced value of **1** is written as is ([WriteMask](#) is **0xFF**).

The subsequent processing of the UI elements of the [Mask](#) component's children (and grandchildren, great-grandchildren, and other underlying components) will be as follows.

1. The fragment shader will be executed.
2. When we read the value of the stencil buffer corresponding to this fragment, we get [1](#) if the pixel in [Mask](#) is present, otherwise we get [0](#) (ReadMask is [0x1](#)).
3. If the stencil test criterion [1](#) equals the value obtained above, the stencil test succeeds and the UI element's pixels are rendered. Otherwise, the stencil test will fail and the UI element's pixels will not be rendered.
4. Since WriteMask is [0x0](#), nothing is written to the stencil buffer.

## Whether or not to use the stencil test

Stencil tests are supported on all platforms currently supported by Unity. This is evident by the fact that [SystemInfo.supportsStencil](#), which returns whether or not stencil tests are supported, always returns `true`. However, unfortunately, stencil tests do not work under some conditions. Some of these conditions are shown below.

- If the [RenderTexture](#) is created with 16 bit depth, the stencil buffer will not be created and the stencil test will not work. Note that the stencil buffer will be created by itself when running on Editor, so you may not be able to check it unless you are using the actual device.
- Stencils will be disabled if **Disable Depth and Stencil** is enabled in the *Player Settings* in Android. This is a trap that the Oculus documentation recommends enabling for better performance.
- Stencils may not work properly on some Android devices, and this should be taken into account when using stencils on platforms that are expected to run on an unspecified number of Android devices, such as Google Play Store. It may be necessary to exclude such devices from support.

If you are rendering to [RenderTexture](#), you should call [RenderTexture.SupportsStencil\(\)](#) to check if the stencil buffer is supported. If you are rendering to [RenderTarget](#), you should call [RenderTarget.SupportsStencil\(\)](#) to check if the stencil buffer is supported.

## Properties of Mask

### graphic

```
public Graphic graphic { get; }
```

Get the [Graphic](#) component that is attached to this component.

Normally, this is used to get the [Image](#) or [RawImage](#) attached to the image to be clipped as a mask.

### rectTransform

```
public RectTransform rectTransform { get; }
```

Get the cache of [RectTransform](#).

### showMaskGraphic

```
public bool showMaskGraphic { get; set; }
```

Gets/Sets whether the image to be used as a mask should be displayed or not.

The default value is [true](#), which means that the shader's [ColorMask](#) will be [0xF](#) (= output all RGBA). In this case, the image for the mask will also be rendered.

If this property is [false](#), the shader's [ColorMask](#) will be set to [0](#), and the image for the mask will not be displayed.

## Public methods of Mask's

### GetModifiedMaterial()

```
public virtual Material GetModifiedMaterial(Material baseMaterial);
```

Gets the material to be passed to the [CanvasRenderer](#).

This method is an implementation of the [IMaterialModifier](#) interface.

Generates a material for stencil testing based on the material passed as an argument, and returns it. If the mask is invalid, it will return the passed material as is.

### IsRaycastLocationValid()

```
public virtual bool IsRaycastLocationValid(Vector2 sp, Camera eventCamera);
```

Performs raycast judgment. If a raycast hit occurs at the [screenPoint](#) given in the argument, [true](#) is returned.

This method is an implementation of the [ICanvasRaycastFilter](#) interface.

If the [GameObject](#) is active and the component is [enabled](#), call [RectTransformUtility.RectangleContainsScreenPoint\(\)](#) to return whether the given position is within the [RectTransform](#).

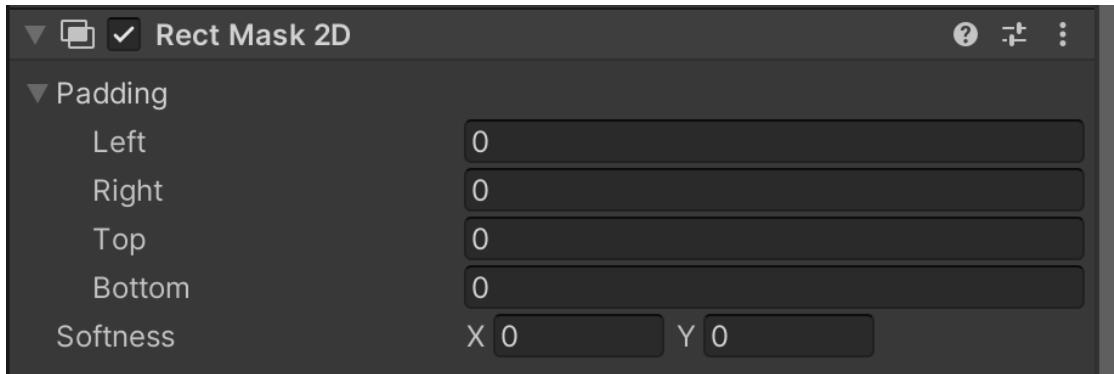
### MaskEnabled()

```
public virtual bool MaskEnabled();
```

Returns whether the mask function is enabled or not.

Returns [true](#) if the [GameObject](#) is active, the component is [enabled](#), and the [Graphic](#) is not [null](#). Otherwise, it returns [false](#).

## RectMask2D component



```
[AddComponentMenu("UI/Rect Mask 2D", 13)]
[ExecuteAlways]
[DisallowMultipleComponent]
[RequireComponent(typeof(RectTransform))]
public class RectMask2D : UIBehaviour, IClipper, ICanvasRaycastFilter
```

RectMask2D is a component for clipping (masking) rectangles in RectTransform, which is similar to the Mask component, but differs in the following ways

- Advantages of RectMask2D over Mask
  - Does not require a stencil buffer.
  - Good performance especially on mobile platforms as it does not increase draw calls.
  - Do not change materials for rendering.
- Disadvantages of RectMask2D compared to Mask
  - Unlike Mask, it cannot be clipped in any way.
  - The mask object must be in the same plane (Z coordinate must be the same).  
However, as long as you are using Canvas normally, this will not be a problem.

Thus, except for the fact that it can only be clipped as a rectangle, RectMask2D is a component with good performance. The default ScrollView uses Mask, but in many cases it is better to replace it with RectMask2D.

The clipping process for RectMask2D is done in the fragment shader, but the calculation of the clipping region is done at the beginning of the Graphic rebuild. There, PerformClipping() of the IClipper registered in ClipperRegistry is called to calculate the clipping area of the child.

Now let's take a look at the clipping process in the fragment shader. If the `UNITY_UI_CLIP_RECT` keyword in the default UI shader is enabled, the alpha value is calculated in the fragment shader. If it is outside the clipping region, the alpha value will be `0` and the pixel will not be displayed.

*Excerpt from UI Default Shaders*

```
// Set a value with CanvasRenderer.EnableRectClipping() from MaskableGraphic.SetClipRect() etc.  
float4 _ClipRect;  
  
...  
  
// Fragment shader  
fixed4 frag(v2f IN) : SV_Target  
{  
    // sample color from texture  
    half4 color = (tex2D(_MainTex, IN.texcoord) + _TextureSampleAdd) * IN.color;  
  
    #ifdef UNITY_UI_CLIP_RECT  
    // Soft mask aware clipping  
    half2 m = saturate((_ClipRect.zw - _ClipRect.xy - abs(IN.mask.xy)) * IN.mask.zw);  
    color.a *= m.x * m.y;  
    #endif
```

In Unity 2020.1 or later, a function to blur the edge of the clipping area called soft mask has been implemented and can be easily used.

## Properties of RectMask2D

### rectTransform

```
public RectTransform rectTransform { get; }
```

Get the cache of [RectTransform](#).

### canvasRect

```
public Rect canvasRect { get; }
```

Get the [RectTransform](#) transformed to the coordinates in the [Canvas](#).

### padding

```
public Vector4 padding { get; set; }
```

Gets/Sets the padding of the clipping area.

This property was introduced in Unity 2020.1.

In [Vector4](#), [x](#) is left, [y](#) is down, [z](#) is right, and [w](#) is up. Negative values make the area larger, while positive values make the area smaller. This padding also affects the ray casting decision.

### softness

```
public Vector2Int softness { get; set; }
```

Gets/Sets the size of the horizontal and vertical soft masks.

This property was introduced in Unity 2020.1.

The larger this value is, the more gradually the edges of the clipping area become transparent.

## Public methods of RectMask2D

### AddClippable()

```
public void AddClippable(IClippable clippable);
```

Register the child UI elements to be clipped with this [RectMask2D](#).

This method is called from [OnEnable\(\)](#), etc. of the target UI element.

The UI element to be clipped is either an object that implements the [IClippable](#) interface or a [MaskableGraphic](#). [MaskableGraphic](#) also implements the [IClippable](#) interface, but [RectMask2D](#) manages them separately (to determine if the canvas has moved by using [Graphic.canvasRenderer.hasMoved](#) and to recalculate the clipping area if it has). [MaskableGraphic](#) also implements the [IClippable](#) interface, but [RectMask2D](#) manages them separately (because it uses [Graphic.CanvasRenderer.hasMoved](#) to determine if the canvas has moved, and recalculates the clipping area if it has).

### RemoveClippable()

```
public void RemoveClippable(IClippable clippable);
```

Deletes the child UI element of the clipping target registered by [AddClippable\(\)](#).

### IsRaycastLocationValid()

```
public virtual bool IsRaycastLocationValid(Vector2 sp, Camera eventCamera);
```

Make a raycast decision.

Returns [true](#) if the raycast hit at the position of [sp](#) given in the argument.

This method is an implementation of the [ICanvasRaycastFilter](#) interface.

If the [GameObject](#) is active and the component is [enabled](#), [RectTransformUtility.RectangleContainsScreenPoint\(\)](#) is called to return whether the given position is within the [RectTransform](#). If the clipping area becomes larger or smaller due to padding, the raycast area will also change.

### PerformClipping()

```
public virtual void PerformClipping();
```

Calculates the clipping area of the child UI element to be clipped.

This method is called at the beginning of the Graphic rebuild, immediately after the Layout rebuild in [CanvasUpdateRegistry.PerformUpdate\(\)](#) has completed. Here, the clipping areas of [MaskableGraphic](#) and [IClippable](#) registered in [AddClippable\(\)](#) are calculated and passed to the shader (along with the size of the soft mask), and a [Canvas](#) rebuild is requested.

### UpdateClipSoftness()

```
public virtual void UpdateClipSoftness();
```

Apply the parameters of the soft mask.

In the case of [MaskableGraphic](#), a parameter is set in [CanvasRenderer.clippingSoftness](#), and that parameter is eventually passed to the shader.

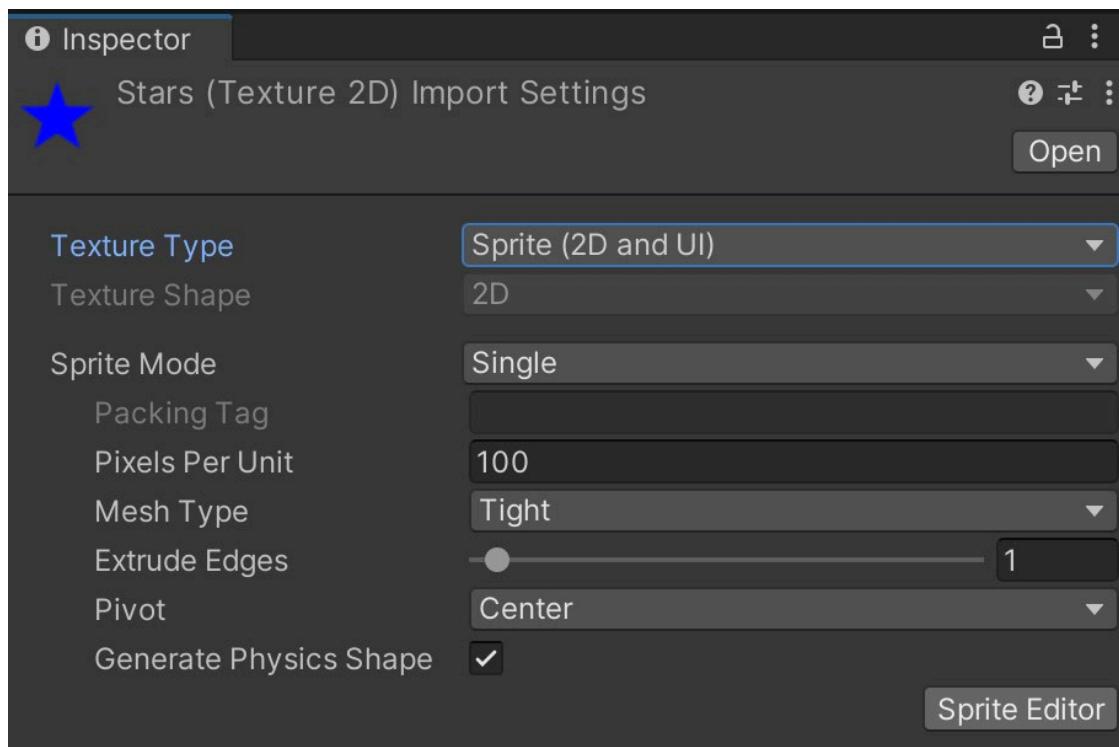
Make the transparent part of the Image unresponsive to touch.

By default, the [Image](#) raycast judgment is also valid for transparent areas. In other words, even if you touch a transparent area, it will respond.

As explained in the [alphaHitTestMinimumThreshold](#) section of the [Image](#) section, if [alphaHitTestMinimumThreshold](#) is set to 1, [texture read/write enabled](#), and atlases are disabled, transparent areas will not react. However, this will double the amount of texture memory used.

In order to make transparent areas unresponsive to touch, let's try using [Sprite](#)'s **Custom Physics Shape**, which allows [Sprite](#) meshes to be hollowed out with images. If we use this mesh to determine ray casting, the transparent area will not react to touch.

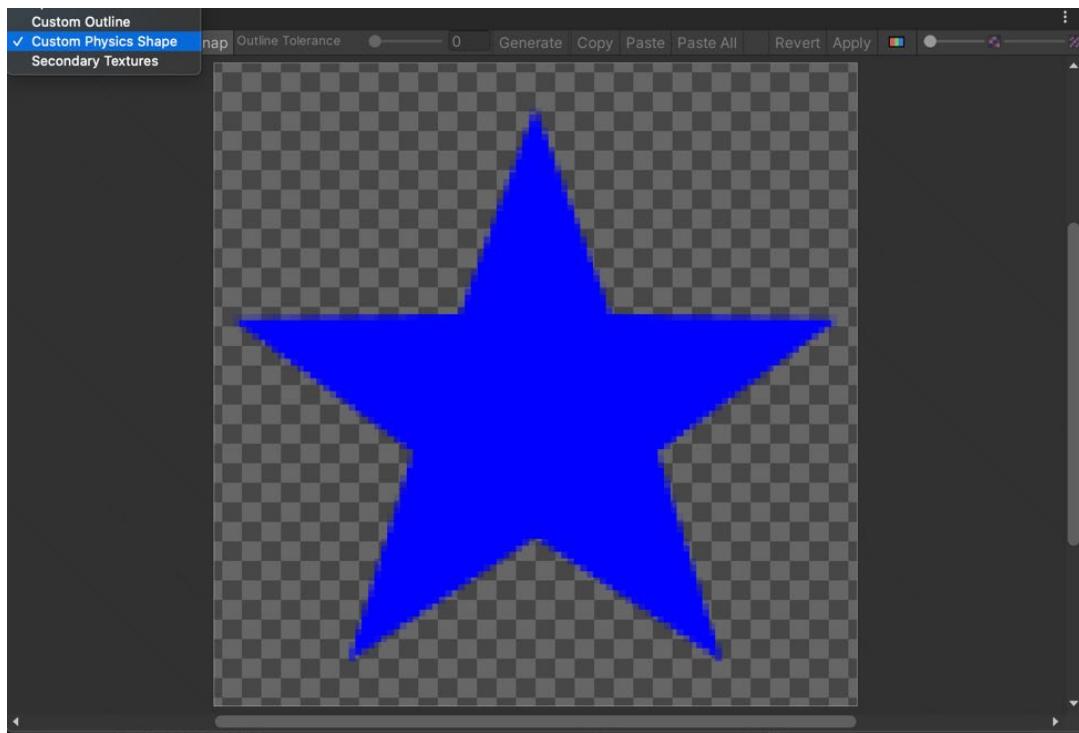
First, create a Physics Shape, make sure the Texture Type is set to **Sprite (2D and UI)** in the Texture **Import Settings**, and click the **Sprite Editor** button.



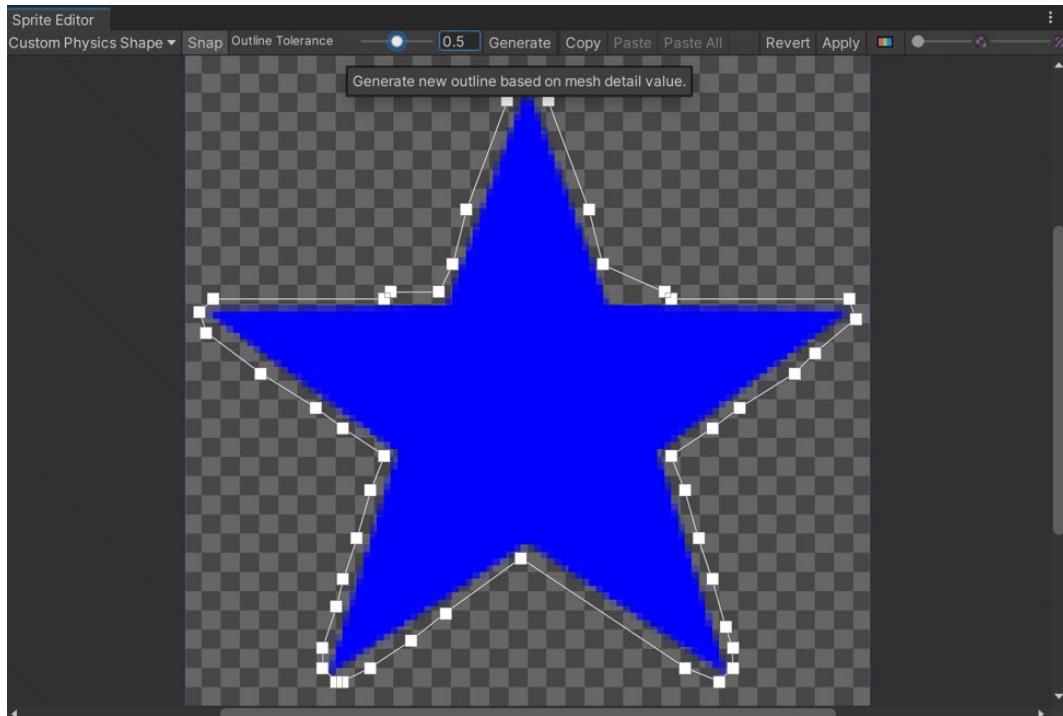
Note

If you don't have the **Sprite Editor** installed, install the **2D Sprite** package from the **Package Manager**.

In the upper left corner, open the drop-down menu labeled "**Sprite Editor**" and select "**Custom Physics Shape**".



Fine-tune the **Outline Tolerance** and press the **Generate** button.



Finally, click the **Apply** button in the upper right corner. The Physics Shape has now been created.

The next step is to use the Physics Shape to determine the raycast location. You can use [PolygonCollider2D](#) to determine if the touched point is inside the Physics Shape.

If the [Sprite](#) has a Custom Physics Shape, it will not respond to touches on transparent areas.

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

// Image that performs Raycast judgment in the form of Sprite's Custom Physics Shaper
[RequireComponent(typeof(PolygonCollider2D))]
public class ShapeRaycastImage : Image
{
    private PolygonCollider2D collider2d;

    protected override void OnEnable()
```

```

{
    base.OnEnable();

    AdjustCollider();

    AdjustCollider(); RegisterDirtyVerticesCallback(AdjustCollider);
}

protected override void OnDisable()
{
    UnregisterDirtyVerticesCallback(AdjustCollider);
    base.OnDisable();
}

public void AdjustCollider()
{
    if (overrideSprite == null)
    {
        return;
    }

    // Create PolygonCollider2D from Sprite's Custom Physics Shaper
    collider2d = GetComponent<PolygonCollider2D>();
    if (collider2d == null)
    {
        collider2d = gameObject.AddComponent<PolygonCollider2D>();
    }

    collider2d.isTrigger = true;
    collider2d.pathCount = overrideSprite.GetPhysicsShapeCount();

    for (int i = 0; i < collider2d.pathCount; i++)
    {
        List<Vector2> physicsShape = new List<Vector2>();
        int pointCount = overrideSprite.GetPhysicsShape(i, physicsShape);

        Vector2[] points = new Vector2[pointCount];
        for (int j = 0; j < points.Length; j++)
        {
            float x = (physicsShape[j].x / overrideSprite.rect.width * overrideSprite.pixelsPerUnit
+ 0.5f - rectTransform.pivot.x) * rectTransform.rect.width;
            float y = (physicsShape[j].y / overrideSprite.rect.height * overrideSprite.pixelsPerUnit

```

```

+ 0.5f - rectTransform.pivot.y) * rectTransform.rect.height;

        points[j] = new Vector2(x, y);
    }

    collider2d.SetPath(i, points);
}
}

// Return true if the RectTransform coordinates overlap PolygonCollider2D.
public bool IsRectTransformPointOverLapped(Vector2 local)
{
    Vector2 point = new Vector2(transform.position.x + local.x * rectTransform.lossyScale.x, transform.position.y + local.y * rectTransform.lossyScale.y);

    return collider2d.OverlapPoint(point);
}

public override bool IsRaycastLocationValid(Vector2 screenPoint, Camera eventCamera)
{
    if (overrideSprite == null)
    {
        return true;
    }

    Vector2 local;
    if (!RectTransformUtility.ScreenPointToLocalPointInRectangle(rectTransform, screenPoint, eventCamera, out local))
    {
        return false;
    }

    // determine if PolygonCollider2D
    if (IsRectTransformPointOverLapped(local))
    {
        return true;
    }

    return false;
}
}

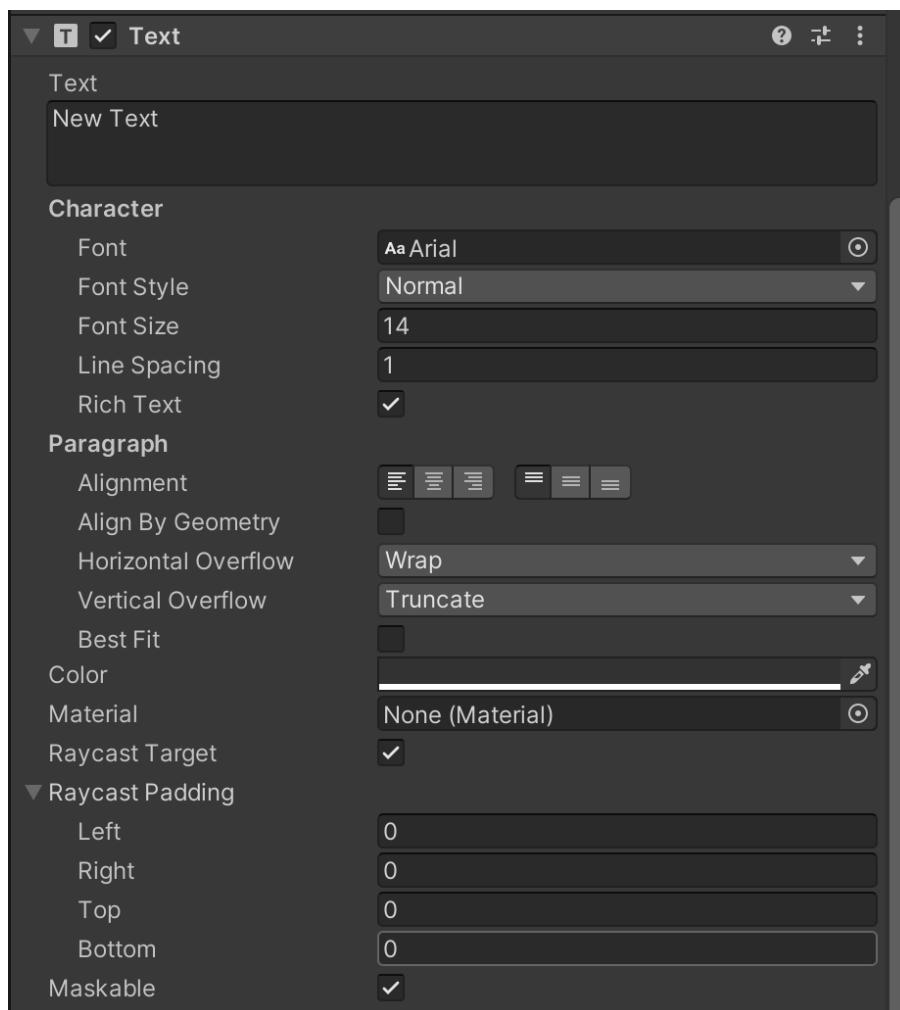
```

## Chapter 6 Text and Fonts

The simplest way to display text in uGUI is to use the [Text](#) component. However, it is troublesome to adjust the spacing between characters and to express decorations with [Text](#) alone. TextMesh Pro (TMP) provides various functions to realize gorgeous text expressions.

However, TextMesh Pro is not always a one-size-fits-all solution, and in some cases it is better to use the [Text](#) component. In this chapter, we will explore the [Text](#) component, font assets, and TextMesh Pro.

## Text component



```
[RequireComponent(typeof(CanvasRenderer))]
[AddComponentMenu("UI/Text", 10)]
public class Text : MaskableGraphic, ILayoutElement
```

The [Text](#) component is a component for displaying rasterized characters in the UI, and it inherits from the [Graphic](#) class via the [MaskableGraphic](#) class, just like [Image](#).

[Text](#) is the first component to be used when displaying text in uGUI, but from a performance point of view, it is a very tricky component: [Text](#) renders one rectangle for each character, but these rectangles have so many transparent parts that they can unintentionally interrupt a batch of Canvas.

Rebuilding the mesh for the text is also a major problem; when the [Text](#) component is changed, the polygons used to display the text need to be recalculated, resulting in a Graphic rebuild. As with other UI components, it is important to reduce the frequency of Graphic rebuilds.

## Properties of Text

text

```
public virtual string text { get; set; }
```

Gets/Sets the value of the string to be displayed.

Definitions are given below.

*UnityEngine.UI/UI/Core/Text.cs*

```
public virtual string text
{
    get
    {
        return m_Text;
    }
    set
    {
        if (String.IsNullOrEmpty(value))
        {
            if (String.IsNullOrEmpty(m_Text))
                return;
            m_Text = "";
            SetVerticesDirty();
        }
        else if (m_Text != value)
        {
            m_Text = value;
            SetVerticesDirty();
            SetLayoutDirty();
        }
    }
}
```

`SetLayoutDirty()` and `Graphic.VerticesDirty()` are called when the string is changed, resulting in Layout rebuild and Graphic rebuild. This is unavoidable. `SetVerticesDirty()` is usually called when

trying to empty a string, but if you don't want to display the string, it is slightly better to set `localScale` to `(0, 0, 0)`.

## font

---

```
public Font font { get; set; }
```

Gets/Sets the font assets to be used for the text. The font assets are described later.

## fontStyle

---

```
public FontStyle fontStyle { get; set; }
```

Gets/Sets the font style used in the text, `FontStyle` is defined as follows

```
public enum FontStyle
{
    // No particular style specified
    Normal,
    // Bold
    Bold,
    // Italic
    Italic,
    // Bold and Italic
    BoldAndItalic
}
```

If a font for Bold or Italic is available in the relevant font (**References to other fonts in project** in the *Import Settings* of the font asset), it will be used. If you don't like the quality of the bold or italic fonts processed by Unity, you may want to get a separate font.

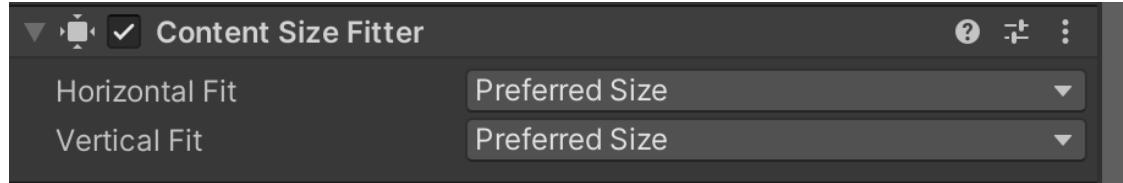
## fontSize

```
public int fontSize { get; set; }
```

Gets/Sets the size of the font used in the text.

The unit is pt (points). If the font size is larger than the [RectTransform](#) of the [Text](#), the text will not be displayed (actually, it will always be displayed if the [horizontalOverflow](#) or [verticalOverflow](#) is [Overflow](#)). (In fact, if [horizontalOverflow](#) or [verticalOverflow](#) is [Overflow](#), the text will always be displayed.) The problem is that the size of the area needed to render the [Text](#) depends on the [scaleFactor](#) of the [Canvas](#). The [scaleFactor](#) of the [Canvas](#) in Screen Space is calculated based on the resolution, so even if the size is correct in the Editor, the text may not be displayed because the [RectTransform](#) size is slightly insufficient.

The appropriate [RectTransform](#) size based on the font size can be obtained from [preferredWidth](#) and [preferredHeight](#). If you want to adjust the [RectTransform](#) size automatically, the easiest way is to attach the [ContentSizeFitter](#) component and set **Horizontal Fit** and **Vertical Fit** to **Preferred Size**.



This [fontSize](#) is the font size of the font texture. Therefore, by setting this size to twice the expected size and setting [RectTransform](#)'s [scale](#) to [\(0.5f, 0.5f, 1.0f\)](#), smooth text can be displayed. On the other hand, if the font size is halved and [scale](#) is set to [\(2, 2, 1\)](#), the text will be blurred but the font texture can be saved. This technique can also be used to save font textures. This technique should be used after considering the balance between appearance and texture memory usage.

## lineSpacing

```
public float lineSpacing { get; set; }
```

Gets/Sets the width of the line spacing.

As for how many pixels are actually left between lines, that depends on the font.

## supportRichText

---

```
public bool supportRichText { get; set; }
```

Enable if you want to support rich text. For an explanation of rich text itself, please read the official documentation.

<https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/StyledText.html>

Note that this property is enabled by default. If rich text is enabled for [Text](#) that can be entered arbitrarily by the player (for example, player name or chat text), it will cause unintended behavior such as changing the color or size. It is recommended to create a preset in [Preset Manager](#).

*Preset descriptions in the Unity User Manual:*

<https://docs.unity3d.com/Manual/Presets.html>

## alignment

---

```
public TextAnchor alignment { get; set; }
```

Gets or sets the relative position of the text with respect to [RectTransform](#).

[TextAnchor](#) can be used to specify up/down and left/center/right.

The definition of [TextAnchor](#) is as follows:

```
namespace UnityEngine
{
    public enum TextAnchor
    {
        UpperLeft,
        UpperCenter,
        UpperRight,
        MiddleLeft,
        MiddleCenter,
        MiddleRight,
        LowerLeft,
        LowerCenter,
```

```
        LowerRight  
    }  
}
```

## alignByGeometry

```
public bool alignByGeometry { get; set; }
```

Align top, bottom, left and right using the geometry of the mesh for the text, not the metrics information of the font (baseline, height, etc.).

If this property is set to `true`, the `RectTransform` borders and the character mesh will match.

## horizontalOverflow

```
public HorizontalWrapMode horizontalOverflow { get; set; }
```

Gets/Sets the behavior when the width of the text is larger than the width of `RectTransform`.

The definition of `HorizontalWrapMode` is as follows.

```
namespace UnityEngine  
{  
    // Behavior when a horizontal boundary is reached  
    public enum HorizontalWrapMode  
    {  
        Wrap,  
        Overflow  
    }  
}
```

The default value is `Overflow`, which means that the text will overflow. On the other hand, `Wrap` means that the text will wrap at the end of the line.

## verticalOverflow

```
public VerticalWrapMode verticalOverflow { get; set; }
```

Gets/Sets the behavior when the height of the text is greater than the [RectTransform](#) height.

```
namespace UnityEngine
{
    // Behavior when a vertical boundary is reached
    public enum VerticalWrapMode
    {
        Truncate,
        Overflow
    }
}
```

The default is [Truncate](#), which means it will be truncated, and [Overflow](#) means it will overflow.

## resizeTextForBestFit

```
public bool resizeTextForBestFit { get; set; }
```

When this property is set to [true](#), the font size will be automatically changed to the largest integer value that will fit the text in the [RectTransform](#) area, which is displayed as **Best Fit** in [Inspector](#).

The default value is [false](#).

In general, this Best Fit setting should not be used. The reason is that it may increase the memory usage of the font texture. Font textures store characters for each font size, so if you use a font of a different size, new characters will be stored in the font texture. This increases the size of the font texture, which increases memory usage. Not only that, it may also cause memory fragmentation. In addition, CPU time is consumed due to repeated calculations to calculate the optimal font size.

One of the few situations where this setting is effective is when you want to fit text into a specific area of a multilingual game. Normally, you should specify the appropriate font size beforehand, but

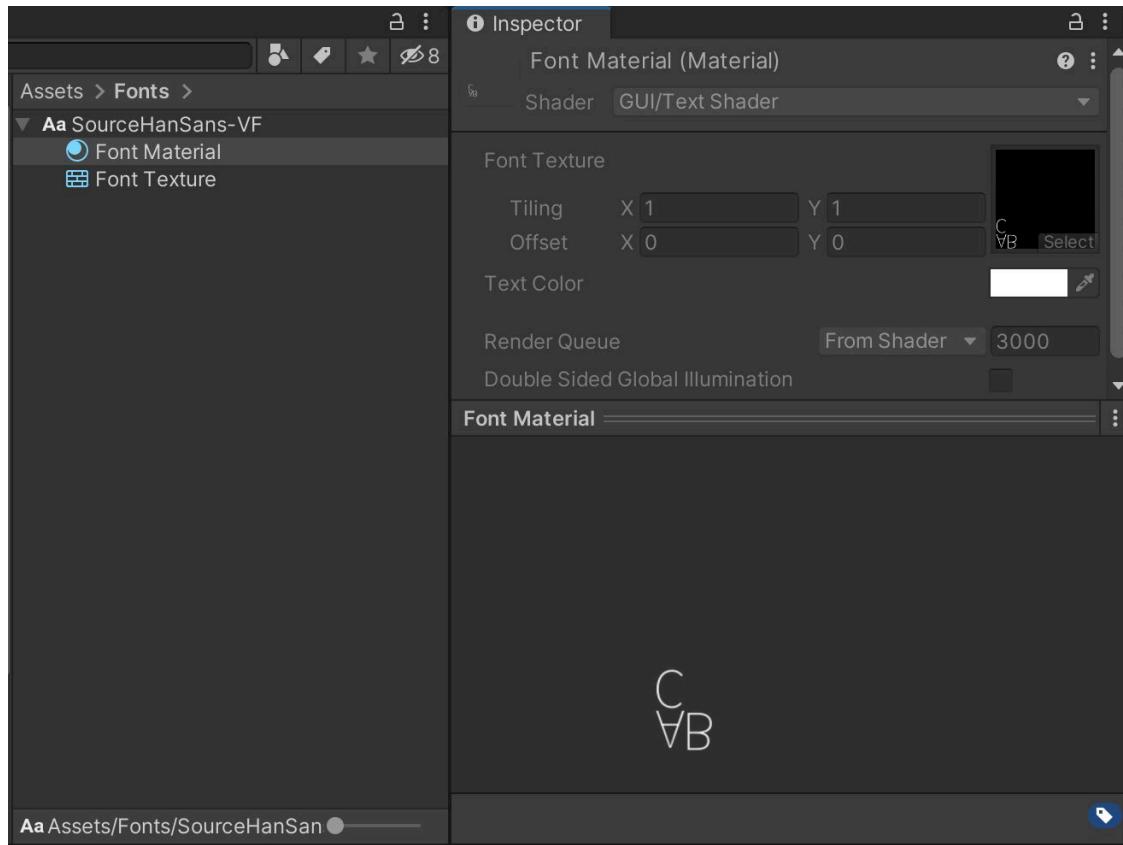
this can be difficult when the number of languages supported or the amount of text is large. It is not a bad idea to keep the Best Fit feature in mind as a way to deal with such special cases.

### mainTexture

```
public override Texture mainTexture { get; }
```

Get the font texture used in this [Text](#).

For example, if you set this font texture as the [RawImage](#) texture, you can visually check the font texture that is currently in use. The same thing can be seen in the [Font Material](#) preview of the font asset.



## resizeTextMinSize

---

```
public int resizeTextMinSize { get; set; }
```

Gets/Sets the minimum font size when Best Fit is enabled.

The default value is 0.

It is displayed as **Min Size** in **Inspector**.

If the font size does not fit in the area with this value, a part of the string will be hidden.

## resizeTextMaxSize

---

```
public int resizeTextMaxSize { get; set; }
```

Gets/Sets the maximum font size when Best Fit is enabled.

The default value is 100.

It is displayed as **Max Size** in **Inspector**.

If this property is set to 1, the value of `fontSize` itself will be set; if it is set to a value smaller than `fontSize`, the value of `fontSize` will be set.

## pixelsPerUnit

---

```
public float pixelsPerUnit { get; }
```

Get the magnification factor of how many points to the screen the font will be displayed.

It will be easier to understand if you actually look at the definition.

*Packages/com.unity.ugui/Runtime/UI/Core/Text.cs*

```
/// <summary>
/// Provides information about the scale of the font relative to the screen
/// </summary>
/// <remarks>
/// For dynamic fonts, this value will be the same as the Canvas scale factor. Otherwise, it is calculated from the size of the text and the size of the font.
/// </remarks>
public float pixelsPerUnit
{
    get
    {
        var localCanvas = canvas;
        if (!localCanvas)
            return 1;
        // For dynamic fonts, ensure that the screen is 1 pixel for 1 pixel.
        if (!font || font.dynamic)
            return localCanvas.scaleFactor;
        // For non-dynamic fonts, calculate pixels per unit based on the ratio of the specified font size to the font object's font size.
        if (m_FontData.fontSize <= 0 || font.fontSize <= 0)
            return 1;
        return font.fontSize / (float)m_FontData.fontSize;
    }
}
```

## cachedTextGenerator

---

```
public TextGenerator cachedTextGenerator { get; }
```

Get the cache of the [TextGenerator](#) used for rendering characters.

The [TextGenerator](#) class is described later in this document.

## cachedTextGeneratorForLayout

---

```
public TextGenerator cachedTextGeneratorForLayout { get; }
```

Get the cache of the [TextGenerator](#) used to determine the Layout.

minWidth

---

```
public virtual float minWidth { get; }
```

Get the minimum width to be used for Auto Layout.

Always returns [0](#). The details are explained in *Chapter 9 Auto Layout*.

minHeight

---

```
public virtual float minHeight { get; }
```

Get the minimum height to be used for Auto Layout.

Always returns [0](#). The details are explained in *Chapter 9 Auto Layout*.

flexibleWidth

---

```
public virtual float flexibleWidth { get; }
```

Get the flexible width used for Auto Layout.

It always returns [-1](#), so the width of flexible will be ignored. The details are explained in *Chapter 9 Auto Layout*.

flexibleHeight

---

```
public virtual float flexibleHeight { get; }
```

Get the flexible height used for Auto Layout.

Since it always returns -1, the height of the flexible will be ignored. The details are explained in [Chapter 9 Auto Layout](#).

## preferredWidth

---

```
public virtual float preferredWidth { get; }
```

Get the width that is sufficient to display all [Text](#), which is used for Auto Layout.

[GetPreferredWidth\(\)](#) divided by [pixelsPerUnit](#), and returns the result.

The details of how [preferredWidth](#) is used are explained in [Chapter 9 Auto Layout](#).

## preferredHeight

---

```
public virtual float preferredHeight { get; }
```

Get the height that is sufficient to display all [Text](#), which is used for Auto Layout.

[GetPreferredHeight\(\)](#) divided by [pixelsPerUnit](#) to return the result.

The details of how [preferredWidth](#) is used are explained in [Chapter 9 Auto Layout](#).

## Static methods of Text

### GetTextAnchorPivot

```
public static Vector2 GetTextAnchorPivot(TextAnchor anchor);
```

When a `TextAnchor` is passed, the corresponding `Vector2` is returned.

This will be easier to understand if you actually look at the code.

*Packages/com.unity.ugui/Runtime/UI/Core/Text.cs*

```
static public Vector2 GetTextAnchorPivot(TextAnchor anchor)
{
    switch (anchor)
    {
        case TextAnchor.MiddleLeft: return new Vector2(0, 0);
        case TextAnchor.LowerCenter: return new Vector2(0.5f, 0);
        case TextAnchor.LowerRight: return new Vector2(1, 0);
        case TextAnchor.MiddleLeft: return new Vector2(0, 0.5f);
        case TextAnchor.MiddleCenter: return new Vector2(0.5f, 0.5f);
        case TextAnchor.MiddleRight: return new Vector2(1, 0.5f);
        case TextAnchor.UpperLeft: return new Vector2(0, 1);
        case TextAnchor.UpperCenter: return new Vector2(0.5f, 1);
        case TextAnchor.UpperRight: return new Vector2(1, 1);
        default: return Vector2.zero;
    }
}
```

## Public methods of Text

### FontTextureChanged

```
public void FontTextureChanged();
```

Called when the font texture is changed.

In fact, the [FontUpdateTracker](#) class sets itself as a callback to [Font.textureRebuilt](#) and calls [FontTextureChanged\(\)](#) on all [Text](#) components when the callback is called.

In [FontTextureChanged\(\)](#), the [cachedTextGenerator](#) is reset, and [SetAllDirty\(\)](#) is called, resulting in a Graphic rebuild and a Layout rebuild.

### GetGenerationSettings

```
public TextGenerationSettings GetGenerationSettings(Vector2 extents);
```

Returns an instance of the [TextGenerationSettings](#) class that contains the current [Text](#) settings.

This method is useful for writing text generation settings. The [TextGenerationSettings](#) obtained here can be passed to the [TextGenerator](#).

### CalculateLayoutInputHorizontal

```
public virtual void CalculateLayoutInputHorizontal();
```

This method is an implementation of [CalculateLayoutInputHorizontal\(\)](#) of the [ILayoutElement](#) interface, but it is empty.

### CalculateLayoutInputVertical

```
public virtual void CalculateLayoutInputVertical();
```

This method is an implementation of `CalculateLayoutInputVertical()` of the `ILayoutElement` interface, but it is empty.

## The TextGenerator class

```
[StructLayout(LayoutKind.Sequential)]
[UsedByNativeCode]
[NativeHeader("Modules/TextRendering/TextGenerator.h")]
public sealed class TextGenerator : IDisposable
```

TextGenerator is a class used by Text to render text.

The TextGenerator is used to generate the vertex data corresponding to the string, etc., and the result is used to flow from Text.OnPopulateMesh() to the vertex to be drawn via VertexHelper.

TextGenerator is a class that belongs to the UnityEngine namespace, but some C# code is publicly available to help you understand it.

<https://github.com/Unity-Technologies/UnityCsReference/blob/master/Modules/TextRendering/TextGenerator.cs>

The actual code to get the text rendering information using TextGenerator is as follows.

```
public void TextGeneratorSample()
{
    // Load the default font
    Font font = Resources.GetBuiltinResource<Font>("Arial.ttf");

    // The string we want to render
    string targetString = "A" + System.Environment.NewLine + "BC";

    // Settings to pass to the TextGenerator
    TextGenerationSettings settings = new TextGenerationSettings();

    // Display area
    settings.generationExtents = new Vector2(100.0f, 100.0f);

    // Update the display area if horizontalOverflow and verticalOverflow are Overflow
    settings.updateBounds = true;

    // Pivot the generated vertices
    settings.pivot = new Vector2(0.5f, 0.5f);
```

```

// usually pass canvasScaler's scaleFactor
settings.scaleFactor = 1;

// the following is the same as setting the Text component
settings.font = font;
settings.FontStyle = FontStyle.Normal;
settings.fontSize = 20;
settings.lineSpacing = 1;
settings.richText = false;
settings.textAnchor = TextAnchor.MiddleCenter;
settings.alignByGeometry = false;
settings.horizontalOverflow = HorizontalWrapMode.Overflow;
settings.verticalOverflow = VerticalWrapMode.Overflow;
settings.resizeTextForBestFit = true;
settings.resizeTextMinSize = 1;
settings.resizeTextMaxSize = 100;
settings.color = Color.white;

// Generate data for rendering by receiving string and settings.
// Pass the length of the string to the constructor to avoid unnecessary memory allocation (default is 50 characters)
TextGenerator generator = new TextGenerator(targetString.Length);

bool result = generator.PopulateWithErrors(targetString, settings, this.gameObject);
Debug.LogFormat("Rendering succeeded {0}", result);
Debug.LogFormat("Generated text area size {0}", generator.rectExtents);
Debug.LogFormat("Generated character count {0}", generator.characterCount);
Debug.LogFormat("Generated number of visible characters {0}", generator.characterCountVisible);
Debug.LogFormat("Font size in case of BestFit {0}", generator.fontSizeUsedForBestFit);
Debug.LogFormat("Number of lines generated {0}", generator.lineCount);
Debug.LogFormat("Number of generated vertices {0}", generator.vertexCount);

Debug.Log("Generated character array");
for (int i = 0; i < generator.characters.Count; i++)
{
    Debug.LogFormat("  Cursor position of {0}th character {1}", i + 1, generator.characters[i].cursorPos);
    Debug.LogFormat("  {0}th character width {1}", i + 1, generator.characters[i].charWidth);
}

```

```
Debug.Log("Information on each generated line");
for (int i = 0; i < generator.lines.Count; i++)
{
    Debug.LogFormat(" {0} Index of the first character of the first line {1} ", i + 1, generator.lines[i].StartCharIdx);
    Debug.LogFormat(" {0} height of the first line {1} ", i + 1, generator.lines[i].height);
    Debug.LogFormat(" {0} Y position above line {1} ", i + 1, generator.lines[i].topY);
    Debug.LogFormat(" {0} number of pixels between the line and the next line {1} ", i + 1, generator.lines[i].leading);
}

Debug.LogFormat("Generated vertex array {0}", generator.verts);
for (int i = 0; i < generator.verts.Count; i++)
{
    Debug.LogFormat(" {0}th vertex position {1} ", i + 1, generator.verts[i].Position);
    Debug.LogFormat(" UV0 of {0}-th vertex {1} ", i + 1, generator.verts[i].uv0);
}
```

## Properties of TextGenerator

### characterCount

```
public int characterCount { get; }
```

Get the number of characters generated as a result of calling [Populate\(\)](#) or [PopulateWithErrors\(\)](#).

Note that this includes newlines and other characters that are not displayed.

### characterCountVisible

```
public int characterCountVisible => characterCount - 1;
```

The number of characters generated as a result of calling [Populate\(\)](#) or [PopulateWithErrors\(\)](#) that can be displayed.

characterCount - 1 is returned.

### characters

```
public IList<UICharInfo> characters { get; }
```

Gets a list of character information ([UICharInfo](#) structure) generated as a result of calling [Populate\(\)](#) or [PopulateWithErrors\(\)](#). Note that the list includes characters that are not displayed, such as newlines.

The definition of the [UICharInfo](#) structure is as follows.

```
using UnityEngine.Scripting;  
  
namespace UnityEngine.  
{  
    // Renderable character information
```

```
[UsedByNativeCode]
public struct UICharInfo
{
    // Position of each character cursor in the local space (where the text is generated)
    public Vector2 cursorPos;

    // width of the character
    public float charWidth;
}
```

## fontSizeUsedForBestFit

```
public int fontSizeUsedForBestFit { get; }
```

Gets the adjusted font size if Best Fit is enabled for the text generated as a result of calling [Populate\(\)](#) or [PopulateWithErrors\(\)](#).

If there is no change in the font size, the value of the font size passed in the settings will be returned.

## lineCount

```
public int lineCount { get; }
```

Get the number of lines of text generated as a result of calling [Populate\(\)](#) or [PopulateWithErrors\(\)](#).

## lines

```
public IList<UILineInfo> lines { get; }
```

Gets a list of information ([UILineInfo](#) structure) for each line of text generated as a result of calling [Populate\(\)](#) or [PopulateWithErrors\(\)](#).

The definition of the [UILineInfo](#) structure is as follows.

```
using UnityEngine.Scripting;

namespace UnityEngine.
{
    // Information for each line of the generated text
    [UsedByNativeCode]
    public struct UILineInfo
    {
        // Index of the first character of the line
        public int startCharIdx;

        // height of the line
        public int height;

        // Y-coordinate pixels above the line
        // Used to annotate InputField caret, selection boxes, etc.
        public float topY;

        // the number of pixels of space between this line and the next line
        public float leading;
    }
}
```

rectExtents

```
public Rect rectExtents { get; }
```

Get the text area generated as a result of calling `Populate()` or `PopulateWithErrors()`.

vertexCount

```
public int vertexCount { get; }
```

Get the number of vertices generated as a result of calling `Populate()` or `PopulateWithErrors()`.

verts

---

```
public IList<UIVertex> verts { get; }
```

An array of vertices ([UIVertex](#)) generated as a result of calling [Populate\(\)](#) or [PopulateWithErrors\(\)](#).

## Public methods of TextGenerator's

### Populate

```
public bool Populate(string str, TextGenerationSettings settings);
```

Generate vertices and other data based on the given string and settings.

The generated data is cached, and if a call is made with the same string and the same settings as the previous time, the cache is returned.

### PopulateWithErrors

```
public bool PopulateWithErrors(string str, TextGenerationSettings settings, GameObject context);
```

Similar to `Populate()`, but returns `false` if the data could not be generated, and outputs an error message in the context of the `GameObject` given as an argument.

The type of error is defined as `TextGenerationError`.

```
enum TextGenerationError
{
    // No error (default)
    None = 0,
    // Non-dynamic font requested a different size than the original
    CustomSizeOnNonDynamicFont = 1,
    // custom style requested for a non-dynamic font
    CustomStyleOnNonDynamicFont = 2,
    // font does not exist
    NoFont = 4
}
```

What I mean by "GameObject context" is that the Debug.Log() error output method has an overload that passes a GameObject.

```
public static void Log(object message);
public static void Log(object message, Object context);
```

If you pass a GameObject as context, clicking on the error message displayed in the **Console** will highlight the object in the **Hierarchy**.

## Invalidate

```
public void Invalidate();
```

Mark as invalid the data generated as a result of calling [Populate\(\)](#) or [PopulateWithErrors\(\)](#) (but do not destroy the cached data).

The next time it is called, the data will be generated even if it is the same string and the same settings as the previous time.

## GetCharacters

```
public void GetCharacters(List<UICharInfo> characters);
```

It stores and returns the [UICharInfo](#) list generated as a result of calling [Populate\(\)](#) or [PopulateWithErrors\(\)](#) in the passed list.

Even if the cache inside the [TextGenerator](#) is valid, the generation process will be performed. Also, if the cache is invalid when trying to get [characters](#), this method will be called.

## GetCharactersArray

```
public UICharInfo[] GetCharactersArray();
```

Returns an array of [UICharInfo](#) generated as a result of calling [Populate\(\)](#) or [PopulateWithErrors\(\)](#).

Note that the data inside is the same as that returned by `GetCharacters()`, but calling this method will result in a ***GC Alloc***.

### GetLines

---

```
public void GetLines(List<UILineInfo> lines);
```

Generates information for each line by passing `List<UILineInfo>` and stores the result in the passed list.

If the cache is invalid when trying to retrieve `lines`, this method will be called.

### GetLinesArray

---

```
public UILineInfo[] GetLinesArray();
```

Generates information for each row and returns an array of `UILineInfo` as a result.

The data inside is the same as that returned by `GetLines()`; note that, as with `GetCharArray()`, calling this method will result in ***GC Alloc***.

### GetPreferredWidth

---

```
public float GetPreferredWidth(string str, TextGenerationSettings settings);
```

Calculates and returns the width of the display area based on the string and settings passed in.

In practice, we set `Overflow` to `horizontalOverflow` and `verticalOverflow`, override `updateBounds` to `true`, call `Populate()`, and return the result of dividing the `width` of the resulting `rectExtents` by `pixelsPerUnit`. The result is used in the `preferredWidth()` of the `Text` component.

### GetPreferredHeight

---

```
public float GetPreferredHeight(string str, TextGenerationSettings settings);
```

Calculates and returns the height of the display area based on the passed string and settings.

In practice, `Populate()` is called with `verticalOverflow` set to `Overflow` and `updateBounds` set to `true`, and returns the result of dividing the `height` of the resulting `rectExtents` by `pixelsPerUnit`. It is used in the `preferredHeight()` of the `Text` component.

### GetVertices

---

```
public void GetVertices(List<UIVertex> vertices);
```

Generates vertex information by passing `List<UIVertex>` and stores the result in the passed list.

This method will be called if the cache is invalid when trying to get the `verts`.

### GetVerticesArray

---

```
public UIVertex[] GetVerticesArray();
```

Generates vertex information and returns an array of `UIVertex` as a result.

The data inside is the same as that returned by `GetVertices()`. Note that calling this method will result in a **GC Alloc**.

## Use TextGenerator to abbreviate overflowed characters with ellipsis (...)

The `TextGenerator` is useful when you want to fit a string into a specific size, but want to express the rest of the string as *ellipsis like "..."* when it overflows. The code to do this is shown below.

```
using UnityEngine;
using UnityEngine.UI;

[RequireComponent(typeof(Text))]
public class ApplyTextEllipsis : MonoBehaviour
{
    private RectTransform rectTransformCache;

    private Text textCache;

    private readonly string ellipsisString = "...";

    private void Start()
    {
        // For performance reasons, we cache RectTransform and Text as member variables
        rectTransformCache = transform as RectTransform;
        textCache = GetComponent<Text>();

        // Make sure we get a callback when the layout changes
        textCache.RegisterDirtyLayoutCallback(Apply);
    }

    // Depending on the order of script execution, the layout of the text may change before the callback is set. Call it once
    Apply();
}

private void OnDestroy()
{
    // Don't forget to release the callback
    textCache.UnregisterDirtyLayoutCallback(Apply);
}

public void Apply()
{
    string str = textCache.text;
```

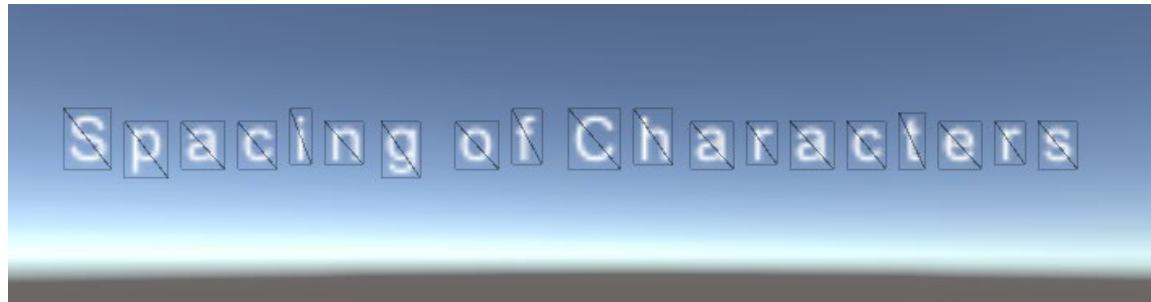
```
// TextGenerator and TextGenerationSettings will be used for the Text layout
TextGenerator generator = textCache.cachedTextGeneratorForLayout;
TextGenerationSettings settings = textCache.GetGenerationSettings(rectTransformCache.re
ct.size);

// Trim the string until the size fits
while (str.Length - ellipsisString.Length > 0)
{
    // Finish when it fits the height of RectTransform
    if (generator.GetPreferredHeight(str, settings) <= rectTransformCache.rect.height)
    {
        break;
    }

    // Add ellipsis and remove one character from the back
    str = str.Substring(0, str.Length - 1 - ellipsisString.Length) + ellipsisString;
}

// Reset the final string to Text
textCache.text = str;
}
```

## Control the spacing of characters



The `Text` component does not allow you to control the horizontal spacing of the characters. This may be the reason why you often use TextMesh Pro.

To put it simply, you can adjust the spacing between characters by adjusting the position of the meshes representing each character. What we actually need to do is as follows.

- Adjust the mesh position for each character with `OnPopulateMesh()`.
- Make sure to return the appropriate `preferredWidth`.
- Handle horizontal alignment as appropriate.
- Handle `horizontalOverflow` and `verticalOverflow` as appropriate.

If we extend the `Text` component to take these into account, it will look like this.

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
#if UNITY_EDITOR
using UnityEditor;
#endif

// Text to adjust the width between characters
public class CustomWidthPaddingText : Text
{
    // padding value between characters
    public float paddingWidth = 0.0f;

    private readonly UIVertex[] m_TempVerts = new UIVertex[4];

    protected override void OnPopulateMesh(VertexHelper toFill)
```

```

{
    if (font == null) return;

    m_DisableFontTextureRebuiltCallback = true;

    Vector2 extents = rectTransform.rect.size;
    string targetText = text;

    var settings = GetGenerationSettings(extents);
    int charCount;
    int lineCount;

    if (horizontalOverflow == HorizontalWrapMode.Overflow)
    {
        // Populate as usual and adjust the horizontal vertex position later
        cachedTextGenerator.PopulateWithErrors(targetText, settings, gameObject);
        charCount = cachedTextGenerator.characterCount;
        lineCount = cachedTextGenerator.lineCount;
    }
    else // if horizontalOverflow is HorizontalWrapMode.
    {
        // Populate once to get the width of the character
        // Both Overflow settings should be Overflow since we'll be breaking lines ourselves
        settings.horizontalOverflow = HorizontalWrapMode.Overflow;
        settings.verticalOverflow = VerticalWrapMode.Overflow;
        cachedTextGenerator.PopulateWithErrors(targetText, settings, gameObject);
        charCount = cachedTextGenerator.characterCount;

        // Use a StringBuilder for performance
        var targetStringBuilder = new System.Text.StringBuilder(text.Length);

        // X coordinate of the current character
        float xpos = 0;

        // insert a new line if it extends horizontally
        for (int i = 0; i < charCount - 1; i++)
        {
            // Does it still fit on this line with this character?
            if (xpos + cachedTextGenerator.characters[i].charWidth <= rectTransform.rect.width)
            {
                targetStringBuilder.Append(text[i]);
                xpos += cachedTextGenerator.characters[i].charWidth;
            }
        }
    }
}

```

```

    }

    else
    {
        // If we insert this character, it won't fit on the line, so we'll insert it after a line break
        targetStringBuilder.Append("\n");
        targetStringBuilder.Append(text[i]);
        xpos = cachedTextGenerator.characters[i].charWidth;
    }

    if (text[i] == '\n')
    {
        // if it's a new line, the X coordinate goes to the beginning
        xpos = 0;
    }
    else
    {
        // If we add more padding, will it still fit on this line?
        if (xpos + paddingWidth <= rectTransform.rect.width)
        {
            xpos += paddingWidth;
        }
        else
        {
            // If we add padding, it won't fit in this line, so we'll add a new line at this point.
            if (i < charCount - 2 && text[i + 1] != '\n')
            {
                targetStringBuilder.Append("\n");
            }
            xpos = 0;
        }
    }
}

// string with line breaks is fixed
targetText = targetStringBuilder.ToString();

// Populate again to get the vertices
settings.verticalOverflow = verticalOverflow;
cachedTextGenerator.PopulateWithErrors(targetText, settings, gameObject);
charCount = cachedTextGenerator.characterCount;
lineCount = cachedTextGenerator.lineCount;
}

```

```

IList<UIVertex> verts = cachedTextGenerator.verts;
float unitsPerPixel = 1 / pixelsPerUnit;
int vertCount = verts.Count;

if (vertCount <= 0)
{
    toFill.Clear();
    return;
}

Vector2 roundingOffset = new Vector2(verts[0].position.x, verts[0].position.y) * unitsPerPixel;
roundingOffset = PixelAdjustPoint(roundingOffset) - roundingOffset;
toFill.Clear();

// padding to be added to each character
float[] charPaddingX = new float[charCount];

// how to add padding depends on horizontal alignment
if (alignment == TextAnchor.UpperLeft || alignment == TextAnchor.MiddleLeft || alignment == TextAnchor.LowerLeft)
{
    int currentLine = 0;
    int paddingCount = 0;
    int paddingIndex = 0;

    // Just add the padding from left to right
    for (int i = 0; i < charCount; i++)
    {
        // Is the line broken?
        if (currentLine + 1 < lineCount && i == cachedTextGenerator.lines[currentLine + 1].startCharIdx)
        {
            paddingCount = 0;
            currentLine++;
        }

        // go to next character if width is 0
        if (Mathf.Approximately(0, cachedTextGenerator.characters[i].charWidth))
        {
            continue;
        }
    }
}

```

```

        }

        charPaddingX[paddingIndex] = paddingCount * paddingWidth;
        paddingIndex++;
        paddingCount++;
    }
}

else if (alignment == TextAnchor.UpperCenter || alignment == TextAnchor.MiddleCenter ||
alignment == TextAnchor.LowerCenter)
{
    int currentLine = 0;
    int paddingIndex = 0;
    int charCountInCurrLine = 0;
    int firstIndexInCurrLine = 0;
    int firstIndexInNextLine = 0;

    for (int i = 0; i < charCount; i++)
    {
        // Is this a new line or the last character?
        if ((currentLine + 1 < lineCount && i == cachedTextGenerator.lines[currentLine + 1].startCharIdx) || i == charCount - 1)
        {
            // add padding to the character on the left
            if (charCountInCurrLine % 2 == 0)
            {
                for (int j = 0; j < charCountInCurrLine / 2; j++)
                {
                    charPaddingX[firstIndexInCurrLine + charCountInCurrLine / 2 - 1 - j] = -(0.5f
+ j) * paddingWidth;
                }

                for (int j = 0; j < charCountInCurrLine / 2; j++)
                {
                    charPaddingX[firstIndexInCurrLine + charCountInCurrLine / 2 + j] = (0.5f + j)
* paddingWidth;
                }
            }
            else // Add padding to the character on the right.
            {
                for (int j = 0; j < charCountInCurrLine / 2; j++)
                {
                    charPaddingX[firstIndexInCurrLine + charCountInCurrLine / 2 - 1 - j] = -(1 + j)
* paddingWidth;
                }
            }
        }
    }
}

```

```

        * paddingWidth;
    }

    for (int j = 0; j < charCountInCurrLine / 2; j++)
    {
        charPaddingX[firstIndexInCurrLine + j + charCountInCurrLine / 2 + 1] = (1 + j
) * paddingWidth;
    }
}

firstIndexInCurrLine = firstIndexInNextLine;
charCountInCurrLine = 0;
currentLine++;
}

// go to the next character if the character width is 0
if (Mathf.Approximately(0, cachedTextGenerator.characters[i].charWidth))
{
    continue;
}

paddingIndex++;
charCountInCurrLine++;
firstIndexInNextLine = paddingIndex;
}

}
else
{
    int currentLine = 0;
    int paddingIndex = 0;
    int firstIndexInCurrLine = 0;

    for (int i = 0; i < charCount; i++)
    {
        // Is this a new line or the last character?
        if ((currentLine + 1 < lineCount && i == cachedTextGenerator.lines[currentLine + 1].
startCharIdx) || i == charCount - 1)
        {
            for (int j = firstIndexInCurrLine; j < paddingIndex; j++)
            {
                charPaddingX[j] = -(paddingIndex - j - 1) * paddingWidth;
            }
        }
    }
}

```

```

        currentLine++;
        firstIndexInCurrLine = paddingIndex;
    }

    // if the width is 0, go to the next character
    if (Mathf.Approximately(0, cachedTextGenerator.characters[i].charWidth))
    {
        continue;
    }

    paddingIndex++;
}
}

if (roundingOffset != Vector2.zero)
{
    int paddingIndex = -1;
    bool addPadding = false;

    for (int i = 0; i < vertCount; ++i)
    {
        int tempVertsIndex = i & 3;
        m_TempVerts[tempVertsIndex] = verts[i];
        m_TempVerts[tempVertsIndex].position *= unitsPerPixel;
        m_TempVerts[tempVertsIndex].position.x += roundingOffset.x;
        m_TempVerts[tempVertsIndex].position.y += roundingOffset.y;

        // if it's a degenerate polygon, don't add padding
        if (i % 4 == 0)
        {
            if (verts[i + 1].position.x - verts[i].position.x > 0)
            {
                paddingIndex++;
                addPadding = true;
            }
            else
            {
                addPadding = false;
            }
        }
    }
}

```

```

// add padding
if (addPadding)
{
    m_TempVerts[tempVertsIndex].position.x += charPaddingX[paddingIndex];
}

if (tempVertsIndex == 3)
    toFill.AddUIVertexQuad(m_TempVerts);
}

else
{
    int paddingIndex = -1;
    bool addPadding = false;

    for (int i = 0; i < vertCount; ++i)
    {
        int tempVertsIndex = i & 3;
        m_TempVerts[tempVertsIndex] = verts[i];
        m_TempVerts[tempVertsIndex].position *= unitsPerPixel;

        // if it's a degenerate polygon, don't add padding
        if (i % 4 == 0)
        {
            if (verts[i + 1].position.x - verts[i].position.x > 0)
            {
                paddingIndex++;
                addPadding = true;
            }
            else
            {
                addPadding = false;
            }
        }

        // add padding
        if (addPadding)
        {
            m_TempVerts[tempVertsIndex].position.x += charPaddingX[paddingIndex];
        }

        if (tempVertsIndex == 3)

```

```

        toFill.AddUIVertexQuad(m_TempVerts);
    }
}

m_DisableFontTextureRebuiltCallback = false;
}

// Area width used by Auto Layout functions such as ContentSizeFitter
public override float preferredWidth
{
get
{
    // Calculate the height by breaking the string on its own
    Rect rect = GetPixelAdjustedRect();
    var settings = GetGenerationSettings(new Vector2(rect.size.x, rect.size.y));
    settings.horizontalOverflow = HorizontalWrapMode.Overflow;

    // Use cachedTextGeneratorForLayout instead of cachedTextGenerator
    cachedTextGenerator.PopulateWithErrors(text, settings, gameObject);
    int charCount = cachedTextGeneratorForLayout.characterCount;

    // Set the preferredWidth to be the largest width of all the lines
    float maxWidth = 0;
    float xpos = 0;

    for (int i = 0; i < text.Length && i < charCount; i++)
    {
        if (text[i] == '\n')
        {
            xpos = 0;
            continue;
        }

        xpos += cachedTextGeneratorForLayout.characters[i].charWidth;

        if (i < text.Length - 1 && text[i + 1] != '\n')
        {
            xpos += paddingWidth;
        }
    }

    maxWidth = Mathf.Max(maxWidth, xpos);
}
}

```

```

        return maxWidth;
    }
}
}

// Display the paddingWidth in the Editor
#if UNITY_EDITOR
[CustomEditor(typeof(CustomWidthPaddingText), true)]
public class CustomWidthPaddingTextEditor : UnityEditor.UI.TextEditor
{
    private SerializedProperty paddingWidthProp;

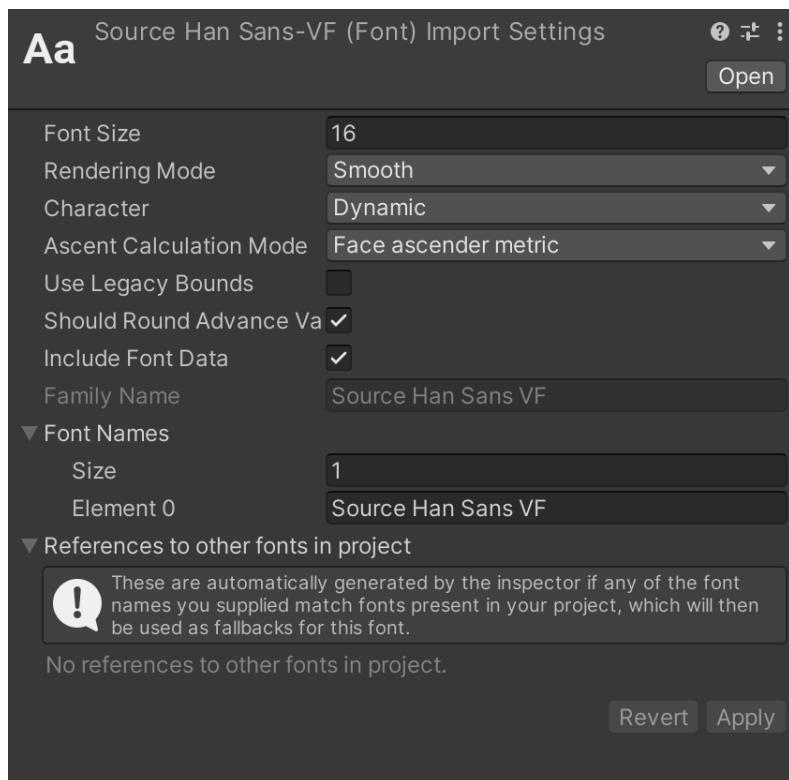
    protected override void OnEnable()
    {
        base.OnEnable();
        paddingWidthProp = serializedObject.FindProperty("paddingWidth");
    }

    public override void OnInspectorGUI()
    {
        base.OnInspectorGUI();
        serializedObject.Update();
        EditorGUILayout.PropertyField(paddingWidthProp);
        serializedObject.ApplyModifiedProperties();
    }
}
#endif

```

## Font assets

### Font class



```
[NativeClass ("TextRendering::Font")]
[NativeHeader ("Modules/TextRendering/Public/Font.h")]
[NativeHeader ("Modules/TextRendering/Public/FontImpl.h")]
[StaticAccessor ("TextRenderingPrivate", StaticAccessorType.DoubleColon)]
public sealed class Font : UnityEngine.Object
```

The [Font](#) class is a class that represents a font asset.

By setting an object of the [Font](#) class to the [font](#) property of the [Text](#) component, we can specify the font to be rendered. First, let's take a look at the properties and methods of the [Font](#) class.

## Properties of the Font class

### dynamic

```
public bool dynamic { get; }
```

Gets whether the font is a dynamic font or not.

It is read-only and cannot be changed. Whether a font is dynamic or not is determined at import time. Details are given in the section on *Import Settings* for font assets.

### fontSize

```
public int fontSize { get; }
```

Get the default size of a font.

It is read-only and cannot be changed.

### material

```
public Material material { get; set; }
```

Gets/Sets the material used to draw the font.

Font textures can be accessed via the `texture` property of a material. We will discuss font textures in detail in *Font textures*. Note that changing the `color` or other properties of this material will not affect the `Text` rendering result.

### characterInfo

```
public CharacterInfo[] characterInfo { get; set; }
```

Obtain an array of information about the characters in the font texture of this font.

The definition of the `CharacterInfo` structure is as follows.

```
// Stores information about how to render characters from font textures
public struct CharacterInfo
{
    // Unicode value of the character
    public int index;

    // UV coordinates of the character in the texture
    // Deprecated, use uvBottomLeft/uvBottomRight/uvTopRight/uvTopLeft instead
    public Rect uv;

    // Screen coordinates of the characters in the generated text mesh
    // Deprecated, use minX, maxX, minY, maxY instead.
    // minX == vert.xMin, maxX == vert.xMax,
    // minY == vert.yMax, maxY == vert.yMin (It is not a mistake that min and max of Y are reversed)
    public Rect vert;

    // The distance between this character and the next character.
    // Deprecated, use advance instead.
    public float width;

    // The size of this character. Default size is 0.
    public int size;

    // The style of this character.
    public FontStyle style;

    // Whether or not this character is rotated in the font texture.
    // This property is deprecated because it cannot correctly determine if the screen orientation changes.
    // An alternative method is described below.
    public bool flipped;

    // Rounded horizontal distance to the nearest integer.
    // This is the distance between the origin of this character and the origin of the next character.
    public int advance;
```

```

// Width of the character image.
public int glyphWidth;

// height of the glyph image
public int glyphHeight;

// distance of the horizontal gap from the origin of this glyph to the start of the glyph image (bearing value)
public int bearing;

// minimum Y coordinate of the glyph image
public int minY;

// maximum Y-coordinate of the letter image
public int maxY;

// minimum X coordinate of the character image
public int minX;

// maximum X coordinate of the character image
public int maxX;

// uvBottomLeft of the uv coordinate in the font texture
public Vector2 uvBottomLeft;

// lower right of the uv coordinate in the font texture
public Vector2 uvBottomRight;

// top right of uv coordinate in font texture
public Vector2 uvTopRight;

// top left of uv coordinate in font texture
public Vector2 uvTopLeft;
}

```

Since `CharacterInfo.flipped` is *Deprecated*, we need to use a different method to determine if a character is rotated in a font-atlas texture. The code to do so is shown below.

```

// Determine if the character shape is rotated in the font texture (instead of flipped, which is deprecated)

```

```
public bool AlternativeFlipped(CharacterInfo characterInfo, Font font)
{
    // If the width and height match, it won't rotate in the first place.
    if (characterInfo.glyphHeight == characterInfo.glyphWidth)
    {
        return false;
    }

    // If the width in the texture matches the height of the character, it is rotated
    float widthInTexture = Mathf.Abs(characterInfo.uvTopRight.x - characterInfo.uvBottomLeft.x
) * font.material.mainTexture.width;
    if (widthInTexture == characterInfo.glyphHeight)
    {
        return true;
    }

    // otherwise, no rotation
    return false;
}
```

ascent

```
public int ascent { get; }
```

Obtain the font's ascent value, which is the distance between the baseline and the top line of the font.

lineHeight

```
public int lineHeight { get; }
```

Get the line height of a font.

## public methods of Font

### GetCharacterInfo

```
public bool GetCharacterInfo(char ch, out CharacterInfo info, int size = 0, FontStyle style = FontStyle.Normal);
```

Obtain [CharacterInfo](#) for a specific character.

If [size](#) is [0](#), the default size of the font will be used; note that if [size](#) is specified, it must be multiplied by the [scaleFactor](#) of the [Canvas](#), not by the [fontSize](#) of the [Text](#) component itself.

If the character specified by the argument exists in the font texture, then [true](#) will be returned as the return value and the appropriate [CharacterInfo](#) will be obtained, but if the character does not exist in the font texture, then only [false](#) will be returned.

### HasCharacter

```
public bool HasCharacter(char c);
```

Returns whether or not a particular character is contained in the font.

### RequestCharactersInTexture

```
public void RequestCharactersInTexture(string characters, int size = 0, FontStyle style = FontStyle.Normal);
```

Add a string to the font texture if the font is a dynamic font.

Note, however, that even if you add a string here, it may disappear when you resize the font texture. We will discuss how to deal with this later.

## Static methods of Font

### GetOSInstalledFontNames

```
public static string[] GetOSInstalledFontNames();
```

Get an array of font names installed in the execution environment.

Either the resulting array or the array itself can be passed to [CreateDynamicFontFromOSFont\(\)](#) to obtain a [Font](#) object.

### CreateDynamicFontFromOSFont

```
public static Font CreateDynamicFontFromOSFont(string fontname, int size);
public static Font CreateDynamicFontFromOSFont(string[] fontnames, int size);
```

Get the fonts installed in the execution environment.

Basically, you pass the font names (array) obtained by [GetOSInstalledFontNames\(\)](#).

### GetPathsToOSFonts

```
public static string[] GetPathsToOSFonts();
```

Get the path to the fonts installed in the OS.

By passing this path to the constructor of the [Font](#) class, you can create a [Font](#) object. For example, to create a [Font](#) object for the installed MS Gothic, the following code would be used.

```
// Create a Font for the installed MS Gothic
public Font CreateMSGothicFont()
{
    // Get the paths to all installed fonts
    string[] fontPaths = Font.GetPathsToOSFonts();
```

```
foreach (var path in fontPaths)
{
    // Get the file name without extension
    string fontFileName = System.IO.GetFileNameWithoutExtension(path).ToLower();

    // If the name matches, create and return a Font
    if (fontFileName == "arial")
    {
        return new Font(fontFileName);
    }
}

return null;
}
```

## GetMaxVertsForString

```
public static int GetMaxVertsForString(string str);
```

Returns the maximum number of vertices that the [TextGenerator](#) will return for the given string.

In fact, it returns the number obtained by multiplying (string length + 1) by 4 (= the four corner vertices).

```
public static int GetMaxVertsForString(string str)
{
    return str.Length * 4 + 4;
}
```

## Events of Font

### textureRebuilt

```
public static event Action<Font> textureRebuilt;
```

Called when the font-atlas texture is resized during rebuild.

It can be used in combination with [Font.RequestCharactersInTexture\(\)](#) when you want to include arbitrary strings in a font-atlas texture in advance. This callback is called from [Canvas.SendWillRenderCanvases](#) during Canvas rebuild. Details of rebuilding font-atlas textures are described in *Font-atlas Textures*.

## Font asset file format

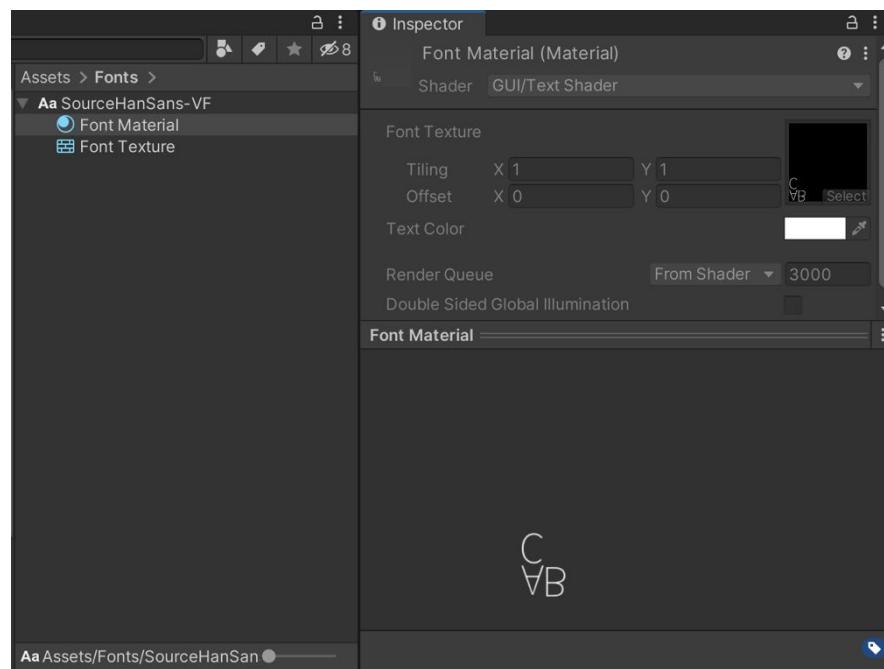
The font asset formats supported by Unity by default are **TrueType (.ttf)** and **OpenType (.otf)**. By default, *Arial* is used as the dynamic font. To load *Arial* from a script, you can load it via Resources.

```
// Load the default font  
Font font = Resources.GetBuiltinResource<Font>("Arial.ttf");
```

If the *Arial* font cannot be found on the computer (e.g. if the font is not installed), the *Liberation Sans* font bundled with Unity will be used as a fallback.

## Dynamic font

Dynamic fonts are often used when there are a large number of character types (such as Japanese or Chinese) and it is not possible to determine the font type to be used before execution. The generated font atlas texture will appear in the **Inspector** preview when you select the **Font Material** font asset in the **Project** view (select **Font Material**, not **Font Texture**; **Font Texture** contains a Font Texture shows the pre-generated texture for non-dynamic fonts).



To access the font-atlas texture from a script, simply access the `mainTexture` from the `Font` object's material.

```
Font font = Resources.GetBuiltinResource<Font>("Arial.ttf");

// Get the font atlas texture
Texture texture = font.material.mainTexture;
```

If **Dynamic** is set for **Characters**, font-atlas textures will not be generated in advance, but will be generated at runtime using the font rendering engine of FreeType (library). On the other hand, if you specify something other than **Dynamic**, the font-atlas texture will be generated in advance using only the characters specified in **Custom Charas**.

## Font-atlas texture

Each loaded font asset holds its own font atlas texture. Different font assets for the same font family will be assigned different font atlas. For example, if one [Text](#) component uses *Arial* (`arial.ttf`) with [Bold FontStyle](#), and another [Text](#) component uses *Arial Bold* (`arialbd.ttf`), the output will look the same, but two font atlases would be generated.

If there are two [Text](#) components, both displaying the letter "A" in the same font, the situation would be as follows.

- If the font size of the two [Text](#) components is the same, the font Atlas texture will contain only one font shape.
- If the font sizes of the two [Text](#) components are different (one is 16 points and the other is 24 points), then the font-atlas texture will contain a total of two shapes, 16 points and 24 points, for the letter "A".
- If one [Text](#) component is bold and the other is not, the font-atlas texture will contain a total of two forms: a bold "A" and a normal "A".

When a [Text](#) object is using a dynamic font and tries to draw a shape that has not yet been rasterized into a font-atlas texture, the font-atlas texture will be rebuilt. The first step in the rebuilding process is to rebuild the font-atlas texture with the current size. The font shape used in this rebuild is the one of the string set in the [text](#) of the active [Text](#) component. If all currently used forms do not fit into the new font atlas texture, the smaller length of the current font atlas texture will be doubled. For example, a [512x512](#) font atlas texture will be enlarged to a [512x1024](#) atlas, and a [512x1024](#) font atlas texture will be enlarged to [1024x1024](#).

In this way, the font-atlas texture of a dynamic font becomes larger and larger. In the case of Japanese, it is not uncommon for the font-atlas texture to grow to its maximum size of [4096x4096](#) due to the large number of letterforms. Considering the cost of rebuilding font atlas textures, minimizing rebuilds is a must. Since rebuilding font-atlas textures during the game will affect the frame rate, it is a good idea to call [Font.RequestCharactersInTexture\(\)](#) at the start of the game to generate a font-atlas texture containing the required characters in advance.

However, when resizing the next time the font-atlas texture is rebuilt, any shapes that are not included in the currently active [Text](#) component will not be included in the new font-atlas texture. Therefore, the shapes set by [Font.RequestCharactersInTexture\(\)](#) may disappear. To do so, you need to subscribe to the [Font.textureRebuilt](#) event and call [Font.RequestCharactersInTexture\(\)](#) again. Below is the code for a script to add arbitrary characters to a font-atlas texture.

```

using UnityEngine;

public class FontTextureInjection : MonoBehaviour
{
    // the font for which we want to manage font atlas textures
    public Font font;

    // desired font size
    public int fontSize;

    // string we want to pre-populate the font texture with
    public string targetString = "";

    // scaleFactor is needed, so set Canvas
    public Canvas canvas;

    private void Start()
    {
        Debug.Assert(font != null, "Font is not set");
        Debug.Assert(fontSize > 0, "Font size is not set");
        Debug.Assert(canvas != null, "Canvas is not set");

        Texture texture = font.material.mainTexture;
        Debug.LogFormat("Size of font-atlas texture {0} {1}", texture.width, texture.height);

        // Add text to the font atlas texture
        // size should be set to Text's Font Size multiplied by Canvas's scaleFactor
        font.RequestCharactersInTexture(targetString, size: (int)(fontSize * canvas.scaleFactor), style: FontStyle.Normal);

        // Register a callback for rebuilding (which may change the size)
        Font.textureRebuilt += OnTextureRebuilt;
    }

    private void OnDestroy()
    {
        // don't forget to release the callback
        Font.textureRebuilt -= OnTextureRebuilt;
    }

    public void OnTextureRebuilt(Font font)
}

```

```

{
    // do not process anything but the desired font
    if (this.font != font)
    {
        return;
    }

    Texture texture = font.material.mainTexture;
    Debug.LogFormat("Size of the modified font-atlas texture {0} {1}", texture.width, texture.height);

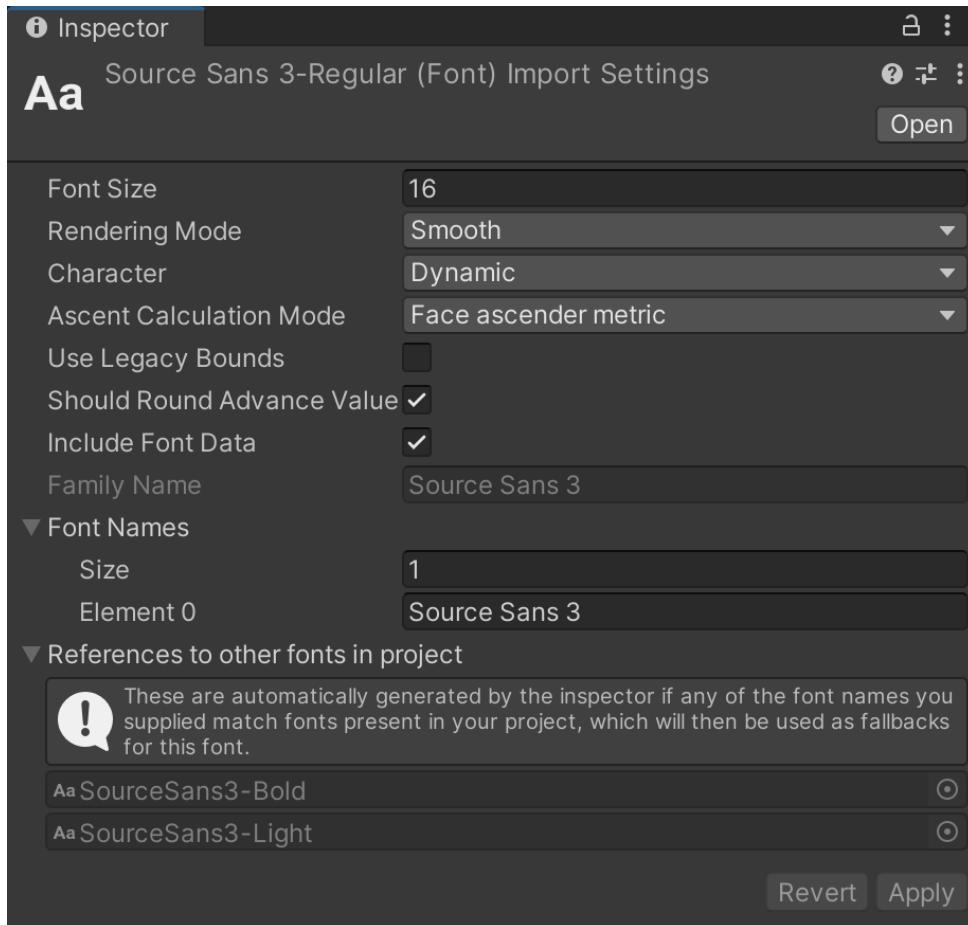
    // Add characters to the font atlas texture
    font.RequestCharactersInTexture(targetString, size: (int)(16 * canvas.scaleFactor), style: FontStyle.Normal);
}

```

Note that an error will be output to the **Console** when the maximum size of the font atlas ([4096 x 4096](#)) is exceeded.

## Font fallback

Fonts listed in the **Font Name** of the font asset import settings will be used as fallback (substitute) if the font shape is not found. The order of fallback is from the top of the **Font Names** list.

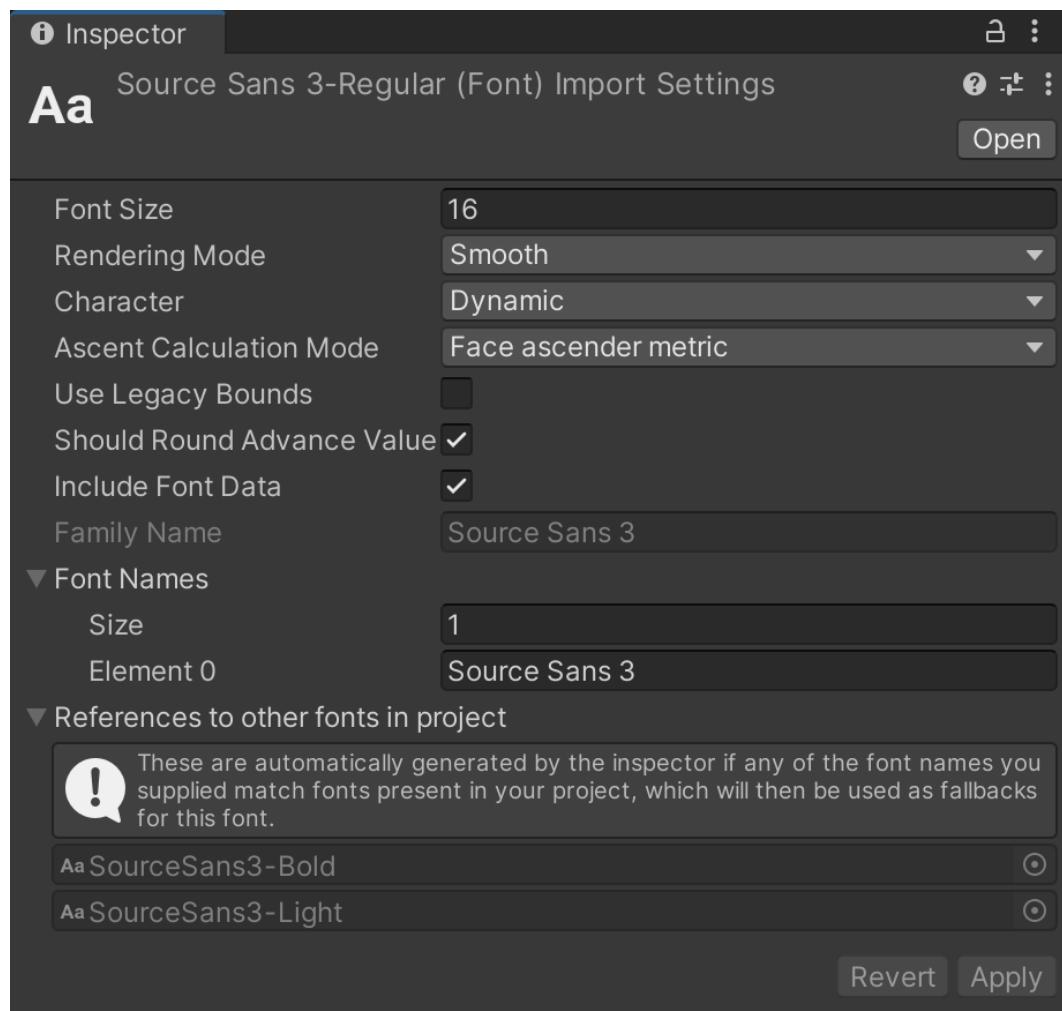


If you are drawing text with a dynamic font and the font is not found (**Include Font Data** is not set or the font is not installed), or the requested font shape is not included in the font data (you are trying to draw Japanese text using a Latin font, or using bold/italic text), the fonts in the list in the **Font Names** field will be checked from top to bottom to see if there is a font that matches the project's font name and if a font with the required font shape is installed. If none of the alternate fonts set as fallback are present, and the requested font shape is not installed, Unity will look for a fallback font in its internally hard-coded font list. This fallback font list contains a variety of international fonts that are normally installed in the current execution environment.

On some platforms (WebGL and some consoles), it is not possible to specify the OS default font that Unity will use for drawing text. For those platforms, the **Include Font Data** setting is ignored and you need to make sure that the font data is always included in the project and that all fonts you want to use as fallback are included in the project. Therefore, if you need to use a font for multilingualization purposes or to draw in bold/italic, you need to prepare a font with only the necessary characters in your project and set that font to **Font Names**. If the font is set correctly, the fallback font will appear as **References to other fonts in the project** in the **Inspector** of the font import settings.

## TrueTypeFontImporter class.

When you select a font asset that has been imported into Unity (i.e. placed under the *Assets* folder) in the **Project** view, the **Import Setting** in the **Inspector** will show the various import settings. This importer is defined as [TrueTypeFontImporter](#), which inherits from [UnityEditor.AssetImporter](#). Although it is called [TrueTypeFontImporter](#), it supports not only *TrueType(.ttf)* but also *OpenType(.otf)*.



## Properties of TrueTypeFontImporter

```
[NativeHeader ("Modules/TextRenderingEditor/TrueTypeFontImporter.h")]
public sealed class TrueTypeFontImporter : AssetImporter
```

### fontSize

```
public int fontSize { get; set; }
```

Gets/Sets the size of individual characters in the font texture for non-dynamic fonts.

The minimum value is `1` and the maximum value is `500`, which is different from the font size of the [Text](#) component. It does not affect the case of dynamic fonts.

### fontRenderingMode

```
public FontRenderingMode fontRenderingMode { get; set; }
```

Gets/Sets the font rendering mode.

This property sets the smoothness of the font, which is displayed as **Rendering Mode** in **Inspector**.

```
public enum FontRenderingMode
{
    /// <summary>
    /// <para> Renders the font using anti-aliasing. For dynamic fonts, this is the fastest mode for
    /// rendering font textures. </para>
    /// </summary>
    Smooth,

    /// <summary>
    /// <para> Renders the font using anti-aliasing and font hinting. Font hinting makes the lines o
    f text follow the boundaries of the pixels, so that they are clearly visible at low resolutions. </par
    a>
    /// </summary>.
```

```

        HintedSmooth,  

    /// <summary>  

    /// <para> Renders the font using only font hinting, not anti-aliasing. This is most suitable for  

    small font sizes, as it produces the clearest borders, but often results in squashed characters in Jap  

    anese. </para>  

    /// </summary>.  

        HintedRaster,  

    /// <summary>  

    /// <para> Use the default OS settings. Valid for dynamic fonts only. </para>  

    /// </summary>  

        OSDefault  

}

```

## fontTextureCase

---

```
public FontTextureCase fontTextureCase { get; set; }
```

Gets/Sets the character set of the font to import into the font texture.

**Inspector** displays it as a **Character**.

The definition of **FontTextureCase** is as follows.

```

public enum FontTextureCase
{
    /// <summary>.  

    /// <para> Dynamic font. Draws the font shape at runtime. </para>  

    /// </summary>  

    Dynamic = -2,  

    /// <summary>  

    /// <para> Unicode character set for Latin characters. </para>  

    /// </summary>  

    Unicode,  

    /// <summary>
}

```

```

/// <para> ASCII characters. </para>
/// <summary>
[InspectorName ("ASCII default set")]
ASCII,

/// <summary>
/// <para> Uppercase ASCII characters. </para>
/// <summary>
ASCIIUpperCase,

/// <summary>
/// <para> Lowercase ASCII character. </para>
/// <summary>
ASCIILowerCase,

/// <summary>
/// <para> A custom character set. When you select this, the Inspector will allow you to specify Custom Chars, and you can specify the character set there. </para>
/// </summary>
CustomSet
}

```

## ascentCalculationMode

---

```
public AscentCalculationMode ascentCalculationMode { get; set; }
```

Gets/Specifies the method used to calculate the font's ascent.

The ascent of a font is the distance between the baseline and the topline of the font. Different fonts calculate the ascent in different ways. Some fonts use the height of the bounding box, others use the height of the caps, and some take into account diacritics such as accent marks. Because these differences affect the vertical alignment of text, Unity provides multiple methods for determining the ascent.

```
public enum AscentCalculationMode
{
    /// <summary>
    /// <para> The previously used method of using bounding boxes. This method calculates the a
}
```

scent using the highest value at the top of the bounding box of the letterforms in the font's character set. This may result in a smaller ascent if the font is not a dynamic font (since it does not calculate all the letterforms). This mode is left for compatibility and should not be used in general. **</para>**

**/// </summary>.**

[InspectorName ("Legacy version 2 mode (glyph bounding boxes)")]  
Legacy2x,

**/// <summary>**

**/// <para>** Use the ascender values defined in the font. **</para>**

**/// </summary>**

[InspectorName ("Face ascender metric")]  
FaceAscender,

**/// <summary>**

**/// <para>** Use the top of the bounding box as defined in the B font. **</para>**

**/// </summary>**

[InspectorName ("Face bounding box metric")]  
FaceBoundingBox

}

## shouldRoundAdvanceValue

```
public bool shouldRoundAdvanceValue { get; set; }
```

Gets/Sets whether the internal advance width of the font should be rounded to the nearest integer.

When placing characters, the spacing between characters may appear uneven due to cumulative errors caused by the rounded advance width. However, the horizontal distance between characters (**advance** in **characterInfo** of the **Font** class) is always rounded to the nearest integer regardless of this setting.

## includeFontData

```
public bool includeFontData { get; set; }
```

Gets/Sets whether the font file is included in the build or not.

Only dynamic fonts will be displayed in the **Inspector**. If this property is set to `true`, the font file (.ttf or .otf) will be included in the build and the font will be used at runtime. When this property is set to `false`, it is assumed that the same fonts are installed in the execution environment. Note that embedding font files may not be allowed, especially for Japanese fonts.

### fontNames

---

```
public string[] fontNames { get; set; }
```

Gets/Sets an array of font names that will be used to find fonts when `includeFontData` is `false`.

When trying to render a font that is not available in this font, it will look for another font in `fontReferences` that matches each font name in this list that has that font form. If it still can't find it, it will look for it among the fonts installed in the OS.

### fontReferences

---

```
public Font[] fontReferences { get; set; }
```

Gets/Sets a list of fallback (i.e. alternate) fonts to use when a font or character is not available.

Unity will automatically find and set the fonts in the project.

### customCharacters

---

```
public string customCharacters { get; set; }
```

Gets/Sets the string to be font-textured if `CustomSet` is specified for `fontTextureCase`.

### characterPadding

---

```
public int characterPadding { get; set; }
```

Gets/Sets the number of pixels to add to the character border for padding.

This may be used when using a shader that draws the outline of a character. If this value is greater than 0, the perimeter of the font texture and the rectangle of each character in [Text](#) will be larger.

This property is not visible in the [Inspector](#), so if you want to change it, you need to write a custom importer.

```
using UnityEditor;

public class CustomAssetPostprocessor : AssetPostprocessor
{
    // There is no method called OnPreprocessFont, so we'll use OnPreprocessAsset
    private void OnPreprocessAsset()
    {
        TrueTypeFontImporter fontImporter = assetImporter as TrueTypeFontImporter;
        if (fontImporter != null)
        {
            // Get the font file name.
            string fontFileName = System.IO.Path.GetFileNameWithoutExtension(fontImporter.assetPath);

            // If it's the font we want (in this case we'll use Genno Kaku Gothic Normal)
            if (fontFileName == "SourceHanSans-Normal")
            {
                // set Padding to 2
                fontImporter.characterPadding = 2;
            }
        }
    }
}
```

One quick way is to open the font meta file in a text editor and edit the [characterPadding](#) value directly, but this is not recommended.

## characterSpacing

```
public int characterSpacing { get; set; }
```

Gets/Sets the space of the area around the font texture's letterforms.

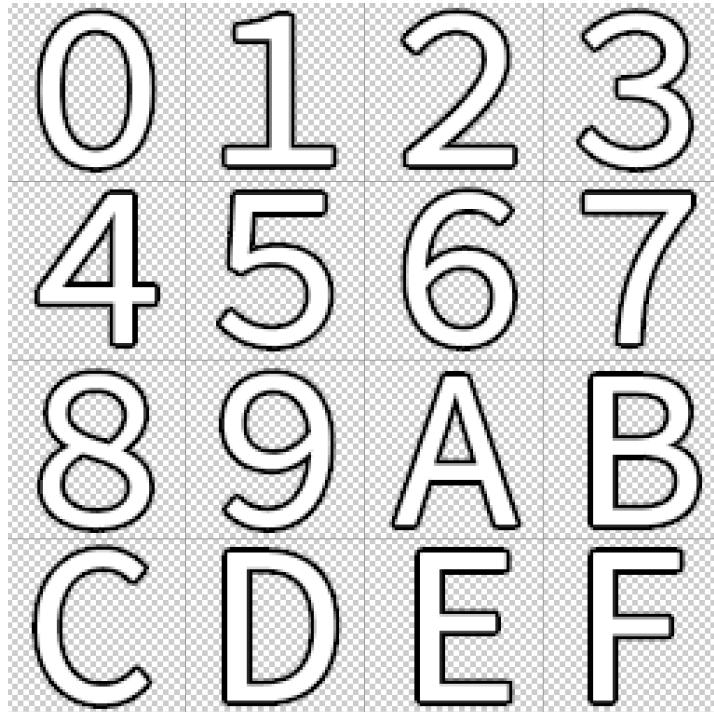
As with [characterPadding](#), this property is not visible in the **Inspector**. Unlike [characterPadding](#), it does not affect the rectangle of each character in the [Text](#). Unlike [characterPadding](#), it does not affect the rectangle of each character in the text. If you want to use outline shaders, but [characterPadding](#) is set, you can leave this value at [0](#).

## Use Legacy Bounds

This is an unspecified property that is displayed in **Inspector**. There was no documentation or source code for this property, so I could not confirm how it behaves. It is probably best to leave it off.

## Custom fonts

To create a custom font, we will first prepare a font texture. In this case, we have prepared a 256x256 size texture containing a total of 16 characters: the numerals [0](#) (48 in ASCII code) through [9](#) (57 in ASCII code) and the lowercase letters [a](#) (97 in ASCII code) through [f](#) (102 in ASCII code), as shown below. In other words, the size of each letter shape is 64x64.



For font texture import settings, leave **Texture Type** set to *Default*. If you don't need mipmaps, leave **Generate Mip Maps** off.

The next step is to create a material with this font texture, which we will set to a shader that can be textured from the **Inspector**. The content of the shader can be the same as [UI/Default](#). If you want to create another shader that has the same properties as the other shaders, it is easier to just specify the shader in [FallBack](#).

```
Shader "UI/CustomFont"
{
    Properties
    {
        // Remove [PerRendererData] as we want to set it from Inspector
        _MainTex("Font Texture", 2D) = "white" {}
        _Color("Tint", Color) = (1,1,1,1)

        _StencilComp("Stencil Comparison", Float) = 8
        _Stencil("Stencil ID", Float) = 0
        _StencilOp("Stencil Operation", Float) = 0
        _StencilWriteMask("Stencil Write Mask", Float) = 255
        _StencilReadMask("Stencil Read Mask", Float) = 255

        _ColorMask("Color Mask", Float) = 15

        [Toggle(UNITY_UI_ALPHACLIP)] _UseUIAlphaClip("Use Alpha Clip", Float) = 0
    }

    FallBack "UI/Default"
}
```

Now let's move on to creating a custom font asset. To create a custom font asset, select *Assets->Create->Custom Font* from the menu.

Let's fill in the various settings from the **Inspector** of the custom font asset.

- **Line Spacing** is the distance between the lines. In this case, we will set it to [64](#) pixels, the same as the height of the letterforms. Multiply this value by the **Line Spacing** of the **Text** component to get the actual number of pixels between lines.
- For the **Default Material**, set the material with the font texture that you just created.
- The **Ascii Start Offset** specifies the number of indices to start with for **Character Rects**. For example, if the **Ascii Start Offset** is [0](#), the character '0' will have an index of [48](#). If the **Ascii Start Offset** is [48](#), the character '0' will have an index of 0. If the Ascii Start Offset is 48, the index of the letter '0' will be [0](#). In this case, we will set the **Ascii Start Offset** to [0](#).

- **Tracking** sets the distance between adjacent characters in the same line. If you want to have no gaps, set it to 1. If you want to increase the gaps, enter 1.2, etc. to increase the spacing.
- If **Convert Case** is 1, lowercase letters are capitalized (e.g., 'a' in ASCII code 97 becomes 'A' in ASCII code 65).
- **Character Spacing**, **Character Padding**, and **Font Rendering Mode** are not mentioned in the documentation, and changing their values does not seem to affect rendering.
- In the **Character Rects** section, configure the settings for each character in the font.
- **Size** sets the number of characters in the font. In this case, we are using 16 characters, so we will set it to 16. This will create elements from **Element 0** to **Element 15**, and we will enter the information for each character in each of them. In **Element 0**, we will enter the information for the top-left character, Shape 0.
- Enter a Unicode (UTF-16) value (or an ASCII code value if it is in the ASCII code range) for **Index**. The ASCII code for character type 0 is 48, so enter 48.
- **Uv** is the UV coordinate (between 0 and 1) in the font texture; **X** and **Y** are the distances from the bottom left corner; **X** is 0, Y is 3/4 from the bottom, so 0.75; **W** and **H** are the width and height, both 0.25.
- **Vert** is the coordinate and size (both in pixels) of the vertex to be displayed, and **X** and **Y** can be 0 if you don't want to shift the display position. 64 is the width as is for **W**, and -64 is the height multiplied by -1 for **H**.
- **Advance** is the distance (in pixels) between the next character and the next character.
- If **Flipped** is true, the letterforms are said to be flipped vertically and rotated 90 degrees counterclockwise in the font texture.

In this way, we repeatedly input numerical values for each character form. However, doing all this by hand can be quite an arduous task. Of course, under normal circumstances, when creating a custom font, you would probably look for a convenient asset to use. However, if you are a reader of this book, you will probably want to build your own tools, so here is a sample code for setting parameters based on the textures in this article.

```
public static void UpdateCustomFontSetting()
{
    // Path name of the custom font to be set
    string path = "Assets/Runtime/Fonts/MyCustomFont.fontsettings";

    // Load the font assets
    Font font = AssetDatabase.LoadMainAssetAtPath(path) as Font;
    if (font == null)
    {
        Debug.LogErrorFormat("Font file not found {0}");
        return;
    }
}
```

```

// Serialize the contents of the Custom Font and set the values
SerializedObject serializedObject = new SerializedObject(font);
serializedObject.Update();

// You can check the property name by opening the file with a text editor
serializedObject.FindProperty("m_LineSpacing").floatValue = 64.0f;
serializedObject.FindProperty("m_AsciiStartOffset").intValue = 0;
serializedObject.FindProperty("m_Tracking").intValue = 1;
serializedObject.FindProperty("m_ConvertCase").intValue = 0;

// Set the contents of the m_CharacterRects array
SerializedProperty characterRectArray = serializedObject.FindProperty("m_CharacterRects");
int arraySize = characterRectArray.arraySize;

for (int i = 0; i < arraySize; i++)
{
    // each element
    SerializedProperty characterRect = characterRectArray.GetArrayElementAtIndex(i);

    if (i < 10)
    {
        // characters from 0 to 9
        characterRect.FindPropertyRelative("index").intValue = 48 + i;
    }
    else
    {
        // a, b, c, d, e, f character shapes
        characterRect.FindPropertyRelative("index").intValue = 97 + (i - 10);
    }

    SerializedProperty uv = characterRect.FindPropertyRelative("uv");
    uv.FindPropertyRelative("x").floatValue = (i % 4) * 0.25f;
    uv.FindPropertyRelative("y").floatValue = (3 - (i / 4)) * 0.25f;
    uv.FindPropertyRelative("width").floatValue = 0.25f;
    uv.FindPropertyRelative("height").floatValue = 0.25f;

    SerializedProperty vert = characterRect.FindPropertyRelative("vert");
    vert.FindPropertyRelative("x").floatValue = 0;
    vert.FindPropertyRelative("y").floatValue = 0;
    vert.FindPropertyRelative("width").floatValue = 64;
    vert.FindPropertyRelative("height").floatValue = -64;
}

```

```
characterRect.FindPropertyRelative("advance").floatValue = 64;
characterRect.FindPropertyRelative("flipped").intValue = 0;
}

// Save the configuration
serializedObject.ApplyModifiedProperties();
serializedObject.SetIsDifferentCacheDirty();
AssetDatabase.SaveAssets();
}
```

## TextMesh Pro

TextMesh Pro (TMP) is a powerful mechanism for text rendering that can be used in place of the [Text](#) component. TextMesh Pro uses *Signed Distance Field (SDF)* as its main rendering pipeline and can render text beautifully at any point size and resolution. TextMesh Pro uses Signed Distance Field (SDF) as its main rendering pipeline and can render text beautifully at any point size and resolution. It also allows you to adjust kerning, which is not possible with standard [Text](#). Other effects such as outlines, soft shadows, and bevels can be achieved by using custom shaders for TextMesh Pro.

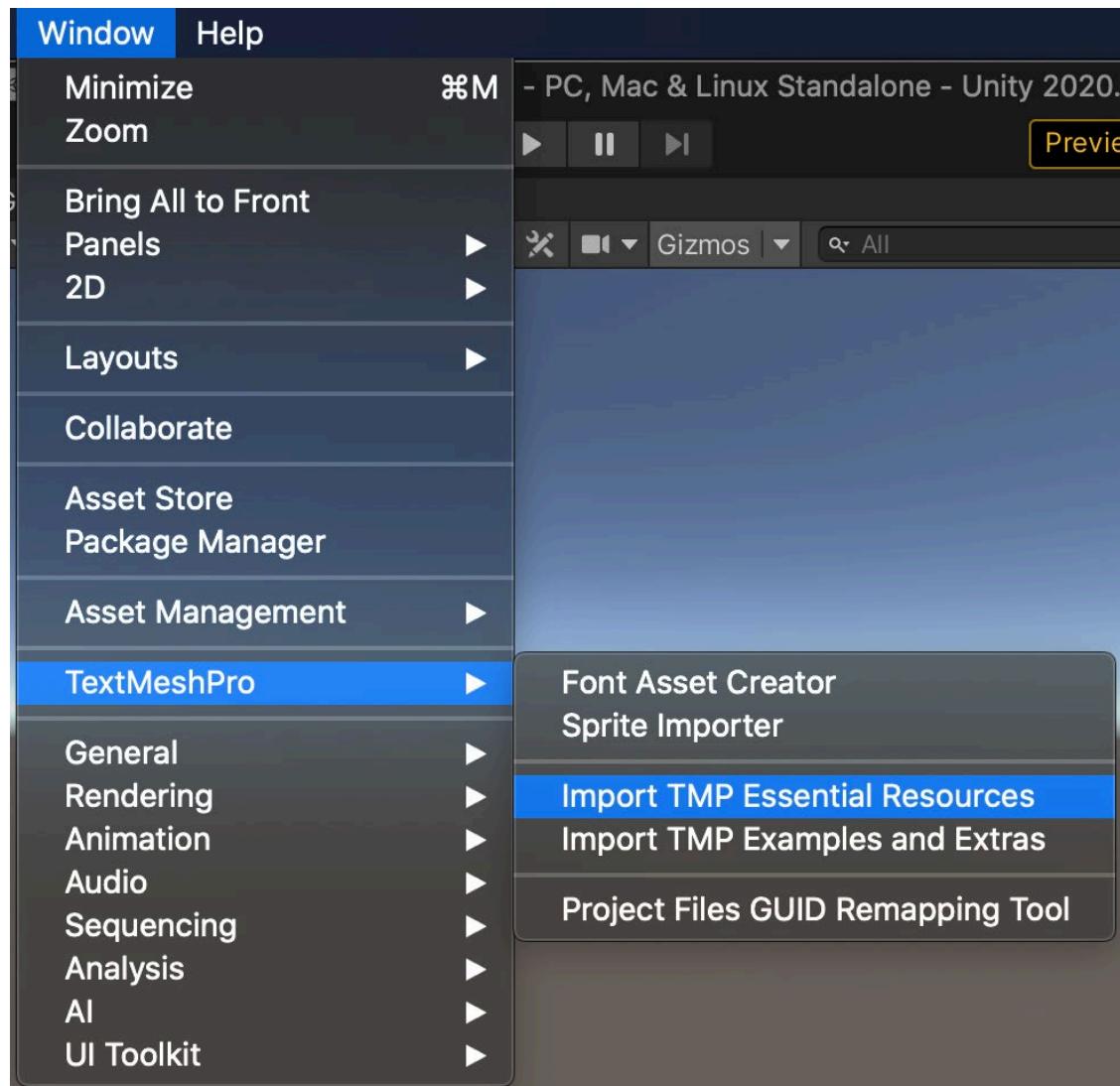
Before Unity 2018.1, TextMesh Pro was included in some projects as an Asset Store package, but since Unity 2018.1, it is available as a Package Manager package. In addition, TextMesh Pro 1.4, available since Unity 2018.3, supports fallback and dynamic fonts, making it easier to use Japanese with a large number of characters. This document is based on Text Mesh Pro 3.0, which is available in Unity 2020.1.

There are two types of TextMesh Pro components: the [TextMeshPro](#) component, which is drawn in 3D space by the [MeshRenderer](#), and [TextMeshProUGUI](#) component, which is drawn in [Canvas](#). For text displayed in world space, it is more efficient to use the [TextMeshPro](#) component instead of the [TextMeshProUGUI](#) component.

As with Unity's built-in [Text](#) component, changing the text displayed by TextMesh Pro will result in a Canvas rebuild. So, as with the [Text](#) component, try to keep runtime changes to the [TextMeshProUGUI](#) component to a minimum.

## Install TextMesh Pro

In Unity 2018.1 or later, you can select *Window -> TextMeshPro -> Import TMP Essential Resources* from the menu to open the **TMP Importer** dialog and install the basic resources of TextMesh Pro as a package. You can install the basic resources of TextMesh Pro as a package.

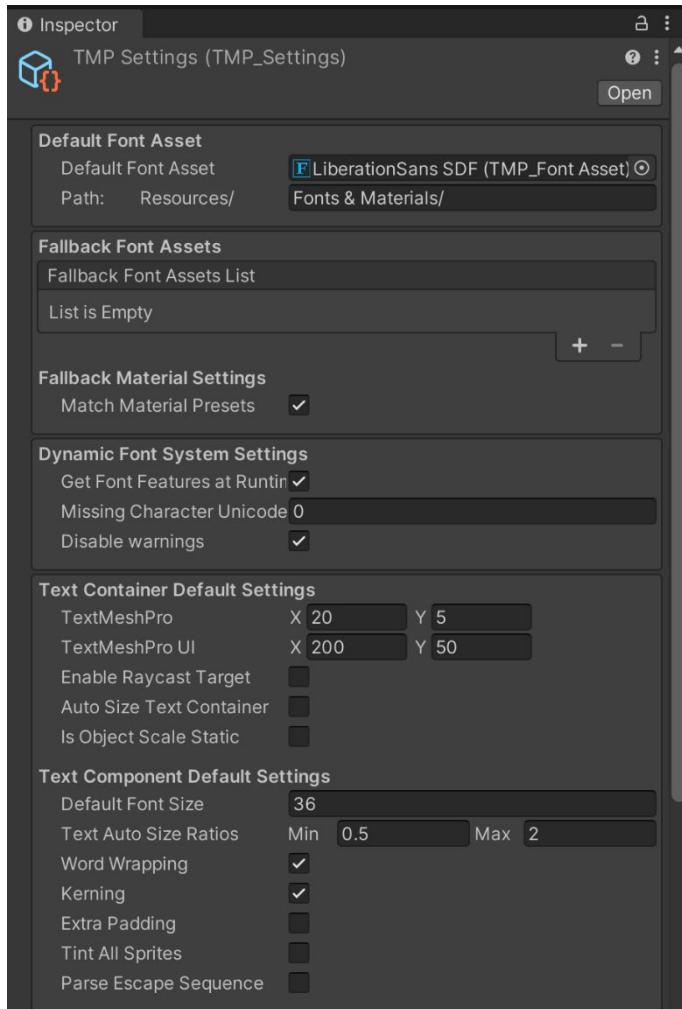


Alternatively, you can right-click in the **Hierarchy** and select *UI -> Text - TextMeshPro*, etc. to display the **TMP Importer** dialog as well.

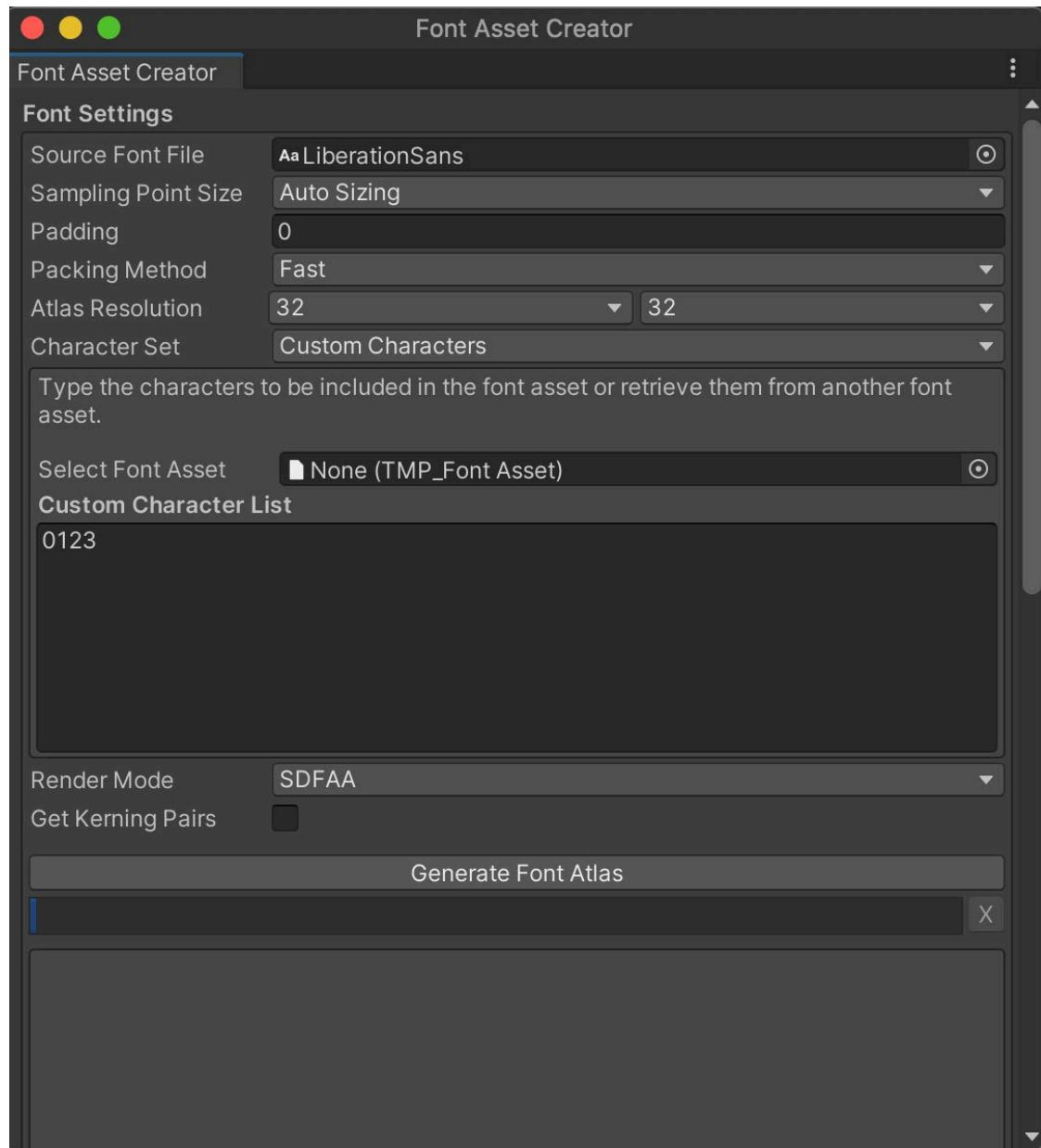
Select **Import TMP Essential Resources** to install basic resources, and then select **Import TMP Examples & Extras** to install sample scenes, fonts, and materials under the Examples & Extras folder of TextMesh Pro. and fonts under the TextMesh Pro Examples & Extras folder. It is a good idea to take a look at the sample scenes for now.

## TMP Settings

Various default settings are described in the *Assets/TextMesh Pro/Resources/TMP Settings* file. For example, it may be useful to uncheck **Enable Raycast Target**.



## Setting up Font Asset Creator



TextMesh Pro has its own font asset format that needs to be generated from TrueType (.ttf) and OpenType (.otf) files using the **Font Asset Creator**. To run the Font Asset Creator, select *Window -> TextMeshPro -> Font Asset Creator* in the Editor.

## Source Font File

---

Set the font file (.ttf or .otf) from which the TextMesh Pro font assets will be generated. The source font file set here does not need to be included in the build.

## Sampling Point Size

---

Specify the size of the font shape in the font texture; if **Auto Sizing** is set automatically, the size of the font shape will be set so that it just barely fills the texture. If the size is set automatically with Auto Sizing, the **Font Size** is set so that it is just barely full. If you specify twice the size, the quality may be too high.

You can check the size of the letterforms when **Auto Sizing** is specified by looking at the **Sampling Point Size** in the **Generation Settings** of the font asset import settings after generation. If this size is too small or too large compared to the display size, consider adjusting the font texture size in **Atlas Resolution**.

## Padding

---

It specifies the gap between characters in a font texture in pixels; the larger the padding, the smoother the font texture and the better the quality of outlines and other effects, according to the Signed Distance Field (SDF) mechanism. As a rule of thumb, about [5](#) pixels is sufficient for a [512x512](#) texture. If you don't want to add any special effects, [2](#) pixels is fine.

## Packing Method

---

If you specify **Optimum**, the font texture will be automatically adjusted to the largest possible font size; if you specify **Fast**, the font texture generation time will be slightly shorter and the font size will be roughly adjusted. If you specify **Fast**, the font texture generation time will be slightly shorter and the font size will be roughly adjusted. You may want to specify **Fast** when adjusting, and **Optimum** when you have finished adjusting the final parameters.

## Atlas Resolution

---

The larger the font texture, the higher the quality of the character rendering. For most fonts, a size of [512x512](#) is sufficient to achieve good quality for ASCII characters only. Japanese fonts will require larger textures, but keep in mind that there are a good number of Android devices that are limited to a maximum texture size of [4096x4096](#). Even if the texture size is [4096x4096](#), the memory consumed will be 16MB (1 byte per pixel, so [4096x4096x1](#)), which is a large percentage

of memory consumed for a smartphone application. Try to reduce the amount of memory used by using fallback fonts as described below.

## Character Set

---

Specifies the Unicode (UTF16) of the characters to be included in the font texture (without leading 0x) in hexadecimal comma separated format. The following items are available as presets.

- ASCII, Extended ASCII (ISO 8859)
- ASCII Lowercase (lowercase only)
- ASCII Uppercase (uppercase only)
- Numbers + Symbols (numbers and ASCII symbols)
- Custom Range (Unicode in decimal, such as 20-40, or comma separated)
- Unicode Range (Hex) (range or comma separated using - in hexadecimal without leading 0x)
- Custom Characters (enter the characters themselves)
- Characters from File (set a UTF8 Text Asset with the characters you want to use drawn on it)

For Japanese, one might consider setting up JIS level 1 kanji (2956 characters) in addition to hiragana and katakana, but that is still a large enough number and includes kanji that are not often used. Even if you narrow it down to the regular kanji, you still end up with 2136 characters. The most effective way to do this is to identify all the characters that are known to be used in the game, set them in the **Character Set**, and generate the other characters at runtime as **Dynamic** or use the **Text** component instead of TextMesh Pro. Do not create font textures with more than 2,000 characters (including many characters that will likely not be used) without thinking about it.

## Render Mode

---

Specifies the rendering method for the distance field; **SDFAA** is the most balanced, so unless otherwise specified, **SDFAA** should be used.

- **SMOOTH\_HINTED**: 8-bit bitmap font with hinting
- **SMOOTH**: 8-bit bitmap font
- **RASTER\_HINTED**: 1-bit bitmap font with hinting
- **RASTER**: 1-bit bitmap font
- **SDF**: 1-bit Signed Distance Field
- **SDF8**: 1-bit Signed Distance Field upscaled by a factor of 8
- **SDF16**: Upscales a 1-bit Signed Distance Field by a factor of 16
- **SDF32**: Upscales a 1-bit Signed Distance Field by a factor of 32
- **SDFAA\_HINTED**: 8-bit Signed Distance Field with hinting
- **SDFAA**: 8-bit Signed Distance Field

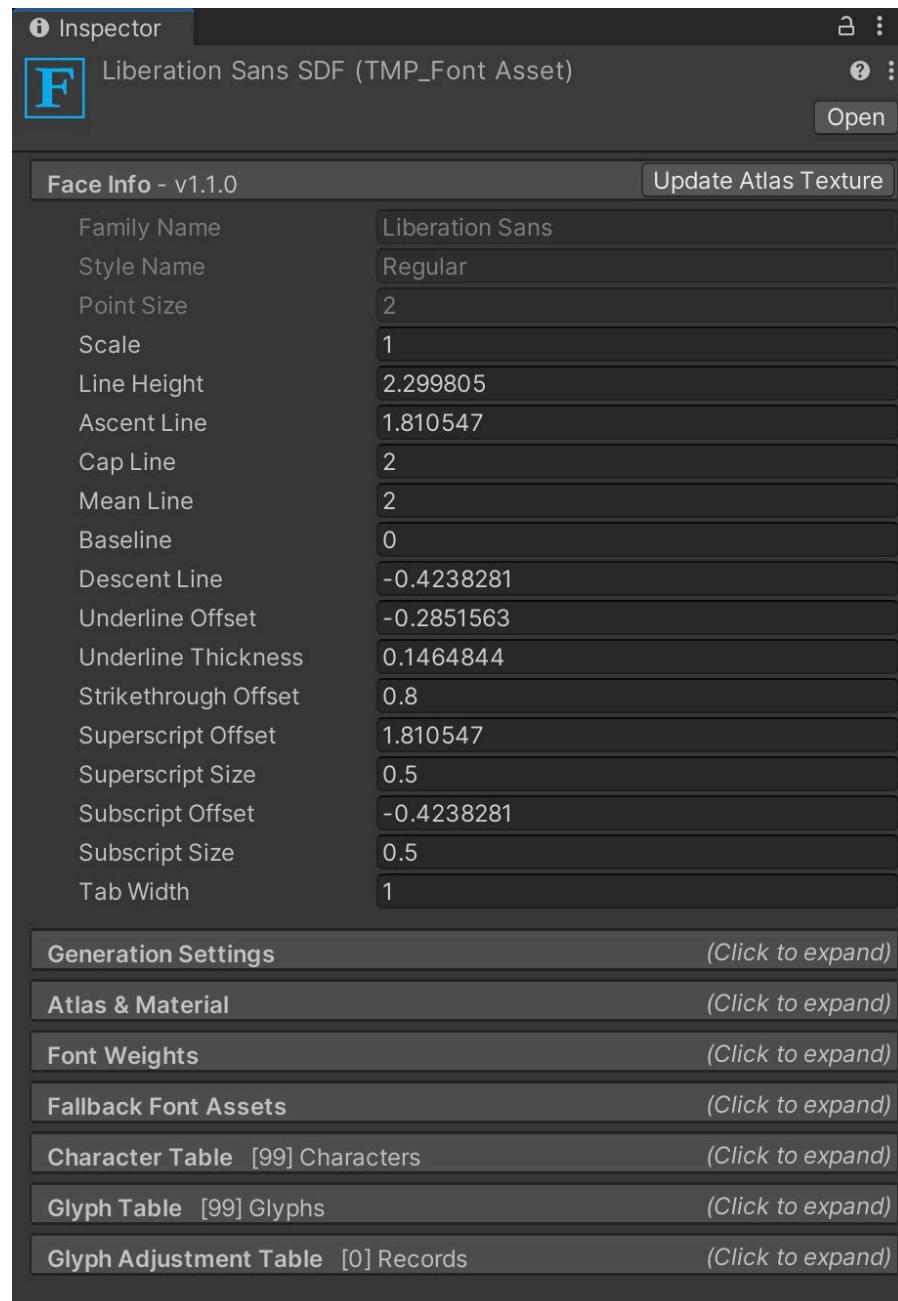
## Get Kerning Pairs

---

If this is checked, the kerning information set in the font can be used. This information can be used to adjust the space between characters for certain combinations of characters, resulting in a better appearance. However, many fonts do not have kerning pair information.

## Generation Settings in TextMesh Pro

Select the TextMesh Pro font asset file (.asset) created with the **Font Asset Creator** in the **Project** view, and the information will appear in the **Inspector**.



Press the **Update Atlas Texture** button in the upper right corner to open the **Font Asset Creator** window based on the settings of the currently selected TextMesh Pro font asset.

Most of the information displayed in the **Inspector** is calculated based on **Font Asset Creator** settings, but there are two important settings to be made here: **Atlas Population Mode** and **Fallback Font Assets**. One is the Atlas Population Mode, and the other is Fallback Font Assets.

### Atlas Population Mode

---

You can specify whether you want Dynamic or Non-Dynamic(Static) fonts; if Dynamic, you can add shapes to the font texture if they are not already there at runtime. If you use Static, you cannot use any font texture other than the one already created.

At first glance, there seems to be no reason to choose Static, but there are some traps in the behavior when Dynamic is selected.

1. Adding letterforms to font textures at runtime has a significant performance impact. Especially in the case of SDFs, adding shapes takes a lot of time.
2. Added shapes will not be deleted even if they are no longer used. If you use the standard [Text](#) component, unused shapes will be removed from the font texture, but not in TextMesh Pro. If you want to remove a font, you must call [TMP\\_FontAsset.ClearFontAssetData\(\)](#) to clear all the shapes in the font texture. However, then we would have to add the shapes to the font texture again, and as mentioned above, we would suffer a performance penalty when adding font texture shapes. To make matters worse, it has been suggested that [TMP\\_FontAsset.ClearFontAssetData\(\)](#) may not be able to be called from outside in the future.

As a result, TextMesh Pro does not allow the use of dynamic fonts as easily as the standard [Text](#) component. In order to solve the above problem, we need to set up a good fallback font.

### Multi Atlas Texture

---

The Multi-Atlas Texture feature allows font assets to create new atlas textures as needed when they are no longer able to add shapes to the main atlas. These added atlas textures will inherit the settings of the main atlas texture as defined in the font asset's **Generation Settings**. All atlas textures, including the main atlas texture, will be included in the [Texture2D](#) array, with the main always being index [0](#).

The Multi-Atlas Texture feature can be enabled only when the **Atlas Population Mode** is **Dynamic**.

Changes to font assets, including the addition of new atlas textures, are persistent in the Editor, but not at runtime. For example, Dynamic font assets with multiatlas textures enabled will have atlas textures enlarged or added as needed to handle the coverage of the letterforms during play, but will revert to their original state the next time you play.

Changes made to font assets in the Editor are permanent, but it is possible to reset font assets with the Reset Context Menu option. This will clear the **Glyph**, **Charater**, and **Glyph Adjustment Tables**. It also clears all atlas textures.

When you make changes to the **Gereration Settings**, such as the atlas texture size, in the Editor, it will automatically regenerate and repack the existing letterforms, and adjust the number of atlas textures as needed based on the new settings. This makes it easy and convenient to optimize the font assets and atlas later. The multi-atlas texturing feature applies to all Dyanmic font assets.

As mentioned earlier, these atlas textures are contained in an array, and in this initial implementation, a draw call is generated for each atlas texture. However, we plan to add support for texture arrays to the shaders in the future (texture arrays are also supported on modern mobile devices). This will make it possible to draw with a single draw call.

## Fallback Font Assets

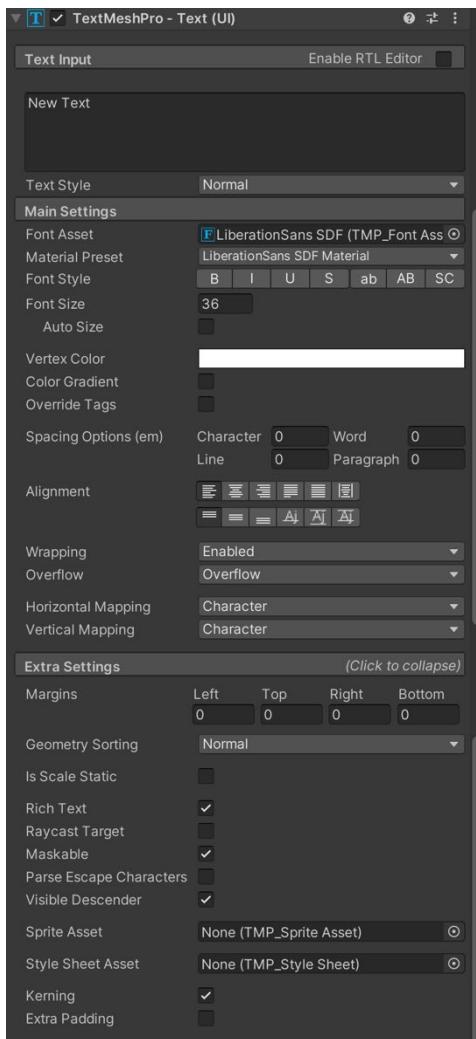
---

A proper understanding of how to use fallback fonts is necessary to optimize performance when using TextMesh Pro.

The search for a font shape in TextMesh Pro is recursive. If a letterform is not found in TextMesh Pro's font assets, the list of fallback assets is examined in turn. If the shape is still not found, it will check the font assets assigned to the same text and the fallback font assigned to that font asset. If you still can't find the font, check the Fallback **Font Assets** under *TextMesh Pro -> Settings in Project Settings*. If you still can't find it, check the **Default Sprite Asset** in *TextMesh Pro -> Settings in Project Settings*. As a last resort, display the characters in **Missing Character Unicode in** TextMesh Pro -> Settings in **Project Settings**.

It is important to note that fallback fonts are loaded recursively when the [TextMeshProUGUI](#) component containing the original font is activated. If there are many fonts, the fallback fonts will also be loaded into memory. Therefore, projects that support multiple languages should be careful not to load extra fallback fonts. Creating an asset bundle for each language (not just fonts) is an efficient way to manage assets.

## TextMeshProUGUI component



```
[DisallowMultipleComponent]
[RequireComponent(typeof(RectTransform))]
[RequireComponent(typeof(CanvasRenderer))]
[AddComponentMenu("UI/TextMeshPro - Text (UI)", 11)]
[ExecuteAlways]
[HelpURL("https://docs.unity3d.com/Packages/com.unity.textmeshpro@3.0")]
public partial class TextMeshProUGUI : TMP_Text, ILayoutElement

public abstract class TMP_Text : MaskableGraphic
```

The [TextMeshProUGUI](#) component inherits from the [MaskableGraphic](#) component via the [TMP\\_Text](#) component. Therefore, the basic performance considerations are the same as for [Graphic](#) components such as [Text](#).

The [TMP\\_Text](#) component is also the parent of the [TextMeshPro](#) component, which is used by the [MeshRenderer](#) to draw text in 3D space.

```
MonoBehaviour ← UIBehaviour ← Graphic ← MaskableGraphic ← TMP_Text ← TextMeshPro  
← TextMeshProUGUI
```

Note that [TextMeshProUGUI](#) adds [TexCoord1](#), [Normal](#), and [Tangent](#) to the [additionalShaderChannels](#) of [Canvas](#) when creating a text mesh. So please be careful if you have made any changes to the [additionalShaderChannels](#) of [Canvas](#).

There are so many properties and public methods of [TMP\\_Text](#) that it is difficult to explain them all in this document. Therefore, in the following, only the properties and public methods defined in the [TextMeshProUGUI.cs](#) file and the properties displayed in the [Inspector](#) will be explained.

## Properties of TextMeshProUGUI

### text

```
public virtual string text { get; set; }
```

Gets/Sets the value of the text to be displayed.

As with the [Text](#) component's `text`, Layout rebuild and Graphic rebuild occur when the string is changed.

### isRightToLeftText

```
public bool isRightToLeftText { get; set; }
```

Gets/Sets whether the text should be displayed from right to left instead of left to right.

The default value is `false`.

If this property is changed, Layout rebuild and Graphic rebuild will occur.

### textStyle

```
public TMP_Style textStyle { get; set; }
```

Gets/Sets the display style of the text.

Styles are defined by tags like rich text. Available styles are stored as style sheets. To check the default style sheet, you can look at the contents of the style sheet (*Assets/TextMesh Pro/Resources/Style Sheets/Default Style Sheet*) set in *TMP Settings*.

The list of available tags can be found on the old official website of TextMesh Pro.  
<http://digitalnativestudios.com/textmeshpro/docs/rich-text/>

## font

---

```
public TMP_FontAsset font { get; set; }
```

Gets/Sets the font assets used to draw the text.

[TMP\\_FontAsset](#) is a [ScriptableObject](#). By default, it is set to *Assets/TextMeshPro/Resources/Fonts & Materials/LiberationSans SDF.asset*.

If this property is changed, Layout rebuild and Graphic rebuild will occur.

```
public virtual Material fontSharedMaterial { get; set; }
```

Gets/Sets the material used to draw the text.

In the Editor, several materials are shown in a drop-down menu. The materials shown in this menu are those set in the font asset set to [font](#), and those retrieved from the [AssetDataBase](#) using the name of the [font](#) (the first substring separated by a space).

If this property is changed, a Graphic rebuild will occur.

## fontStyle

---

```
public FontStyle fontStyle { get; set; }
```

Gets/Sets font styles. [FontStyles](#) is defined as follows

```
public enum FontStyle
{
    Normal = 0x0,
    Bold = 0x1,
    Italic = 0x2,
    Underline = 0x4, // Underline
    LowerCase = 0x8, // lowercase
    UpperCase = 0x10, // uppercase
    SmallCaps = 0x20, // uppercase and lowercase except at the beginning of words
```

```
    Strikethrough = 0x40, // strikethrough  
    Superscript = 0x80, // superscript, not displayed in Inspector. Used in <sup> tags.  
    Subscript = 0x100, // Subscript, not displayed in Inspector. Used in <sub> tags.  
    Highlight = 0x200, // highlight, not shown in Inspector. Used in <mark> tags.  
};
```

`FontStyles` is a bitflag that can be applied multiple times. If this property is changed, a Layout rebuild and a Graphic rebuild will occur.

## fontSize

```
public float fontSize { get; set; }
```

Gets/Sets the font size.

If `enableAutoSizing` is not enabled (default), the default font size is `36`. If this property is changed, Layout rebuild and Graphic rebuild will occur.

## enableAutoSizing

```
public bool enableAutoSizing { get; set; }
```

Gets/Sets whether the font size should be automatically changed to fill the text display area (taking into account ascenders, etc.).

The default value is `false`.

It behaves in the same way as the Best Fit setting for the `Text` component, but since the adjustment is made in units of `0.05` points, the load will be even higher than Best Fit when enabled. Therefore, it should only be used to calculate the best font size in advance, and should not be used in Play mode while it is set to `true`.

If this property is changed, Layout rebuild and Graphic rebuild will occur.

## fontSizeMin / fontSizeMax

---

```
public float fontSizeMin { get; set; }
public float fontSizeMax { get; set; }
```

Gets/Sets the minimum/maximum font size value when `enableAutoSizing` is true.

The default value of `fontSizeMin` is the font size multiplied by `TMP_Settings.defaultTextAutoSizingMinRatio`, and the default value of `fontSizeMax` is `TMP_Settings.defaultTextAutoSizingMaxRatio`.

The value of `TMP_Setting` is defined in the `ScriptableObject` in *Assets/TextMesh Pro/Resources/TMP Settings*, and by default, `defaultTextAutoSizingMinRatio` is 0.5 and `defaultTextAutoSizingMaxRatio` is 2.

If this property is changed, Layout rebuild and Graphic rebuild will occur.

## characterWidthAdjustment

---

```
public float characterWidthAdjustment { get; set; }
```

Gets/Sets the percentage of the text width that will be shrunk if the text is not cured in the display area when `enableAutoSizing` is true.

The default value is 0, and the maximum value is 50.

If this property is changed, Layout rebuild and Graphic rebuild will occur.

## lineSpacingAdjustment

---

```
public float lineSpacingAdjustment { get; set; }
```

Takes a value of 0 or negative. The default value is 0, and the maximum value is 0.

If this property is changed, Layout rebuild and Graphic rebuild will occur.

## color

---

```
public override Color color { get; set; }
```

Gets/Sets the base color (i.e., vertex color) of the text.

If `overrideColorTags` is `true`, the `<color>` tag will be ignored and the text will be rendered in this color. Conversely, if `overrideColorTags` is `false` (which is the default), the `<color>` tag can override this color.

If this property is changed, a Graphic rebuild will occur.

## enableVertexGradient

---

```
public bool enableVertexGradient { get; set; }
```

Gets/Sets whether the vertex color should be gradient enabled or not.

When this property is set to `true`, the **Color Preset**, **Color Mode**, and **Colors** will be displayed in the **Inspector**.

## colorGradientPreset

---

```
public TMP_ColorGradient colorGradientPreset { get; set; }
```

Gets/Sets the vertex color gradient preset.

The preset defines the **ColorMode** (described below) and the vertex colors of the four corners.

*If you select **Import TMP Examples & Extras** when importing TextMesh Pro, or if you have manually installed the *TMP Examples & Extras* package, you will find several presets under *Assets/TextMesh Pro/Examples & Extras/Resources/Color Gradient Presets*.*

If this property is changed, a Graphic rebuild will occur.

## m\_colorMode

---

```
protected ColorMode m_colorMode = ColorMode.FourCornersGradient;
```

Indicates the type of gradient.

It is not a property, but it is a variable that is displayed in the **Inspector**, so we will explain it here. **ColorMode** is defined as follows.

```
public enum ColorMode
{
    Single, // single color
    HorizontalGradient, // horizontal gradient
    VerticalGradient, // vertical gradient
    FourCornersGradient // four corners gradient
}
```

## colorGradient

---

```
public VertexGradient colorGradient { get; set; }
```

Gets/Sets the vertices of the four corners of the gradient.

The definition of **VertexGradient** is as follows.

```
[Serializable]
public struct VertexGradient
{
    public Color topLeft;
    public Color topRight;
    public Color bottomLeft;
    public Color bottomRight;
    ...
}
```

If this property is changed, a Graphic rebuild will occur.

## overrideColorTags

```
public bool overrideColorTags { get; set; }
```

Gets/Sets whether the base color will override the color of the [color](#) tag.

The default value is [false](#), which allows the [color](#) tag to override the base color. Conversely, if this property is [true](#), the [color](#) tag will be ignored and the text will be drawn in the base color.

## characterSpacing / wordSpacing / lineSpacing / paragraphSpacing

```
public float characterSpacing { get; set; }
public float wordSpacing { get; set; }
public float lineSpacing { get; set; }
public float paragraphSpacing { get; set; }
```

Get/Sets the amount of additional space.

Set the font size to [100](#). In other words, a value of [100](#) will result in a space of one character. It is also possible to specify a negative number. The default value for all is [0](#).

[CharacterSpacing](#) is the space between characters, [wordSpacing](#) is the space between words, [lineSpacing](#) is the space between lines, and [paragraphSpacing](#) is the space between paragraphs. Paragraphs are separated by line breaks or the Unicode paragraph separator ([0x2029](#)).

When these properties are changed, Layout rebuild and Graphic rebuild will occur.

## horizontalAlignment / verticalAlignment

```
public HorizontalAlignmentOptions horizontalAlignment { get; set; }
public VerticalAlignmentOptions verticalAlignment { get; set; }
```

Get/Sets the horizontal/vertical alignment.

The definitions of [HorizontalAlignmentOptions](#) and [VerticalAlignmentOptions](#) are as follows

```

public enum HorizontalAlignmentOptions
{
    Left = 0x1, // left alignment
    Center = 0x2, // center alignment
    Right = 0x4, // right alignment
    Justified = 0x8, // Align both ends (do nothing if width is smaller than display area)
    Flush = 0x10, // Equalize (increase spacing if width is less than display area)
    Geometry = 0x20 // centering by geometry (similar behavior to centering by alignByGeometry
in the Text component)
}

public enum VerticalAlignmentOptions
{
    Top = 0x100, // top alignment
    Middle = 0x200, // center alignment
    Bottom = 0x400, // bottom alignment
    Baseline = 0x800, // Baseline alignment for fonts
    Geometry = 0x1000, // Align height of geometry, displayed as Midline in Inspector.
    Capline = 0x2000, // justify height of capital letters
}

```

`HorizontalAlignmentOptions` and `VerticalAlignmentOptions` are bit flags, but it is assumed that one of each is selected.

`Justified` and `Flush` for `HorizontalAlignmentOptions` and `Baseline`, `Geometry` and `Capline` for `VerticalAlignmentOptions` are alphabetic alignments and are rarely used when using Japanese fonts. It is unlikely that they will be used when using Japanese fonts.

If these properties are changed, a Graphic rebuild will occur.

`enableWordWrapping`

```

public bool enableWordWrapping { get; set; }

```

Gets/Sets whether the text will wrap when it reaches the full width.

The default value is the value set in `TMP_Settings.enableWordWrapping` (`true` in the default setting).

When these properties are changed, Layout rebuild and Graphic rebuild will occur.

### overflowMode

```
public TextOverflowModes overflowMode { get; set; }
```

Gets/Sets the behavior when the text reaches the full height.

The definition of `TextOverflowModes` is as follows.

```
public enum TextOverflowModes
{
    Overflow = 0, // overflow
    Ellipsis = 1, // use "..." to indicate the overflow
    Masking = 2, // same behavior as Overflow for now
    Truncate = 3, // truncate the overflow
    ScrollRect = 4, // same behavior as Overflow for now
    Page = 5, // Separate by page. The page number can be specified using pageToDisplay, but note that the first page is 1, not 0.
    Linked = 6 // pass the overflow text to the object set in linkedTextComponent
};
```

When these properties are changed, Layout rebuild and Graphic rebuild will occur.

### horizontalMapping

```
public TextureMappingOptions horizontalMapping { get; set; }
```

If you are not using a texture on the Face or Outline, this will have no effect.

The definition of `TextureMappingOptions` is as follows.

```
public enum TextureMappingOptions
{
    Character = 0, // Texture will be applied to individual characters based on the width and height of each character.
```

```
Line = 1, // Texture will be applied to each line based on its length/width.  
Paragraph = 2, // Texture will be applied to each paragraph based on its length/width.  
MatchAspect = 3 // Texture will be applied to text that matches the aspect ratio of horizontalMapping/verticalMapping.  
};
```

The default value is `TextureMappingOptions.Character`.

If this property is changed, a Graphic rebuild will occur.

### verticalMapping

```
public TextureMappingOptions verticalMapping { get; set; }
```

Gets/Sets the vertical application of the Face and Outline textures (i.e., how the UV2 coordinates are set).

If you are not using a texture for the Face or Outline, it will not be affected.

The default value is `TextureMappingOptions.Character`.

If this property is changed, a Graphic rebuild will occur.

### mappingUvLineOffset

```
public float mappingUvLineOffset { get; set; }
```

Specifies the offset of the texture when `horizontalMapping` or `verticalMapping` is `Line`, `Paragraph`, or `MatchAspect`.

The default value is `0`.

If this property is changed, a Graphic rebuild will occur.

## margin

---

```
public virtual Vector4 margin { get; set; }
```

Gets/Sets the margin of the text area.

If [enableAutoSizing](#) is [true](#), the font size will change according to the text display area, so it is easy to see the effect of this property.

The default value is [\(0, 0, 0, 0\)](#).

If this property is changed, a Graphic rebuild will occur.

## geometrySortingOrder

---

```
public VertexSortingOrder geometrySortingOrder { get; set; }
```

Gets/Sets the text geometry sorting method.

It is used to adjust the way text is displayed when it overlaps.

The definition of [VertexSortingOrder](#) is as follows.

```
public enum VertexSortingOrder
{
    Normal,
    Reverse
};
```

If this property is [Normal](#), the first character of the string (the one on the left) is drawn first, so when characters overlap, the first character is drawn at the bottom. If this property is [Reverse](#), the character on the back (right side) of the string is drawn first, so when characters overlap, the back side is drawn lower. The default value is [Normal](#).

If this property is changed, a Graphic rebuild will occur.

## isTextObjectScaleStatic

---

```
public bool isTextObjectScaleStatic { get; set; }
```

Gets/Sets whether or not to receive callbacks when the scale of this text object or its parent is changed.

The default value is [false](#), but the default setting value can be specified in *TMP Settings*.

If this property is set to [true](#), each object will not be notified when the scale is changed, and SDF will not scale properly, but performance will be improved. In a mobile environment, if the scale changes frequently or there are many text objects, it is a good idea to set this property to [true](#).

## raycastTarget

---

```
public virtual bool raycastTarget { get; set; }
```

Gets/Sets the raycast target (i.e., whether to enable touch detection), which is the same as the Graphic component's [raycastTarget](#), but the default setting can be specified in the *TMP Settings* file.

## richText

---

```
public bool richText { get; set; }
```

Gets/Sets whether or not to use rich text.

The default value is [true](#).

If you don't intend to use rich text, it is better to set this property to [false](#) for performance reasons. If this property is changed, Layout rebuild and Graphic rebuild will occur.

## parseCtrlCharacters

---

```
public bool parseCtrlCharacters { get; set; }
```

Gets/Sets whether the escape sequence should be processed or not.

The default value is `true`, but the default setting value can be specified in *TMP Settings*.

Let's assume that the string "`1\\n2`" is passed to `Text` as shown below.

```
var tmp = GetComponent<TextMeshProUGUI>();
tmp.text = "1\\n2";
```

If this property is `true`, the first line will display `1`, and the second line will display `2`.

If this property is `false`, then `1\\n2` will be displayed in the first line.

If this property is changed, Layout rebuild and Graphic rebuild will occur.

### useMaxVisibleDescender

```
public bool useMaxVisibleDescender { get; set; }
```

Gets/Sets whether the vertical alignment of the text should be aligned with the font descender.

The default value is `true`.

It is used in the *17 - Old Computer Terminal* scene in *Assets/TextMesh Pro/Examples & Extras/Scenes*, but it is not very useful.

If this property is changed, a Graphic rebuild will occur.

### spriteAsset

```
public TMP_SpriteAsset spriteAsset { get; set; }
```

Gets/Sets the sprite asset. Used when no sprite asset is specified in the `<sprite>` tag

For example, a tag such as `<sprite index=0>` displays the 0th sprite of the sprite asset for this property. When this property is changed, a Layout rebuild and a Graphic rebuild are triggered.

## styleSheet

---

```
public TMP_StyleSheet styleSheet { get; set; }
```

Gets/Sets the style sheet that defines the styles that can be used in this text.

If this property is not set, the style sheet set in *TMP Settings* will be used. You can select a style from among the style sheets and set it with [textStyle](#).

If this property is changed, Layout rebuild and Graphic rebuild will occur.

## enableKerning

---

```
public bool enableKerning { get; set; }
```

Sets whether or not to use the kerning pair information contained in the font asset.

The default value is set to [true](#) in *TMP Settings*. However, many font files do not contain any information about kerning pairs.

If this property is changed, Layout rebuild and Graphic rebuild will occur.

## extraPadding

---

```
public bool extraPadding { get; set; }
```

Gets/Sets whether or not padding should be added to the mesh for each character.

The default value is set to [false](#) in *TMP Settings*. If this property is set to [true](#), the mesh of each character will be slightly larger. This prevents the characters from being displayed at all when the font size is very small. Basically, you can leave it [false](#). If this property is changed, a Graphic rebuild will occur.

## autoSizeTextContainer

---

```
public override bool autoSizeTextContainer { get; set; }
```

Gets/Sets whether the size of `RectTransform` should be adjusted to the size of the text display area at rebuild time.

When this property is set to `true`, the behavior is the same as when the `ContentSizeFitter` component is attached and **Preferred Size** is set for **Horizontal Fit** and **Vertical Fit**. If this property is changed, a Layout rebuild will occur.

## canvasRenderer

---

```
public CanvasRenderer canvasRenderer { get; }
```

Get the `CanvasRenderer` that this component is using.

If it is not `null`, the value will be cached.

## maskOffset

---

```
public Vector4 maskOffset { get; set; }
```

Gets/Sets the mask offset...it should, but the value set here is not actually used.

The mask area is determined by its own calculation when the value is set.

## materialForRendering

---

```
public override Material materialForRendering { get; }
```

Get the actual material that the `CanvasRenderer` will use for rendering.

The basic behavior of the `Graphic` component is the same as that of `materialForRendering`.

mesh

---

```
public override Mesh mesh { get; }
```

Get the mesh of the text.

## Public methods of TextMeshProUGUI

### CalculateLayoutInputHorizontal

```
public void CalculateLayoutInputHorizontal();
```

This method is an implementation of [CalculateLayoutInputHorizontal\(\)](#) of the [ILayoutElement](#) interface, but it is empty.

### CalculateLayoutInputVertical

```
public void CalculateLayoutInputVertical();
```

This method is an implementation of [CalculateLayoutInputVertical\(\)](#) of the [ILayoutElement](#) interface, but it is empty.

### ClearMesh

```
public override void ClearMesh();
```

Set the [CanvasRenderer](#)'s mesh to [null](#).

### ComputeMarginSize

```
public override void ComputeMarginSize();
```

Calculate the size of the margin.

### cull

```
public override void Cull(Rect clipRect, bool validRect);
```

Based on the clipping area, pass the setting of whether or not to cull to the `cull` of this `CanvasRenderer`.

This method is an override of `MaskableGraphic's Cull()`.

### ForceMeshUpdate

---

```
public override void ForceMeshUpdate(bool ignoreActiveState = false, bool forceTextReparsing = false);
```

Recreate the text mesh.

### GetModifiedMaterial

---

```
public override Material GetModifiedMaterial(Material baseMaterial);
```

Returns the material to use for rendering.

This method is an implementation of `IMaterialModifier`.

### GetTextInfo

---

```
public override TMP_TextInfo GetTextInfo(string text);
```

Get the `TMP_TextInfo` for the string given in the argument.

`TMP_TextInfo` contains almost all the information needed to render a string.

### Rebuild

---

```
public override void Rebuild(CanvasUpdate update);
```

Run the Canvas rebuild.

In [PreLayout](#) stage, [RectTransform](#) sizing is performed when [autoSizeTextContainer](#) is [true](#).

In the [PreRender](#) stage, important (and demanding) processes such as text parsing and geometry re-creation are performed, and if the material is dirty, the material update process is performed.

## RecalculateClipping

---

```
public override void RecalculateClipping();
```

Recalculate the clipping area.

This method is an implementation of the [IClippable](#) interface.

In fact, it calls [RecalculateClipping](#) of the parent class [MaskableGraphic](#) as it is.

## SetAllDirty

---

```
public override void SetAllDirty();
```

Call [SetLayoutDirty\(\)](#), [SetMaterialDirty\(\)](#), and [SetVerticesDirty\(\)](#) to cause Layout rebuild and Graphic rebuild to occur later.

## SetLayoutDirty

---

```
public override void SetLayoutDirty();
```

Mark the Layout as dirty.

If a callback has been set in [RegisterDirtyLayoutCallback\(\)](#), it will be called. If the layout is marked as dirty, a layout rebuild will be performed via [CanvasUpdateRegistry.PerformUpdate\(\)](#). The load during this layout rebuild is measured as **Layout** in the UI area of the **Profiler**.

## SetMaterialDirty

---

```
public override void SetMaterialDirty();
```

Mark the material as dirty.

If a callback has been set up with [RegisterDirtyMaterialCallback\(\)](#), it will be called. If the material is marked as dirty, [Rebuild\(\)](#) will be called via [CanvasUpdateRegistry.PerformUpdate\(\)](#) to reconfigure the material. This is the Graphic rebuild. The load during this graphic rebuild is measured as **Render** in the UI area of the **Profiler**.

## SetVerticesDirty

---

```
public override void SetVerticesDirty();
```

Mark the vertex as dirty.

If a callback has been set up with [RegisterDirtyMaterialCallback\(\)](#), it will be called. If a vertex is marked as dirty, a Graphic rebuild will be performed via [CanvasUpdateRegistry.PerformUpdate\(\)](#) as if the material was dirty, and the load will be measured as a **Render** in the UI area of the Profiler. The load is measured as Render in the UI area of the Profiler.

## UpdateFontAsset

---

```
public void UpdateFontAsset();
```

Load pre-configured font assets.

If the configured font asset is [null](#), the default font asset will be loaded. If the default font asset is defined in *TMP Settings*, and it is also [null](#), then *Fonts & Materials/LiberationSans SDF* will be loaded. When this method is called, the material will be considered dirty.

## UpdateGeometry

```
public override void UpdateGeometry(Mesh mesh, int index);
```

Set up a text mesh.

If `index` is 0, the `mesh` is passed to the `CanvasRenderer` with `SetMesh()`; if `index` is greater than 0, the `mesh` is passed to the `CanvasRenderer` of the subtext object with `SetMesh()`. This method is different from the `UpdateGeometry()` method of the `Graphic` class, which takes no arguments.

## UpdateMeshPadding

```
public override void UpdateMeshPadding();
```

Recalculate the padding.

This method is called when a shader or material is changed from the script.

## UpdateVertexData

```
public override void UpdateVertexData();
public override void UpdateVertexData(TMP_VertexDataUpdateFlags flags);
```

Pass the internal vertex data to the `CanvasRenderer` with `SetMesh()`.

If `TMP_VertexDataUpdateFlags` is specified, data will be set on the mesh according to the flags.

The definition of `TMP_VertexDataUpdateFlags` is as follows.

```
public enum TMP_VertexDataUpdateFlags
{
    None = 0x0,
    Vertices = 0x1,
    Uv0 = 0x2,
    Uv2 = 0x4,
```

```
    Uv4 = 0x8,  
    Colors32 = 0x10,  
    All = 0xFF  
};
```

## Events of TextMeshProUGUI

### OnPreRenderText

```
public override event Action<TMP_TextInfo> OnPreRenderText;
```

You can set the callback to be called in the [PreRender](#) stage of the Canvas rebuild.

The sample code is shown below.

```
using UnityEngine;
using TMPro;

// Receive a callback when the PreRender stage of the Canvas rebuild is executed.
[RequireComponent(typeof(TextMeshProUGUI))]
public class TMPOnPreRenderTextSample : MonoBehaviour
{
    private TextMeshProUGUI tmpUGUI;

    private void Start()
    {
        tmpUGUI = GetComponent<TextMeshProUGUI>();
        tmpUGUI.OnPreRenderText += OnPreRenderText;
    }

    private void OnDestroy()
    {
        if (tmpUGUI != null)
        {
            tmpUGUI.OnPreRenderText -= OnPreRenderText;
        }
    }

    public void OnPreRenderText(TMP_TextInfo textInfo)
    {
        Debug.Log("PreRender stage of Canvas rebuild executed");
        Debug.Log("Object " + textInfo.textComponent.name);
        Debug.Log("Text " + textInfo.textComponent.text);
    }
}
```

```
        Debug.Log("Font " + textInfo.textComponent.font.name);  
    }  
}
```

## Comparison of Text and TextMesh Pro

In general, the reasons for adopting TextMesh Pro would be as follows.

- I want it to look good (especially the spacing and outlines).
- There are only a few types of letterforms to use, and they are almost fixed at build time.
- Available in Japanese only.
- No need for engineering resources to extend the standard [Text](#) on your own.

On the other hand, the reasons for adopting the standard [Text](#) component are as follows

- Rendering speed is more important than visual quality.
- Multiple languages are supported, and the character set used cannot be determined at build time.
- The available memory is severe.

For casual games or individually developed games, TextMesh Pro would be a good choice. On the other hand, if you are developing a large scale mobile social game with a large amount of text resources and a complex workflow, or if you are developing a VR application where performance is important, you may want to consider using the standard [Text](#) component (and your own extensions to the [Text](#) component) as the core, and if there is room for performance and workflow, you may want to adopt TextMesh Pro. If there is room for performance and workflow, it may be a good idea to adopt TextMesh Pro.

## Chapter 7 Selectable

### Selectable

```
[AddComponentMenu("UI>Selectable", 70)]
[ExecuteAlways]
[SelectionBase]
[DisallowMultipleComponent]
public class Selectable : UIBehaviour, IMoveHandler, IPointerDownHandler, IPointerUpHandle
r, IPointerEnterHandler, IPointerExitHandler, ISelectHandler, IDeselectHandler, IEEventSystemH
andler
```

Selectable is a component that represents a selectable UI element.

The components that inherit from **Selectable** are **Button**, **Toggle**, **Slider**, **Dropdown**, **InputField**, **Scrollbar**, and many more. Before we look at those components, let's take a closer look at **Selectable**.

## Selectable selection state

Selectable has five selection states: `Normal`, `Highlighted`, `Pressed`, `Selected`, and `Disabled`, which are defined by the enumerated type `Selectable.SelectionState`.

```
/// <summary>
/// Enumeration of object selection states
/// </summary>
protected enum SelectionState
{
    /// <summary>
    /// UI object is selectable
    /// </summary>
    Normal,

    /// <summary>
    /// UI object is highlighted
    /// </summary>
    Highlighted,

    /// <summary>
    /// UI object is pressed
    /// </summary>
    Pressed,

    /// <summary>
    /// UI object is selected
    /// </summary>
    Selected,

    /// <summary>
    /// UI object is not selectable
    /// </summary>
    Disabled,
}
```

## Static variables for Selectable

### allSelectableCount

```
public static int allSelectableCount { get; }
```

Get the number of Selectable components that are currently active.

### allSelectablesArray

```
public static Selectable[] allSelectablesArray { get; }
```

Copies and returns an array of the currently active Selectables.

Note that `new` is executed internally, so *GC Alloc* will be generated. If you use the static method `AllSelectablesNoAlloc()` (described below), `new` will not be executed because it passes an already allocated array as an argument.

## Properties of Selectable

transition

```
public Selectable.Transition transition { get; set; }
```

Gets/Sets the type of transition to be applied to the `targetGraphic` when the state changes. There are four types of transitions: `None`, `ColorTint`, `SpriteSwap`, and `Animation`, with `ColorTint` being the default.

*Runtime/UI/Core>Selectable.cs*

```
/// <summary>
/// Mode of the Selectable transition
/// </summary>
public enum Transition
{
    /// <summary>
    /// No transition
    /// </summary>
    None,

    /// <summary>
    /// Transitions by color
    /// </summary>
    ColorTint,

    /// <summary>
    /// Transition by Sprite Replacement
    /// </summary>
    SpriteSwap,

    /// <summary>
    /// Transition by animating with Animator
    /// </summary>
    Animation
}
```

If the [Transition](#) is [ColorTint](#), the color of each state is defined by the [ColorBlock](#) structure, which can be obtained/set by the [colors](#) property. 600 KB of *GC Alloc* will be generated during the transition in the case of [ColorTint](#).

If the transition is [SpriteSwap](#), the [Sprite](#) in each state is defined in the [SpriteState](#) structure, and can be obtained/set in the [spriteState](#) property.

When the transition is an [animation](#), the animation trigger for each state is defined in the [AnimationTriggers](#) class and can be obtained/set in the [animationTriggers](#) property.

## colors

```
public ColorBlock colors { get; set; }
```

Gets/Sets the [ColorBlock](#).

The [ColorBlock](#) is a structure that stores the transition states based on color. It contains the color for the **Normal**, **Highlight**, **Pressed (Selected)**, and **Disabled** states, the color coefficient (the higher the coefficient, the brighter the color; the range is from 1 to 5), and the fade animation time. Fade animation time. Their default values are defined as follows.

```
public struct ColorBlock : IEquatable<ColorBlock>
{
    ...
    public static ColorBlock defaultColorBlock
    {
        get
        {
            var c = new ColorBlock
            {
                m_NormalColor = new Color32(255, 255, 255, 255),
                m_HighlightedColor = new Color32(245, 245, 245, 255),
                m_PressedColor = new Color32(200, 200, 200, 255),
                m_SelectedColor = new Color32(245, 245, 245, 255),
                m_DisabledColor = new Color32(200, 200, 200, 128),
                colorMultiplier = 1.0f,
                fadeDuration = 0.1f
            };
            return c;
        }
    }
}
```

```
    }  
}  
...  
}
```

For example, if you wanted to change the color of **Pressed** from a script, you could do something like this

```
using UnityEngine;  
using UnityEngine.UI;  
  
public class SetButtonPressedColorSample : MonoBehaviour  
{  
    public Button button;  
  
    // Change the color of the Button when it is Pressed.  
    public void SetButtonPressedColor(Color pressedColor)  
    {  
        ColorBlock colorBlock = button.colors;  
        colorBlock.pressedColor = pressedColor;  
  
        // ColorBlock is a structure, not a class, so we need to replace it entirely  
        button.colors = colorBlock;  
    }  
    ...  
}
```

spriteState

```
public SpriteState spriteState { get; set; }
```

Gets/Sets the [SpriteState](#).

[SpriteState](#) is a structure that stores the state of transitions caused by sprite replacement, and it holds the [Sprite](#) corresponding to the **Normal**, **Highlight**, **Pressed (Selected)**, and **Disabled** states.

## animationTriggers

---

```
public AnimationTriggers animationTriggers { get; set; }
```

Gets/Sets AnimationTriggers.

AnimationTriggers is a class that holds the state (name) of animation transitions. In Selectable, there are animation triggers called Normal, Highlighted, Pressed, Selected, and Disabled, and AnimationTriggers holds their names.

For example, if you want to set the state of the [Selectable Animator](#) to Pressed from a script, you can get the name of the trigger that corresponds to Pressed from [animationTriggers](#) and pass it to [Animator](#).

```
using UnityEngine;
using UnityEngine.UI;

public class AnimationTriggersSample : MonoBehaviour
{
    public Animator buttonAnimator;
    public Button button;

    public void MakeButtonStatePressed()
    {
        // Make the button state transition to the Pressed state.
        // (This may be useful when making a tutorial)
        buttonAnimator.SetTrigger(button.animationTriggers.pressedTrigger);
    }
}
```

## animator

---

```
public Animator animator { get; }
```

Gets the [Animator](#) that is attached to this [GameObject](#).

Be careful about performance, because internally you are just calling `GetComponent<Animator>()`, and the obtained value is not cached.

## interactable

---

```
public bool interactable { get; set; }
```

Gets/Sets whether the UI element can be selected or not.

For example, if you set the `Button`'s `interactable` to `false`, the `Button` will be in the **Disabled** state and cannot be pressed.

If `interactable` is `false`, the current selection state is **Disabled**; if it is `true`, the current selection state is **Pressed**, **Selected**, **Highlighted**, or **Normal**, depending on various other conditions. If it is true, the state will be Pressed, Selected, Highlighted, or **Normal, depending on various other conditions**. Next, depending on the current selection state, a transition will be initiated, and depending on the transition type, either a `ColorTint`, `SpriteSwap`, or `Animation` transition will be performed.

Also, if the `EventSystem`'s `currentSelectedGameObject` is this object, then `currentSelectedGameObject` will be set to `null`.

## navigation

---

```
public Navigation navigation { get; set; }
```

Specify/get the navigation behavior of this `Selectable`.

Navigation means to change the UI element to be focused when the up, down, left, right or right key is pressed by key operation instead of touch operation. Each `Selectable` specifies which `Selectable` the focus moves to when the up, down, left, right, or right key is pressed, and navigation is performed based on this information.

Let's place the `Button` as shown below, and set `ButtonCenter` to **First Selected** in `EventSystem` from **Inspector**.

If you enter **Play** mode in this state, `ButtonCenter` is initially selected, but you can select `ButtonUp`, `ButtonDown`, `ButtonLeft`, and `ButtonRight` with the up, down, left, and right keys on the keyboard.

The value type of navigation is `Navigation` structure, which is defined in `UI/Core/Navigation.cs`. There are five modes of `navigation`, which are defined as `Navigation`.

```
public struct Navigation : IEquatable<Navigation>
{
    /// <summary>
    /// Navigation mode enumeration type
    /// </summary>
    /// <remarks>
    /// The following values look like they are not flags (of bits), but they are actually flags. This is
    /// because Automatic is both Horizontal and Vertical mode.
    /// </remarks>
    public enum Mode
    {
        /// <summary>
        /// Navigation is not allowed for this object.
        /// </summary>
        None = 0,

        /// <summary>
        /// Horizontal navigation
        /// </summary>
        /// <remarks>
        /// Navigation is only allowed when a left or right movement event occurs
        /// </remarks>
        Horizontal = 1,

        /// <summary>
        /// Vertical navigation
        /// </summary>
        /// <remarks>
        /// Navigation is only allowed when a vertical movement event occurs
        /// </remarks>
        Vertical = 2,

        /// <summary>
        /// Auto navigation
        /// </summary>
        /// <remarks>
        /// Try to find the "best" next object. This will be based on a sensory rule of thumb.
        /// </remarks>
    }
}
```

```
Automatic = 3,  
  
/// <summary>  
/// Explicit navigation  
/// </summary>  
/// <remarks>  
/// The user should explicitly specify what will be selected for each move event  
/// </remarks>  
Explicit = 4,  
}
```

If the mode is [Horizontal](#), [Vertical](#), or [Automatic](#), the next object to be selected is determined by the [FindSelectable](#) method; if it is [Explicit](#), the next focus object corresponding to the explicitly specified Vertical, Vertical, or Automatic mode is selected. In the case of [Explicit](#), the next focus object corresponding to each of the explicitly specified vertical and horizontal directions is selected.

## targetGraphic

---

```
public Graphic targetGraphic { get; set; }
```

Gets/Sets the [Graphic](#) to be used by this [Selectable](#).

Transitions are applied to this [Graphic](#). Normally, the [Graphic](#) attached to the same [GameObject](#) is used.

## image

---

```
public Image image { get; set; }
```

Gets/Sets the [Image](#) to be used by this [Selectable](#).

The entity is a [targetGraphic](#) casted to an [Image](#).

## Static methods of Selectable's

```
public static int AllSelectablesNoAlloc(Selectable[] selectables);
```

Get all [Selectables](#) as in [allSelectablesArray](#).

However, this method does not allocate memory internally. If the size of the argument array is smaller than the number of all [Selectables](#), the [Selectables](#) are copied for the length of the array. Otherwise, only all [Selectables](#) are copied. The return value of this method is the number of [Selectables](#) actually copied into the array.

## Public methods of Selectable's

### FindSelectable

```
public Selectable FindSelectable(Vector3 dir);
```

Locate and return an adjacent [Selectable](#) object in a specific direction for navigation.

The internal implementation is to scan all [Selectable](#) objects (whose [navigation](#) is not [Navigation.Mode.None](#)) and find the closest object based on the distance and inner product of each.

### FindSelectableOnDown

```
public virtual Selectable FindSelectableOnDown();
```

Locate and return the underlying [Selectable](#) object for navigation.

If [navigation](#) is [Navigation.Mode.Explicit](#), it will return the [Selectable](#) set as bottom. If it contains the [Navigation.Mode.Vertical](#) flag, [FindSelectable\(\)](#) will return the lower [Selectable](#). Otherwise [null](#) is returned.

### FindSelectableOnLeft

```
public virtual Selectable FindSelectableOnLeft();
```

Locate and return the [Selectable](#) object to the left for navigation.

If [navigation](#) is [Navigation.Mode.Explicit](#), it will return the [Selectable](#) set as left. If [navigation](#) contains the flag [Navigation.Mode.Horizontal](#) flag, [FindSelectable\(\)](#) will return the leftside [Selectable](#). Otherwise [null](#) is returned.

## FindSelectableOnRight

---

```
public virtual Selectable FindSelectableOnRight();
```

Locate and return the [Selectable](#) object to the right for navigation.

If [navigation](#) is [Navigation.Mode.Explicit](#), it will return the [Selectable](#) set as right. If [navigation](#) contains the flag [Navigation.Mode.Horizontal](#) flag, [FindSelectable\(\)](#) will return the rightside [Selectable](#). Otherwise [null](#) is returned.

## FindSelectableOnUp

---

```
public virtual Selectable FindSelectableOnUp();
```

Locate and return the [Selectable](#) object above for navigation.

If [navigation](#) is [Navigation.Mode.Explicit](#), it will return the [Selectable](#) set as bottom. If it contains the [Navigation.Mode.Vertical](#) flag, [FindSelectable\(\)](#) will return the upside [Selectable](#). Otherwise [null](#) is returned.

## IsInteractable

---

```
public virtual bool IsInteractable();
```

Returns whether or not the UI element can be selected.

Checks not only its own [interactable](#) property, but also the [interactable](#) of the [CanvasGroup](#) (if it is under the influence of the [CanvasGroup](#)), and returns [true](#) only if both are [true](#). Otherwise, it returns [false](#).

## OnSelect

---

```
public virtual void OnSelect(BaseEventData eventData);
```

When this object is selected, it will be called from [StandAloneInputModule](#) and so on. [StandAloneInputModule](#) is described in detail in *Chapter 10 EventSystem*.

This method is a method that implements the [ISelectHandler](#) interface. It changes the current selection state [currentSelectionState](#) (to [SelectionState.Selected](#)) and then performs the transition.

#### OnDeselect

```
public virtual void OnDeselect(BaseEventData eventData);
```

Called by [StandAloneInputModule](#) and others when this object is deselected.

This method is a method that implements the [IDeselectHandler](#) interface. It changes the current selection state [currentSelectionState](#) (so that [SelectionState.Selected](#) is released) and then performs the transition.

#### OnMove

```
public virtual void OnMove(AxisEventData eventData);
```

Called by [StandAloneInputModule](#) and other modules when moving up, down, left, or right by key input.

This method is a method that implements the [IMoveHandler](#) interface. It determines the direction to look for the [Selectable](#) object based on [AxisEventData](#), and navigates to the next [Selectable](#).

#### OnPointerDown

```
public virtual void OnPointerDown(PointerEventData eventData);
```

Called by [StandAloneInputModule](#) and others when a finger is touched or a mouse is pressed on a touch panel.

This method is an implementation of the [IPointerDownHandler](#) interface.

When this method is called, it registers itself with [EventSystem](#) as the currently selected object, changes the current selection state [currentSelectionState](#) (to [SelectionState.Pressed](#)), and then performs the transition.

### OnPointerUp

---

```
public virtual void OnPointerUp(PointerEventData eventData);
```

Called by [StandAloneInputModule](#) and others when the finger is released from the touch panel or the mouse is released.

This method is an implementation of the [IPointerUpHandler](#) interface.

Change the current selection state [currentSelectionState](#) (so that [SelectionState.Pressed](#) is released) and then perform the transition.

### OnPointerEnter

---

```
public virtual void OnPointerEnter(PointerEventData eventData);
```

Called by [StandAloneInputModule](#), etc. when a touch panel finger or mouse enters the area of a UI element.

This method is an implementation of the [IPointerEnterHandler](#) interface.

Change the current selection state [currentSelectionState](#) (to [SelectionState.Selected](#), etc.) and then perform the transition.

Raycasting is used to determine whether or not the element has entered the area of the UI element. A typical process flow is as follows.

1. The touch panel is touched.
2. Ask the [StandAloneInputModule](#) to throw a raycast to the [EventSystem](#) based on the touched location.
3. The [EventSystem](#) asks the [GraphicRaycaster](#) component, which is attached to the same [GameObject](#) as the [Canvas](#), to throw a raycast.

4. The [GraphicRaycaster](#) calls [Raycast\(\)](#) on all [Graphics](#) under it to determine if the corresponding location is in its own UI area. In most cases, the rectangle of [RectTransform](#) is used for this purpose.

### OnPointerExit

---

```
public virtual void OnPointerExit(PointerEventData eventData);
```

Called by [StandAloneInputModule](#), etc. when the touch screen finger or mouse leaves the UI element area.

This method is an implementation of the [IPointerExitHandler](#) interface.

Change the current selection state [currentSelectionState](#) (so that [SelectionState.Highlighted](#) is released) and then perform the transition.

### Select

---

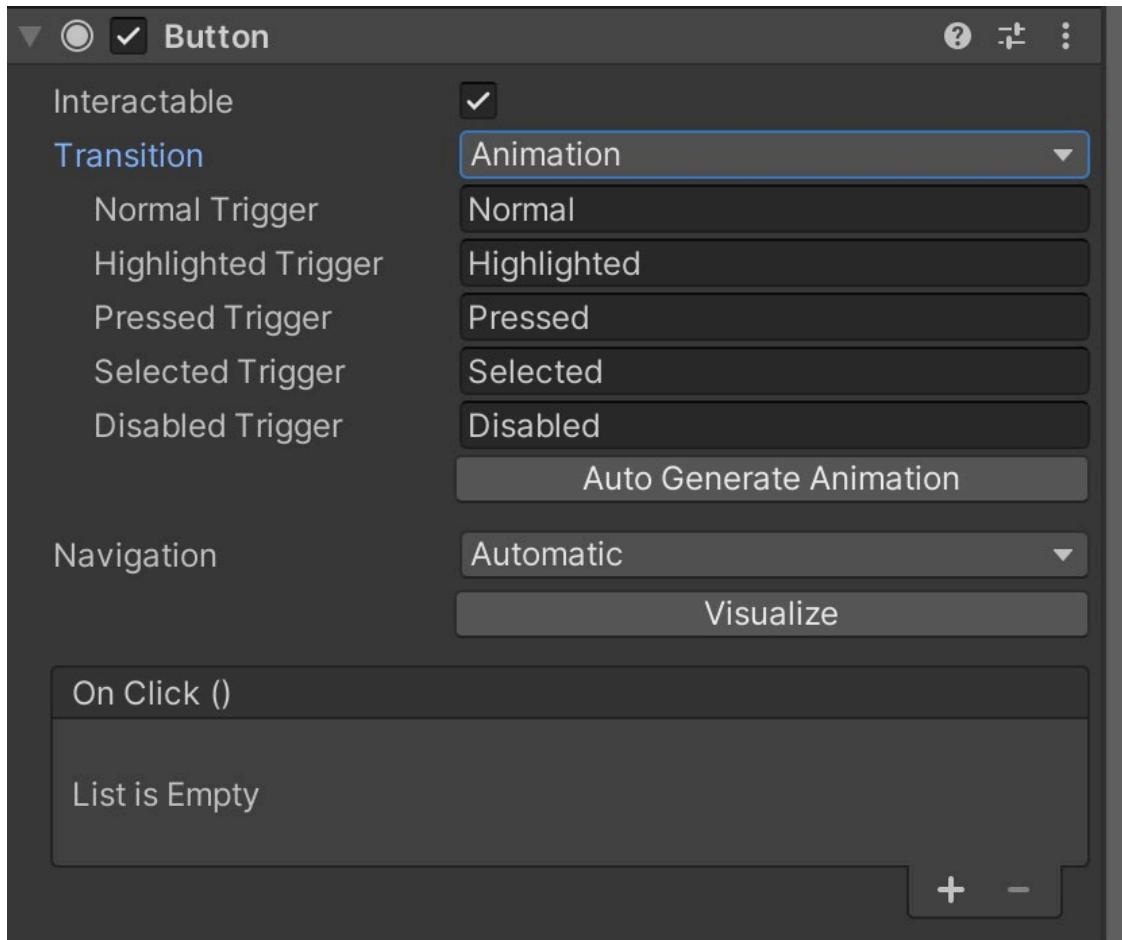
```
public virtual void Select();
```

Registers itself with [EventSystem](#) as the currently selected object.

It is used internally by the [DropDown](#) component. It is rare to call this method explicitly.

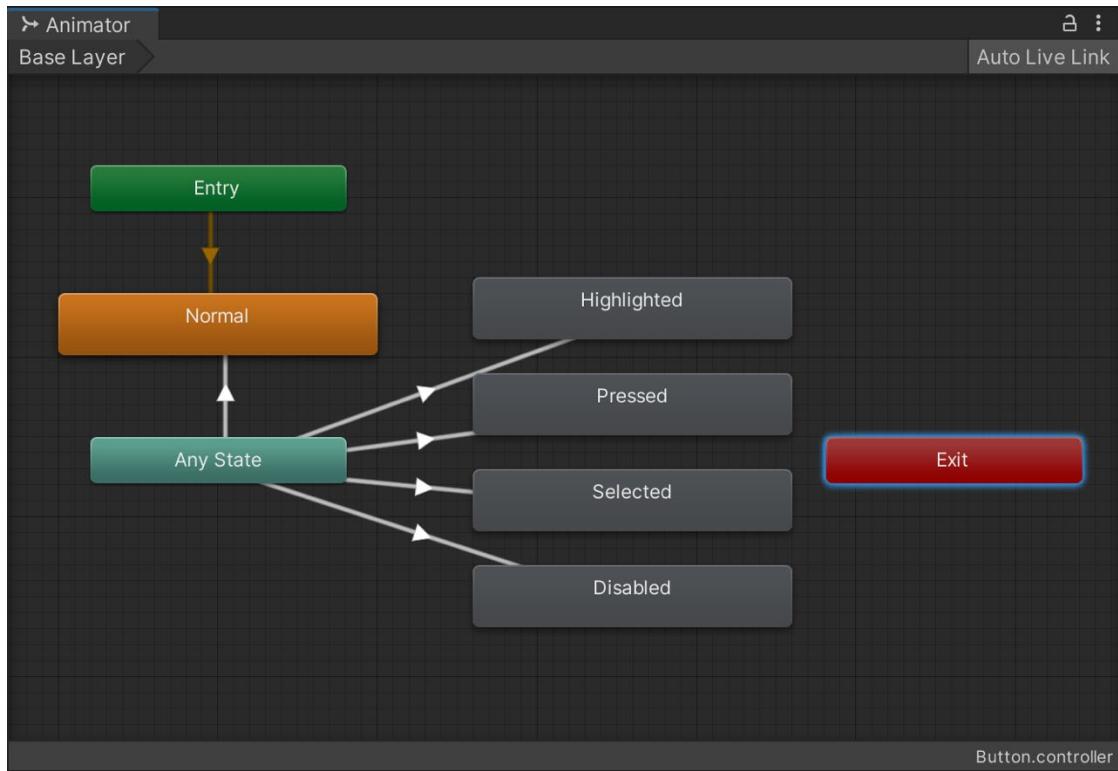
## Animation settings

If [Animator](#) is not attached or `animator.runtimeAnimatorController` is null, **Auto Generate Animation** button appears in the **Inspector**.

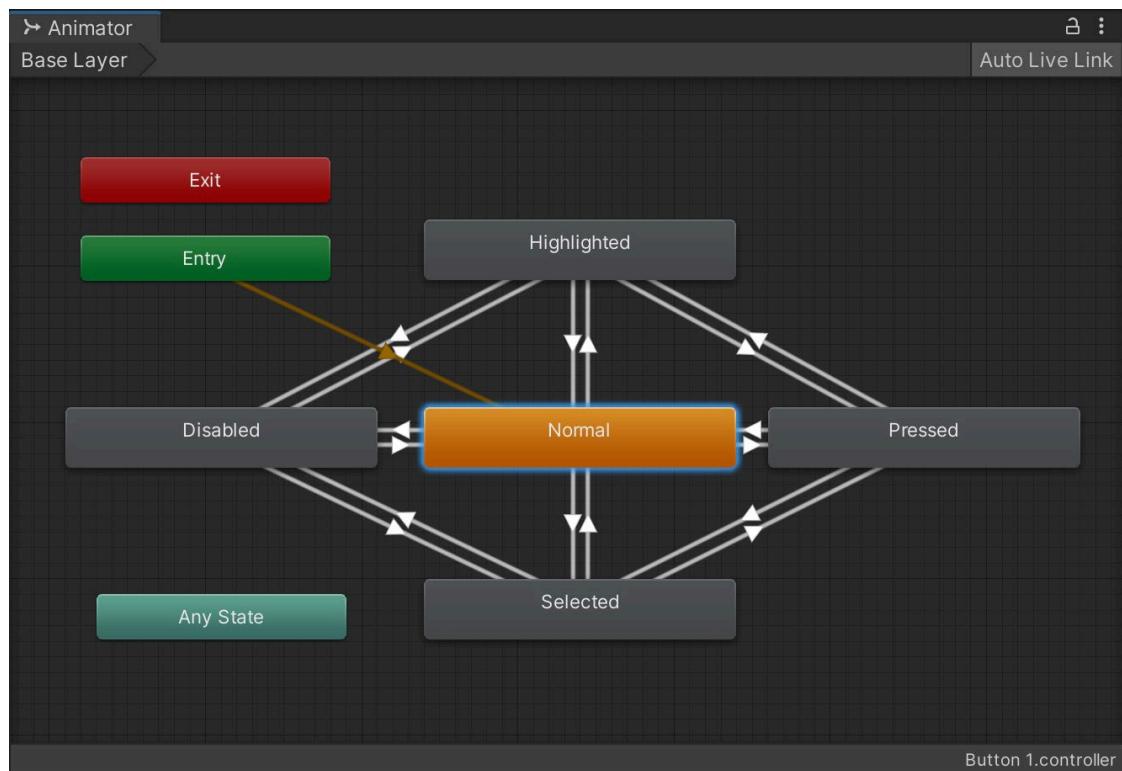


When this button is pressed, an [AnimationController](#) is created in the project, and the Animator set to `runtimeAnimatorController` is attached to the [AnimationController](#).

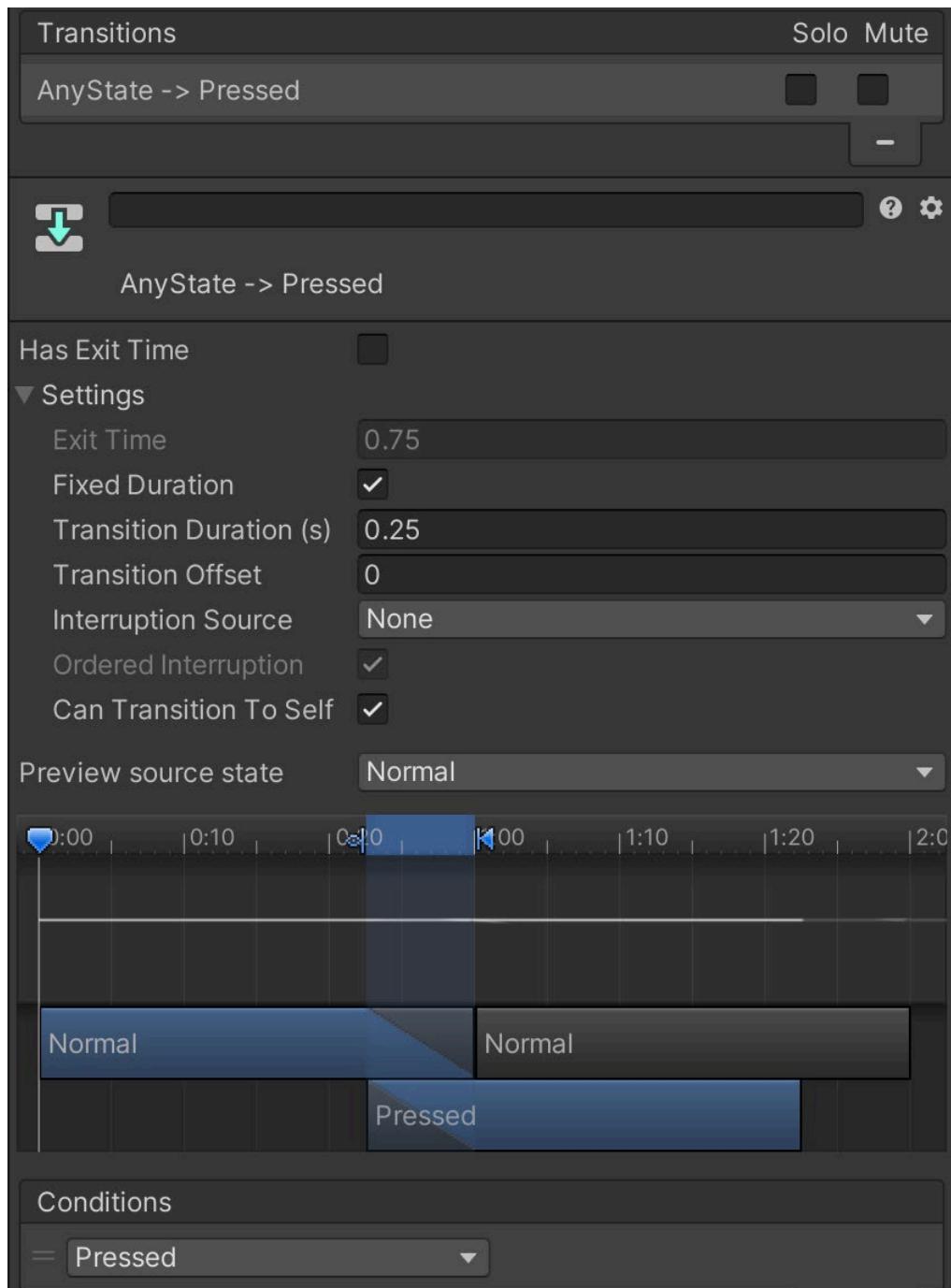
The **AnimationController** created here has a default state of **Normal**, which is a simple animation that transitions to other states via **Any State**.



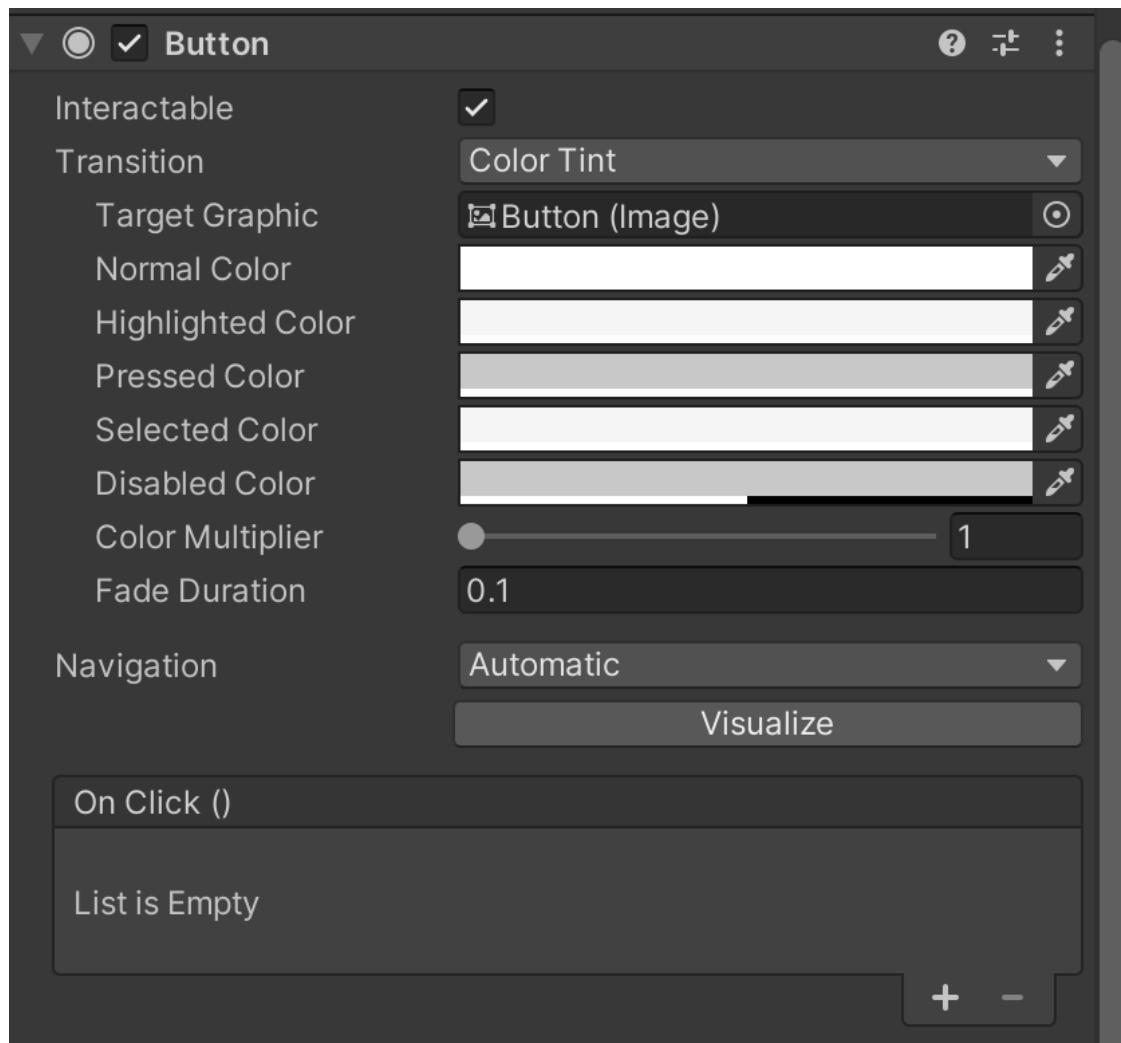
If you want to achieve elaborate animations, you should customize the [AnimationController](#), which allows you to fine-tune the animation depending on the transition pattern. For example, if you want to transition from **Disabled** to another [State](#), you may want to set the animation time to zero. In order to achieve this, it is better to define transitions between individual [State](#) rather than through **Any State**.



In addition, the default animation has a slightly longer **Transition Duration** of **0.25** seconds, so this is another point to customize.



## Button component



```
[AddComponentMenu("UI/Button", 30)]  
public class Button : Selectable, IMoveHandler, IPointerDownHandler, IPointerUpHandler, IPo  
interEnterHandler, IPo  
interExitHandler, ISelectHandler, IDeselectHandler, IPointerClickHandler,  
ISubmitHandler, IEventSystemHandler
```

Most of the basic functionality of the **Button** component is implemented in the **Selectable** component. The additional functions implemented are to control the behavior of the button when clicked and when it is submitted.

## Properties of Button

onClick

```
public Button.ButtonClickedEvent onClick { get; set; }
```

Get/Sets the callback when the button is clicked. A sample code of how to use it is shown below.

```
using UnityEngine;
using UnityEngine.UI;

[RequireComponent(typeof(Button))]
public class ButtonClickSample : MonoBehaviour
{
    void Start()
    {
        var button = GetComponent<Button>();

        // Add a callback for when the button is clicked
        button.onClick.AddListener(() =>
        {
            Debug.Log("Button was clicked");
        });
    }
}
```

## Public methods of Button

### OnPointerClick

```
public virtual void OnPointerClick(PointerEventData eventData);
```

Called by [StandAloneInputModule](#) and others when a button is left or right clicked.

This method is an implementation of the [IPointerClickHandler](#) interface.

It is also called when the mouse is right-clicked, so it does not do anything like putting the mouse in the **Pressed** state.

### OnSubmit

```
public virtual void OnSubmit(BaseEventData eventData);
```

It is called by [StandAloneInputModule](#) and other modules when the Enter key is pressed on the keyboard.

This method is an implementation of the [ISubmitHandler](#) interface.

There are two ways to change the input for Submit.

How to change the Submit button in Axes

1. Open *Edit -> Project Settings -> Input*.
2. Expand the Axes section.
3. If Submit does not exist, increase the Size by one, add a section, and name the added section **Submit**.
4. Change the **Positive Button** of **Submit** from [return](#) to something else (e.g. [space](#)).

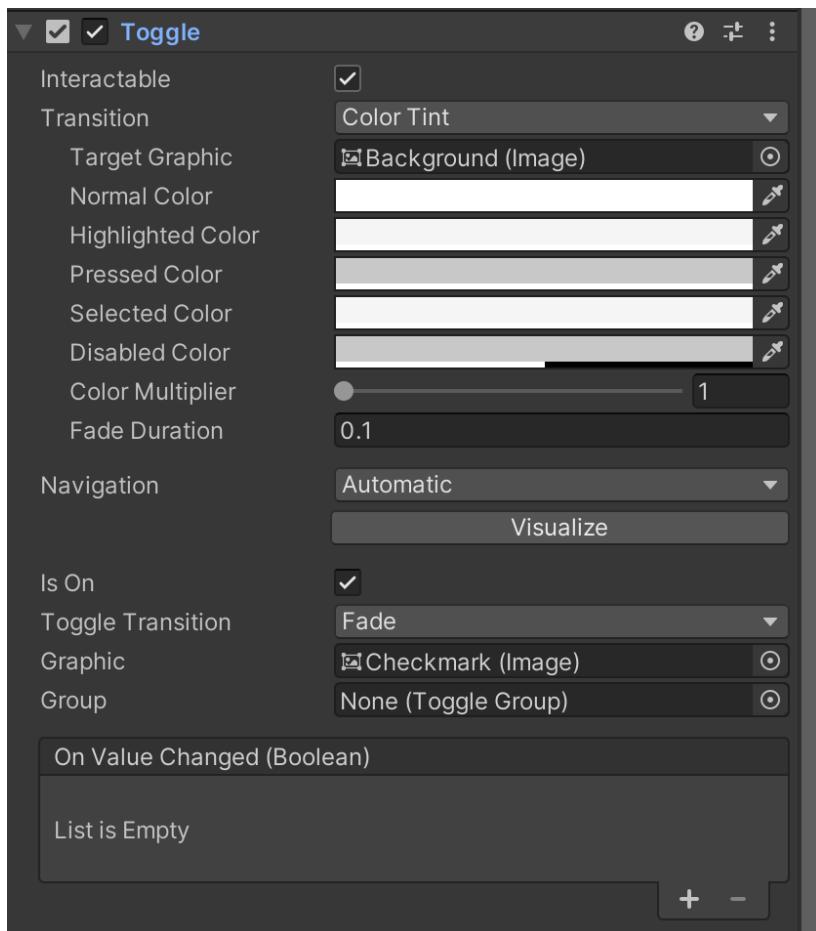
How to change the Submit Button in Input Manager

1. Select the [EventSystem](#) in the [Hierarchy](#).

2. Change the **Submit Button** in the **Inspector**'s **StandaloneInputManager** from **Submit** (as defined in the Axes section of Edit -> Project Settings -> Input) to something else (e.g. Fire1).

3.

### Toggle component



```
[AddComponentMenu("UI/Toggle", 31)]
[RequireComponent(typeof(RectTransform))]
public class Toggle : Selectable, IMoveHandler, IPointerDownHandler, IPointerUpHandler, IPoi
ntrEnterHandler, IPointerExitHandler, ISelectHandler, IDeselectHandler, IPointerClickHandler,
ISubmitHandler, IEventSystemHandler, ICanvasElement
```

The **Toggle** component inherits from the **Selectable** component and controls the **Graphic** for displaying the ON/OFF state. The Toggle component is very quirky and has many behaviors that cannot be customized, so it is not recommended to use it as-is. It is more likely to get the desired

behavior by creating your own toggle component that inherits from [Selectable](#), or by customizing [Button](#).

## Public variables of Toggle

graphic

```
public Graphic graphic;
```

This is the [Graphic](#) that is affected when [Toggle](#) is turned on or off.

The [graphic](#) does not have to be a child element of [Toggle](#). If you want to affect something other than this [Graphic](#) when switching, you need to set a callback to [onValueChanged](#) as described below.

onValueChanged

```
public Toggle.ToggleEvent onValueChanged;
```

This is the callback that is called when the [Toggle](#) is switched on or off.

[Toggle.ToggleEvent](#) is a [UnityEvent<bool>](#), so it can be delegated with [AddListener](#). here is a sample that uses [onValueChanged](#) to log to the console when the toggle is switched The following is a sample that uses [onValueChanged](#) to output a log to the console when the toggle is switched.

```
using UnityEngine;
using UnityEngine.UI;

[RequireComponent(typeof(Toggle))]
public class ToggleCallbackSample : MonoBehaviour
{
    private void Start()
    {
        var toggle = GetComponent<Toggle>();

        toggle.onValueChanged.AddListener((isOn) =>
        {
            Debug.Log("Toggle switched " + isOn);
        });
    }
}
```

```
}
```

## toggleTransition

```
public Toggle.ToggleTransition toggleTransition;
```

Specifies the transition processing when the [Toggle](#) is switched.

There are two types of transitions: [None](#), which immediately switches the [Graphic](#)'s alpha value from [0](#) to [1](#), and [Fade](#), which switches the [Graphic](#)'s alpha value over a period of one second. The other is [Fade](#), which switches the graphic alpha over a period of one second. Unfortunately, the interval of one second for [Fade](#) cannot be customized.

[Toggle.ToggleTransition](#) enumeration is defined as follows.

```
/// <summary>
/// Display setting when toggle is turned on or off
/// </summary>
public enum ToggleTransition
{
    /// <summary>
    /// Show/hide the toggle immediately
    /// </summary>
    None,

    /// <summary>
    /// Gradually fade in/fade out the toggle
    /// </summary>
    Fade
}
```

## Toggle Properties

### group

```
public ToggleGroup group { get; set; }
```

Gets/Sets the [ToggleGroup](#) to which this [Toggle](#) belongs.

[ToggleGroup](#) is a component for grouping multiple [Toggles](#) and controlling only one [Toggle](#) to be turned on.

### isOn

```
public bool isOn { get; set; }
```

Gets/Sets whether this [Toggle](#) is ON or not.

When it is set to the ON state ([true](#)), the following characteristic operations are performed.

1. If [Toggle](#) is set to ON, notify the [group](#) that you are ON.
2. Even if [Toggle](#) is set to OFF, if no toggle in the [group](#) is in the ON state and no toggle in the ON state is allowed to exist ([ToggleGroup.allowSwitchOff](#) is [false](#)), it will turn itself ON and notify the [group](#) that it has turned itself ON. [ToggleGroup.allowSwitchOff](#) is [false](#).
3. If [toggleTransition](#) is [ToggleTransition.None](#), immediately change the [graphic](#)'s alpha to 1 (ON) or 0 (OFF). If [toggleTransition](#) is other than [ToggleTransition.None](#), it will gradually crossfade the alpha value over a period of one second.
4. Call [onValueChanged](#).

## Toggle's public methods

### OnPointerClick

```
public virtual void OnPointerClick(PointerEventData eventData);
```

Called by `StandAloneInputModule` and others when a button is left or right clicked.

This method is an implementation of the `IPointerClickHandler` interface.

If it was a left click, toggle the state of the toggle.

### OnSubmit

```
public virtual void OnSubmit(BaseEventData eventData);
```

It is called by `StandAloneInputModule` and other modules when the Enter key is pressed on the keyboard.

This method is an implementation of the `ISubmitHandler` interface.

When this method is called, it switches the state of the toggle.

### Rebuild

```
public virtual void Rebuild(CanvasUpdate executing);
```

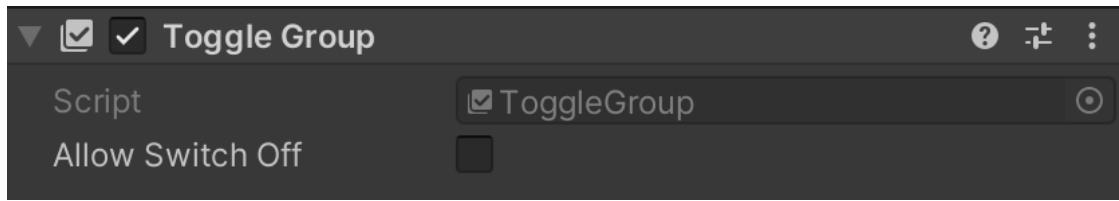
Only when the Editor is running, call `onValueChanged` during the `PreLayout` stage.

### SetIsOnWithoutNotify

```
public void SetIsOnWithoutNotify(bool value)
```

Switch between ON and OFF states without calling `onValueChanged`.

## ToggleGroupcomponent



```
[AddComponentMenu("UI/Toggle Group", 32)]  
[DisallowMultipleComponent]  
public class ToggleGroup : UIBehaviour
```

**ToggleGroup** is a component for grouping multiple [Toggles](#) together and controlling only one [Toggle](#) to be turned on. [ToggleGroup](#) does not necessarily need to be in the parent hierarchy of [Toggle](#).

This component uses [Linq](#), which causes *GC Alloc* in some of the method calls.

## Properties of ToggleGroup

### allowSwitchOff

```
public bool allowSwitchOff { get; set; }
```

Gets/Sets whether or not to allow the situation where no [Toggle](#) is turned on.

If this property is [true](#), pressing a toggle that is currently on will turn it off, and none of the toggles will be turned on. If this property is [false](#), pressing a toggle that is currently ON will not change the state. Even if this property is [false](#), if none of the toggles are turned on immediately after loading or instantiating the scene, it will not force any of them to be turned on immediately. The state will not be corrected until [EnsureValidState\(\)](#) is called in [Start\(\)](#) or [OnEnable\(\)](#).

## Public methods of ToggleGroup

### ActiveToggles

```
public IEnumerable<Toggle> ActiveToggles();
```

Returns all [Toggles](#) that are currently turned on.

However, this does not include [Toggles](#) where the [GameObject](#) is inactive or the [Toggle](#) component is invalid. Note that calling this method will cause a *GC Alloc* to occur.

### AnyTogglesOn

```
public bool AnyTogglesOn();
```

Returns whether there is a [Toggle](#) that is turned on.

### EnsureValidState

```
public void EnsureValidState();
```

We will make sure that this [ToggleGroup](#) is in the proper state. The following is the process of setting it to the appropriate state.

5. If [allowSwitchOff](#) is [false](#), but none of the [Toggles](#) are turned on, then turn on the first [Toggle](#).
6. If more than one [Toggle](#) is ON, turn off the second and subsequent [Toggles](#) that are ON.

This method is called from [Start\(\)](#) or [OnEnable\(\)](#). Conversely, it may be in an inappropriate state until then. Note that calling this method will cause a *GC Alloc* to occur.

## GetFirstActiveToggle

---

```
public Toggle GetFirstActiveToggle();
```

Search for [Toggles](#) that are ON and return the first one found.

Note that calling this method will cause a *GC Alloc* to occur.

## NotifyToggleOn

---

```
public void NotifyToggleOn(Toggle toggle, bool sendCallback = true);
```

Notifies the [ToggleGroup](#) that the [Toggle](#) given in the argument has been turned on.

First, a check is made to see if the [Toggle](#) belongs to this [ToggleGroup](#), and if not, an [ArgumentException](#) exception is thrown. If [sendCallback](#) is [true](#), then [onValueChanged](#) will be called for all other [Toggles](#).

## RegisterToggle

---

```
public void RegisterToggle(Toggle toggle);
```

Register a [toggle](#) to this [ToggleGroup](#).

Immediately after registration, it is necessary to call [NotifyToggleOn\(\)](#) or [EnsureValidState\(\)](#) because the [ToggleGroup](#) may not be in the proper state.

## UnregisterToggle

---

```
public void UnregisterToggle(Toggle toggle);
```

Remove the [toggle](#) from this [ToggleGroup](#).

Immediately after deletion, it is necessary to call `NotifyToggleOn()` or `EnsureValidState()` because the `ToggleGroup` may not be in the proper state.

### SetAllTogglesOff

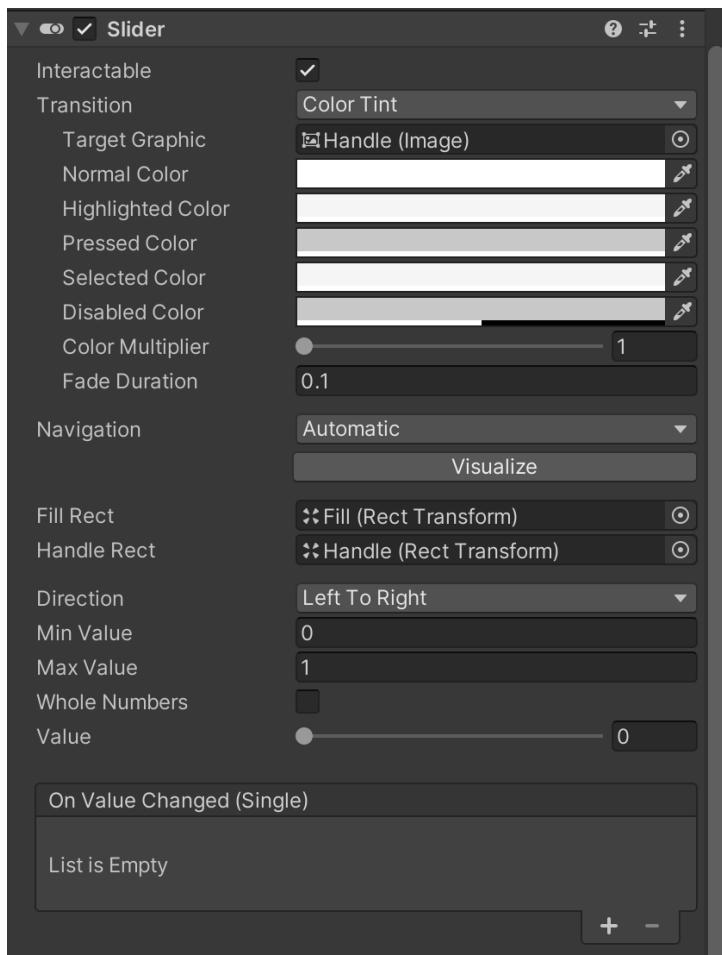
---

```
public void SetAllTogglesOff(bool sendCallback = true);
```

Turn off all `Toggles`.

If `sendCallback` is `true`, then `onValueChanged` for each `Toggle` will be called.

## Slider component



```
[AddComponentMenu("UI/Slider", 33)]
[ExecuteAlways]
[RequireComponent(typeof(RectTransform))]
public class Slider : Selectable, IMoveHandler, IPointerDownHandler, IPointerUpHandler, IPo
terEnterHandler, IPointerExitHandler, ISelectHandler, IDeselectHandler, IDragHandler, IInitializ
ePotentialDragHandler, IEventSystemHandler, ICanvasElement
```

**Slider** is a component to implement a slider that can move between the minimum and maximum values.

The [Slider](#) component inherits from the [Selectable](#) component, which controls the fill and handles. Slider controls the fill and handles, which cover the area from the minimum to the current location. The [RectTransform](#) of the fill and handle is modified by the [Slider](#). When the value of the [Slider](#) is changed, the callback registered in [onValueChanged](#) will be called.

## Properties of Slider

### direction

```
public Slider.Direction direction { get; set; }
```

Gets/Sets the direction of the slider.

Direction is defined as a [Slider.Direction](#) enumeration type.

```
/// <summary>
/// Settings indicating one of the four directions
/// </summary>
public enum Direction
{
    /// <summary>
    /// Left to Right
    /// </summary>
    LeftToRight,

    /// <summary>
    /// Right to Left
    /// </summary>
    RightToLeft,

    /// <summary>
    /// Bottom to Top
    /// </summary>
    BottomToTop,

    /// <summary>
    /// Top to Bottom
    /// </summary>
    TopToBottom,
}
```

For example, in the case of [LeftToRight](#) (left to right), the left is the minimum value and the right is the maximum value.

## fillRect

---

```
public RectTransform fillRect { get; set; }
```

Gets/Sets the [RectTransform](#) of the fill that covers the area from the minimum value to the current location.

Depending on the value of the slider, the [anchorMax](#) or [anchorMin](#) of the [RectTransform](#) changes, which in turn changes the appearance of the fill. If the [Direction](#) is [LeftToRight](#) or [BottomToTop](#), the [anchorMax](#) will change, and if the [Direction](#) is [RightToLeft](#) or [TopToBottom](#), the [anchorMin](#) will change.

## handleRect

---

```
public RectTransform handleRect { get; set; }
```

Gets/Sets the [RectTransform](#) of the handle.

The handle is a UI element in the display, and the dragging decision is made by the [Slider](#) itself.

## maxValue

---

```
public float maxValue { get; set; }
```

Gets/Sets the maximum value of the slider.

The default value is 1.

## minValue

---

```
public float minValue { get; set; }
```

Gets/Sets the maximum value of the slider.

The default value is [0](#).

#### normalizedValue

```
public float normalizedValue { get; set; }
```

Gets/Sets the value of the slider normalized to the range [0](#) to [1](#).

Note that it is possible to set as well as get. [Mathf.InverseLerp\(\)](#) is used to get, and [Mathf.Lerp\(\)](#) is used to set.

#### onValueChanged

```
public Slider.SliderEvent onValueChanged { get; set; }
```

Gets/Sets the callback to be called when the slider is changed.

[Slider.SliderEvent](#) is a [UnityEvent<float>](#).

```
public class SliderEvent : UnityEvent<float> {}
```

The following is a sample that uses [onValueChanged](#) to output a log to the console when the slider is changed.

```
using UnityEngine;
using UnityEngine.UI;

[RequireComponent(typeof(Slider))]
public class SliderCallbackSample : MonoBehaviour
{
    private void Start()
    {
        var slider = GetComponent<Slider>();

        slider.onValueChanged.AddListener((value) =>
    }
```

```
        Debug.Log("Slider changed" + value);
    });
}
```

value

```
public virtual float value { get; set; }
```

Gets/Sets the value of the current slider.

When retrieving the value, if `wholeNumbers` is `false`, the value is returned as is, but if `wholeNumbers` is `true`, `Mathf.Round()` will return the rounded integer value.

When the slider is moved by the UI operation, the `OnMove()` callback is called, and the `value` is changed therein.

wholeNumbers

```
public bool wholeNumbers { get; set; }
```

Sets whether or not to limit the slider value to integers only.

The default value is `false`.

## Public methods of Slider

### FindSelectableOnDown

```
public override Selectable FindSelectableOnDown();
```

Locate and return the underlying [Selectable](#) object for navigation.

This method is an override of [FindSelectableOnDown](#) defined in the [Selectable](#) component.

If [navigation](#) is [Navigation.Mode.Automatic](#) and [direction](#) is vertical ([BottomToTop](#) or [TopToBottom](#)) (since we want to move the slider up and down for up/down key operation), [null](#) will be returned. Otherwise, it returns the result of [Selectable.FindSelectableOnDown\(\)](#).

### FindSelectableOnLeft

```
public override Selectable FindSelectableOnLeft();
```

Locate and return the [Selectable](#) object to the left for navigation.

This method is an override of [FindSelectableOnLeft](#), which is defined in the [Selectable](#) component.

If [navigation](#) is [Navigation.Mode.Automatic](#) and [direction](#) is horizontal ([LeftToRight](#) or [RightToLeft](#)) (because we want the slider to move left and right when the left and right keys are pressed), [null](#) will be returned. Otherwise, it returns the result of [Selectable.FindSelectableOnLeft\(\)](#).

### FindSelectableOnRight

```
public override Selectable FindSelectableOnRight();
```

Locate and return the [Selectable](#) object to the right for navigation.

This method is an override of [FindSelectableOnRight](#), which is defined in the [Selectable](#) component.

If navigation is `Navigation.Mode.Automatic` and direction is horizontal (`LeftToRight` or `RightToLeft`) (because we want the slider to move left and right for left and right key operations), `null` will be returned. Otherwise, it returns the result of `Selectable.FindSelectableOnRight()`.

### FindSelectableOnUp

```
public override Selectable FindSelectableOnUp();
```

Locate and return the `Selectable` object above for navigation.

This method is an override of `FindSelectableOnDown` defined in the `Selectable` component.

If navigation is `Navigation.Mode.Automatic` and direction is vertical (`BottomToTop` or `TopToBottom`) (since we want to move the slider up and down for up/down key operation), `null` will be returned. Otherwise, it returns the result of `Selectable.FindSelectableOnDown()`.

### OnDrag

```
public virtual void OnDrag(PointerEventData eventData);
```

Called by `StandAloneInputModule` when dragged.

This method is an implementation of the `IDragHandler` interface. It calculates and sets the value of the slider based on the current mouse or finger position received from `eventData`.

### OnInitializePotentialDrag

```
public virtual void OnInitializePotentialDrag(PointerEventData eventData);
```

Called by `StandAloneInputModule` just before the drag is detected and starts.

This method is an implementation of the `InitializePotentialDragHandler` interface.

Here, `useDragThreshold` in `eventData` is set to `false`. This means that even if the distance between the previous position of the mouse or finger and the current position is less than a certain value, the

dragging is judged to have started. If `useDragThreshold` is set to `true`, a movement of `10` pixels or less will not be judged as a drag start.

The threshold of `10` pixels to start dragging can be changed in `EventSystem.current.pixelDragThreshold`.

```
EventSystem.current.pixelDragThreshold = 20;
```

Note, however, that this setting will affect the entire current `EventSystem`.

## OnMove

---

```
public override void OnMove(AxisEventData eventData);
```

Called by `StandAloneInputModule` and other modules when moving up, down, left, or right by key input.

This method is an override of `OnMove` defined in the `Selectable` component (which implements the `IMoveHandler` interface).

It receives the movement direction from `eventData` and changes the slider value if it matches the direction of `direction`.

## OnPointerDown

---

```
public override void OnPointerDown(PointerEventData eventData);
```

Called by `StandAloneInputModule` and others when a finger is touched or a mouse is pressed on a touch panel.

This method is defined in the `Selectable` component (which implements the `IPointerDownHandler` interface).

`Selectable.OnPointerDown()` is called first. `OnPointerDown()` is called. Then, if the pressed position is inside the handle, it sets the starting point of the drag. If the pressed position is outside the handle, the slider is immediately moved to the pressed position.

## Rebuild

---

```
public virtual void Rebuild(CanvasUpdate executing);
```

Only when the Editor is running, call `onValueChanged` during the `PreLayout` stage.

## SetDirection

---

```
public void SetDirection(Slider. Direction direction, bool includeRectLayouts);
```

Set the `direction` to indicate the direction of the slider.

If `includeRectLayouts` is `true`, the size and alignment of `RectTransform` will be changed when changing from horizontal to vertical (or vice versa), and the child elements will be changed as well. If `includeRectLayouts` is `false`, no special layout adjustments will be made.

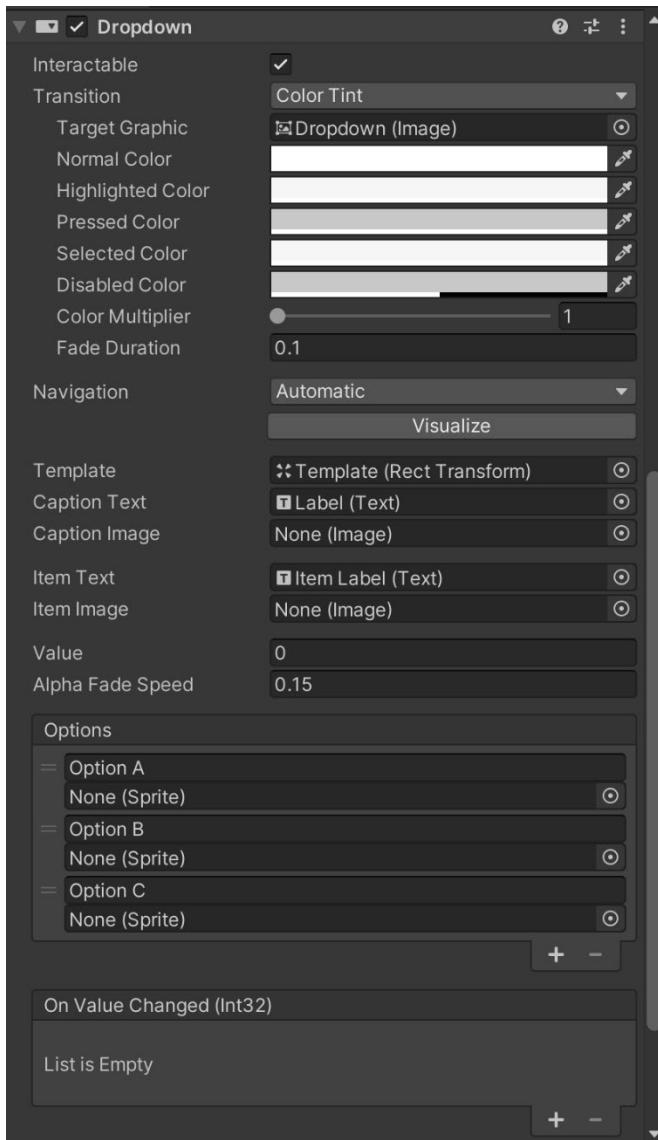
## SetValueWithoutNotify

---

```
public virtual void SetValueWithoutNotify(float input);
```

Change the value of the slider without calling `onValueChanged`.

## Dropdown Components



```
AddComponentMenu("UI/Dropdown", 35)]  
[RequireComponent(typeof(RectTransform))]  
public class Dropdown : Selectable, IMoveHandler, IPointerDownHandler, IPointerUpHandler, IPointerEnterHandler, IPointerExitHandler, ISelectHandler, IDeselectHandler, IPointerClickHandler, ISubmitHandler, ICancelHandler, IEventSystemHandler
```

[Dropdown](#) is a component for implementing a drop-down menu that displays a list of choices when clicked.

When you create a drop-down menu using *UI -> Dropdown* in the Editor, a [GameObject](#) with [Dropdown](#) and [Image](#) attached is generated, and underneath it, a [Label](#) for the display text, [Arrow](#) for the down arrow image, and a Template for the UI element when the menu is opened are generated. [Template](#) is generated.

Although the [template](#) has [ScrollRect](#) and [Scrollbar](#) as children, there is no need to scroll the menu list if the number of items is small. Each item's [GameObject](#) must have a [Toggle](#) attached to it.

When the drop-down menu is opened, an object named *Dropdown List* is instantiated based on the [Template](#), and each item is instantiated as its child (grandchild or great-grandchild). Each item has a component called [DropdownItem](#) attached to it.

A [Canvas](#) component (and a [CanvasGroup](#) component) is attached to the *Dropdown List* object, making it a sub-Canvas. For this sub-Canvas, [overrideSorting](#) is set to [true](#), and [sortingOrder](#) is set to [30000](#). At the same time, a Canvas called *Blocker* is created, and UI elements below the drop-down menu are prevented from being touched. The [sortingOrder](#) of this *Blocker* is set to [29999](#), which is the value of the *Dropdown List* Canvas [sortingOrder](#) - 1.

The value [30000](#) of [sortingOrder](#) in the [Canvas](#) of *Dropdown List* is hard-coded, but it can be customized by overriding [CreateBlocker\(\)](#) and [CreateDropdownList\(\)](#) in components that inherit from [Dropdown](#). Override [CreateBlocker\(\)](#) and [CreateDropdownList\(\)](#) to customize it.

```
using UnityEngine;
using UnityEngine.UI;

#if UNITY_EDITOR
using UnityEditor;
using UnityEditor.UI;
#endif

// Allow customSortingOrder to be changed from Inspector
public class DropdownCustomSortingOrder : Dropdown
{
    public int customSortingOrder = 10000;

    protected override GameObject CreateDropdownList(GameObject template)
    {
        var dropdownlistGO = base.CreateDropdownList(template);
```

```

dropdownlistGO.GetComponent<Canvas>().sortingOrder = customSortingOrder;

return dropdownlistGO;
}

protected override GameObject CreateBlocker(Canvas rootCanvas)
{
    var blockerGO = base.CreateBlocker(rootCanvas);

    blockerGO.GetComponent<Canvas>().sortingOrder = customSortingOrder - 1;

    return blockerGO;
}
}

#if UNITY_EDITOR
[CustomEditor(typeof(DropdownCustomSortingOrder), true)]
[CanEditMultipleObjects]
public class DropdownCustomSortingOrderEditor : DropdownEditor
{
    private SerializedProperty customSortingOrder;

    protected override void OnEnable()
    {
        base.OnEnable();
        customSortingOrder = serializedObject.FindProperty("customSortingOrder");
    }

    public override void OnInspectorGUI()
    {
        base.OnInspectorGUI();
        EditorGUILayout.Space();

        serializedObject.Update();
        EditorGUILayout.PropertyField(customSortingOrder);
        serializedObject.ApplyModifiedProperties();
    }
}
#endif

```

## Properties of Dropdown

### alphaFadeSpeed

```
public float alphaFadeSpeed { get; set; }
```

Gets/Sets the time it takes for the drop-down list to fade when it is shown/hidden.

The default value is `0.15` seconds. If you want a crisp movement, you may want to set it to `0` seconds.

### captionImage

```
public Image captionImage { get; set; }
```

Gets/Sets the `Image` to display the background image or icon image corresponding to the currently selected item.

The default value is `null`.

For each item, the `Sprite` specified in the `options` is set to this `Image`, which is not used very often because of its peculiar behavior, such as the `sprite` of the `captionImage` being rewritten when run in Play mode, and not returning to its original value even after Play is stopped.

### captionText

```
public Text captionText { get; set; }
```

Gets/Sets the `Text` to display the string corresponding to the currently selected item.

The string corresponding to each item is specified in `options`. Actually, this property can be `null` without any problem.

## itemImage

---

```
public Image itemImage { get; set; }
```

This function gets and sets the [Image](#) to display the background image or icon image corresponding to each item when the drop-down list is expanded. The default value is [null](#).

## itemText

---

```
public Text itemText { get; set; }
```

Gets/Sets the [Text](#) to display the string corresponding to each item when the drop-down list is expanded.

## onValueChanged

---

```
public DropdownDropdownEvent onValueChanged { get; set; }
```

Gets/Sets the callback to be called when the selected item of the drop-down menu is changed.

[DropdownDropdownEvent](#) is a [UnityEvent<int>](#).

```
public class DropdownEvent : UnityEvent<int> {}
```

The following is a sample that uses [onValueChanged](#) to output a log to the console when the selected item in the drop-down menu is changed.

```
using UnityEngine;
using UnityEngine.UI;

[RequireComponent(typeof(Dropdown))]
public class DropdownCallbackSample : MonoBehaviour
{
    private void Start()
```

```
{  
    var dropdown = GetComponent<Dropdown>();  
  
    dropdown.onValueChanged.AddListener((index) =>  
    {  
        Debug.LogFormat("{0}th item selected", index);  
    });  
}
```

options

```
public List<Dropdown.OptionData> options { get; set; }
```

Gets/Sets the list of strings and images for each item in the drop-down menu.

Dropdown.OptionData is defined as follows.

```
public class OptionData  
{  
    [SerializeField]  
    private string m_Text;  
    [SerializeField]  
    private Sprite m_Image;  
  
    /// <summary>  
    /// String for this option  
    /// </summary>  
    public string text { get { return m_Text; } set { m_Text = value; } }  
  
    /// <summary>  
    /// Sprite for this option  
    /// </summary>  
    public Sprite image { get { return m_Image; } set { m_Image = value; } }  
    ...  
}
```

If you want to add/remove an item from a script, the code would look like this.

```

using UnityEngine;
using UnityEngine.UI;

[RequireComponent(typeof(Dropdown))]
public class DropdownOptionsCustom : MonoBehaviour
{
    // Assume that the images for each item have been set from the Inspector
    public Sprite[] sprites = new Sprite[3];

    void Start()
    {
        var dropdown = GetComponent<Dropdown>();

        // delete the existing item
        dropdown.ClearOptions();

        // Add the item
        dropdown.options.Add(new Dropdown.OptionData
        {
            text = "Option1",
            image = sprites[0],
        });
        dropdown.options.Add(new Dropdown.OptionData
        {
            text = "Option2",
            image = sprites[1],
        });
        dropdown.options.Add(new Dropdown.OptionData
        {
            text = "Option3",
            image = sprites[2],
        });

        // Notify that the item has been changed
        dropdown.RefreshShownValue();
    }
}

```

## template

---

```
public RectTransform template { get; set; }
```

Gets/Specifies the RectTransform of the GameObject from which the drop-down menu list instance will be created.

There is an object below the GameObject that has a Toogle attached to it, and a DropdownItem is attached to that object, which is instantiated and becomes the runtime item.

The ScrollRect is attached below the default template, but it is not required. Also, the Mask component is used for masking, so you may want to replace it with Rectmask2D as appropriate.

## value

---

```
public int value { get; set; }
```

Gets/Sets the index of the currently selected drop-down menu item.

When the index is changed via this method, the callback function set in onValueChanged will be called. If you do not want to call the callback function, you need to set the value via SetValueWithoutNotify().

## Public methods of Dropdown

### AddOptions

```
public void AddOptions(List<string> options);
public void AddOptions(List<Sprite> options);
public void AddOptions(List<Dropdown.OptionData> options);
```

Add each item to the drop-down menu.

There are three types of overloading: passing only strings, passing only images, and passing both.

```
using UnityEngine;
using UnityEngine.UI;

[RequireComponent(typeof(Dropdown))]
public class DropdownOptionsCustomAdd : MonoBehaviour
{
    // Assume that the images for each item have been set from the Inspector
    public Sprite[] sprites = new Sprite[2];

    void Start()
    {
        var dropdown = GetComponent<Dropdown>();

        // delete the existing item
        dropdown.ClearOptions();

        // Add a string-only item
        Dropdown.AddOptions(new List< string >())
        {
            "string 1",
        });

        // Add an item for images only
        Dropdown.AddOptions(new List<Sprite>())
        {
            sprites[0],
        };
    }
}
```

```
});  
  
// add an item for string and image  
dropdown.AddOptions(new List<Dropdown.OptionData>())  
{  
    new Dropdown.OptionData("string 2", sprites[1])  
};  
}  
}
```

## ClearOptions

---

```
public void ClearOptions();
```

Delete all items from the drop-down menu list.

After deletion, **value** will be reset to **0**.

## Hide

---

```
public void Hide();
```

Close the drop-down menu list that is open.

When the drop-down menu list is closed, the *Dropdown List* object will be destroyed.

## OnCancel(BaseEventData)

---

```
public virtual void OnCancel(BaseEventData eventData);
```

It is supposed to be called by **StandAloneInputModule** when the selection is cancelled by touching another part of the drop-down menu list or pressing the Esc key...but unfortunately, this method is not actually called. But unfortunately, this method is not actually called.

The actual process of closing the drop-down menu list is done by calling `OnCancle()` of `DropdownItem` or `Hide()` via `onClick()` of the `Button` component attached to the *Blocker*.

Since `Hide()` is not a virtual method, you may want to override `DestroyDropdownList()`, which is a protected method called from `Hide()`, in order to have a callback when the drop-down menu list is closed.

```
using UnityEngine;
using UnityEngine.UI;

public class DropdowncustomOnDestroyDropdownList : Dropdown
{
    protected override void DestroyDropdownList(GameObject dropdownList)
    {
        DestroyDropdownList(dropdownList);

        Debug.Log("Dropdown menu list closed");
    }
}
```

However, since `DestroyDropdownList()` is also called when an item is selected, it needs to cooperate with the `onValueChanged` callback called immediately before it to determine whether it is selected or cancelled.

### OnPointerClick

```
public virtual void OnPointerClick(PointerEventData eventData);
```

Called by `StandAloneInputModule` and others when a button is left or right clicked.

This method is a method that implements the `IPointerClickHandler` interface. Here, only `Show()` is called.

### OnSubmit

```
public virtual void OnSubmit(BaseEventData eventData);
```

A callback called by [StandAloneInputModule](#) when the Enter key on the keyboard is pressed.

This method is an implementation of the [ISubmitHandler](#) interface. Here, only [Show\(\)](#) is called.

### RefreshShownValue()

```
public void RefreshShownValue();
```

This method is called when the [options](#), [captionImage](#), or [captionText](#) is changed.

Normally, you do not need to be aware of this method, since it is called automatically when you make changes via each property or via [ClearOptions\(\)](#) or [AddOptions\(\)](#). However, if you change them in some other way (e.g. from outside via Inspector), you need to call this method explicitly.

### SetValueWithoutNotify

```
public void SetValueWithoutNotify(int input);
```

Change the index of the selected item without calling [onValueChanged](#).

### Show()

```
public void Show();
```

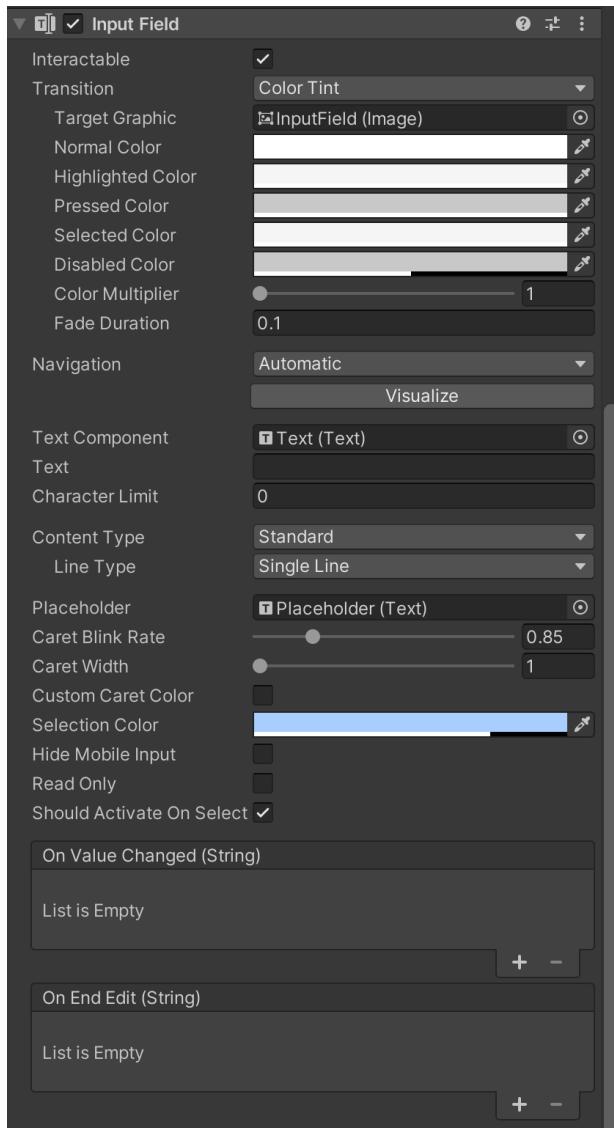
Display the drop-down menu list.

There is a lot of processing to be done in this method, but the following is a summary of what is done.

- Configure the template appropriately.
  - Find a child object that has a [Toggle](#) component and make it the object for the list item.
  - Attach the [DropdownItem](#) component to the object for the list item.
  - Attach a [Canvas](#) component to the template and make it a sub-Canvas.
    - Set the [sortingOrder](#) of the sub-Canvas to [30000](#).
    - Set the same Raycaster as the parent Canvas.

- Set the [CanvasGroup](#).
- Deactivate the template.
- Instantiate a *Dropdown List* object based on the template.
- Only the [Toggle](#) of the *DropdownItem* component of each item instantiated as a child of the *Dropdown List* object with the index equal to [value](#) shall be selected.
- Determines the direction in which the drop-down menu list will be opened. For example, for a drop-down menu list that opens vertically, if the object is located at the top of the Canvas, the list will open at the bottom; conversely, if the object is located at the bottom of the Canvas, the list will open at the top.
- Start the fade-in animation at the time defined by [alphaFadeSpeed](#).
- Create a *Blocker* object.

## InputField component



```
[AddComponentMenu("UI/Input Field", 31)]
```

```
public class InputField : Selectable, IMoveHandler, IPointerDownHandler, IPointerUpHandler, IPointerEnterHandler, IPointerExitHandler, ISelectHandler, IDeselectHandler, IUpdateSelectedHandler, IBeginDragHandler, IDragHandler, IEndDragHandler, IPointerClickHandler, ISubmitHandler, IEventSystemHandler, ICanvasElement, ILayoutElement
```

[InputField](#) is a component that makes text editable.

Unlike other UI components, [InputField](#) itself is used in combination with the UI component to be displayed, not the element to be displayed.

An [InputField](#) component can be attached to an existing [Text](#), and once the [InputField](#) component is attached, it can set itself to the [textComponent](#) property of the [InputField](#).

The [text](#) property of [Text](#) will be changed when the user inputs to the [InputField](#).

Rich text is intentionally not supported. In fact, if you enter rich text markup in an [InputField](#), the markup will be applied immediately, but you will not be able to edit or delete the markup afterwards. So, it is not supported.

To get the text of an [InputField](#), you should access the [text](#) property of the [InputField](#) itself, not the [text](#) property of the [Text](#) component you are displaying. If you want to get the text of an [InputField](#), you should access the [text](#) property of the [InputField](#) itself, not the [text](#) property of the [Text](#) component that you are displaying, because the [Text](#) component's [text](#) may be truncated or "\*" may be used for passwords.

## caret

As a child of [InputField](#), an object called caret is created. The caret is an object that blinks to indicate the position of the character to be input. In the case of smartphones, if the cursor position is controlled by the input keyboard, the caret is not displayed.

The caret is created in the [CanvasUpdate.LatePreRender](#) stage of the rebuild. A new [GameObject](#) is created and the [CanvasRenderer](#) and [LayoutElement](#) components are attached to it. The name of the caret object will be the name of the [InputField GameObject](#) plus "Input Caret".

The [CanvasRenderer](#)'s material is set to the default material ([Graphic.defaultGraphicMaterial](#)), and the texture is set to [Texture2D.whiteTexture](#).

[IgnoreLayout](#) for [LayoutElement](#) is set to [true](#). This allows you to freely change the position of the caret, even if the caret is under some [LayoutGroup](#).

The [RectTransform](#) of the caret copies [localPosition](#), [localRotation](#), [localScale](#), [anchorMin](#), [anchorMax](#), [anchoredPosition](#), [sizeDelta](#), and [pivot](#) from [textComponent](#). The position of the caret and [textComponent](#) will be the same.

## InputField and Japanese input

In MacOS X, the default `InputField` does not support Japanese input, so you need to set the `lineType` property to `InputField.LineType.MultiLineNewline`.

Also, in WebGL, `InputField` cannot be used to input Japanese under any configuration. This can be solved by using the `WebGLNativeInputField` component instead of the `InputField`.

<https://github.com/unity3d-jp/WebGLNativeInputField/>

## Properties of InputField

### asteriskChar

```
public char asteriskChar { get; set; }
```

Gets/Sets the asterisk character used to display the password.

The default value is '\*'.

### caretBlinkRate

```
public float caretBlinkRate { get; set; }
```

Gets/Sets the blinking rate of the caret.

The number of cycles to blink per second is to be specified. The default value is `0.85f`, with a minimum value of `0` and a maximum value of `4`.

### caretColor

```
public Color caretColor { get; set; }
```

Gets/Sets the color of the caret.

The default color is `Color(50f / 255f, 50f / 255f, 50f / 255f, 1f)`. The default value of `customCaretColor` property is `false`, so don't forget to change it to `true`. If `customCaretColor` property is `false`, the color of the caret will be `textComponent.color`.

The color specified here is the `UIVertex` color.

## caretPosition

---

```
public int caretPosition { get; set; }
```

Gets/Sets the current caret position.

This position is also the last position of the selected range. Note that setting this property while a range is selected will cancel the range selection.

The caret position is updated at the rebuild [CanvasUpdate.LatePreRender](#) or at a later time called [LateUpdate](#).

## caretWidth

---

```
public int caretWidth { get; set; }
```

Set/Get the width of the caret in pixels. The default value is [1](#). The minimum value is [1](#), and the maximum value is [5](#).

## characterLimit

---

```
public int characterLimit { get; set; }
```

Gets/Sets the maximum number of characters that can be input to the [InputField](#). The default value is [0](#), but if it is set to [0](#), the maximum value becomes infinite.

## characterValidation

---

```
public InputField.CharacterValidation characterValidation { get; set; }
```

Gets/Sets the type of validation to be performed when a character is input.

[InputField.CharacterValidation](#) is defined as follows.

```

/// <summary>
/// Types of characters that can be added to the string.
/// </summary>
/// <remarks>
/// Character validation is not done for the entire string. Variation is done only for the character you are trying to add.
/// </remarks>
public enum CharacterValidation
{
    /// <summary>
    /// No variation. Any input is valid.
    /// </summary>
    None,

    /// <summary>
    /// Allow (positive and negative) integers.
    /// <remarks>
    /// Numbers from 0 to 9 and - (dash/minus sign) are allowed. Dashes are only allowed on the first character.
    /// <remarks>
    /// </summary>
    Integer,

    /// <summary>
    /// Decimal numbers are allowed.
    /// </summary>
    /// <remarks>
    /// Numbers from 0 to 9, plus . (dot) and - (dash/minus sign) are allowed. Dashes are only allowed in the first character. Only one dot is allowed in a string.
    /// <remarks>
    Decimal,

    /// <summary>
    /// The letters A through Z, the letters a through z, and the digits 0 through 9 are allowed.
    /// </summary>
    Alphanumeric,
}

/// <summary>
/// Only names are allowed, forcing capitulation (initial capitalization)
/// </summary>
/// <remarks>

```

```

    /// Letters, spaces, and ' (apostrophes) are allowed. Letters after a space are automatically converted to uppercase. Characters that are not after a space are automatically converted to lowercase. The character after the apostrophe may be upper or lower case. Only one apostrophe is allowed in a string. Only one apostrophe is allowed in a string; multiple spaces in a string are not allowed.
    ///
    Isletter is considered a character if it is a Unicode character as implemented in .
    /// <remarks>
    Name,
    /// <summary>
    /// Allowed if the character is used in the email address.
    /// Allows the characters that are allowed in an email address.
    /// </summary>
    /// <remarks>
    /// A to Z, a to z, 0 to 9, @, . (dot), !, #, $, %, &, ', *, +, -, /, =, ?, ^, _, ` , {, |, }, ~ are allowed.
    ///
    /// Only one @ is allowed in a string. Multiple dots in a row are not allowed. Note that this is not an exact validation according to RFC.
    /// </remarks>
    EmailAddress
}

```

The validation process is done in the protected method called `Validate()`. If you want to do your own validation, you need to pass your own validation method to the `onValidateInput` property.

## contentType

---

```
public InputField.ContentType contentType { get; set; }
```

When `contentType` is set, `InputType`, `CharacterValidation`, `LineType`, and `TouchScreenKeyboardType` will be set appropriately according to the value of `contentType`. `InputField.ContentType` is defined as follows

```

    /// <summary>
    /// Setting ContentType serves as a shortcut for setting the combination of InputType, CharacterValidation, LineType, and TouchScreenKeyboardType.
    /// </summary>
    /// <remarks>

```

```

/// ContentType affects character validation, the type of keyboard used (on platforms with on-screen keyboards), whether or not to accept multiple lines, whether or not text is autocorrected (on platforms with input autocorrect), whether or not characters are displayed as they are, and whether or not to accept multiple lines. The following settings are available: the type of keyboard used (on platforms with on-screen keyboards), whether to accept multiple lines, whether text is autocorrected (on platforms with input autocorrect), and whether passwords should not display text as is.
/// </remarks>
public enum ContentType
{
    /// <summary>
    /// All input is allowed.
    /// </summary>
    Standard,

    /// <summary>
    /// All input is allowed, and autocorrect is performed if supported by the platform.
    /// </summary>
    Autocorrected,

    /// <summary>
    /// Allow (positive and negative) integers.
    /// </summary>
    IntegerNumber,

    /// <summary>
    /// Allows (positive and negative) decimal numbers.
    /// </summary>
    DecimalNumber,

    /// <summary>
    /// Allows characters from A to Z, characters from a to z, and numbers from 0 to 9.    /// </summary>
    Alphanumeric,
}

/// <summary>
/// InputField is used to input names, and forces capitalization of the first letter of each word. However, it is possible to avoid capitalization by automatically deleting the capitalized letters.
/// </summary>
Name,

/// <summary>
/// The input is used for typing in an email address.

```

```
/// The input is used for typing in an email address.  
/// </summary>  
EmailAddress,  
  
/// <summary>  
/// All input is allowed, but typed characters are hidden by an asterisk.  
/// </summary>  
Password,  
  
/// <summary>  
/// Only integers are allowed, but the characters entered are hidden by an asterisk.  
/// </summary>  
Pin,  
  
/// <summary>  
/// Custom settings defined by the user.  
/// </summary>  
Custom  
}
```

The settings for `lineType`, `inputType`, `keyboardType`, and `characterValidation` for each `ContentType` are as follows.

- If the `contentType` is `Standard`
  - The `lineType` is not set specifically.
  - The `inputType` is `Standard`
  - The `keyboardType` is `Default`.
  - `characterValidation` is `None`
- If the `contentType` is `Autocorrected`
  - The `lineType` is not set specifically.
  - `inputType` is `AutoCorrect`
  - The `keyboardType` is `Default`.
  - `characterValidation` is `None`
- If the `contentType` is `IntegerNumber`
  - `lineType` is `SingleLine`
  - The `inputType` is `Standard`
  - `keyboardType` is `NumberPad`
  - `characterValidation` is an `Integer`
- If the `contentType` is `DecimalNumber`
  - `lineType` is `SingleLine`
  - The `inputType` is `Standard`
  - `keyboardType` is `NumbersAndPunctuation`
  - `characterValidation` is a `Decimal`
- If the `contentType` is `Alphanumeric`
  - `lineType` is `SingleLine`
  - The `inputType` is `Standard`
  - `keyboardType` is `ASCIICapable`
  - `characterValidation` is an `Alphanumeric`

- If the `contentType` is `Name`
  - `lineType` is `SingleLine`
  - The `inputType` is `Standard`
  - `keyboardType` is `NamePhonePad`
  - `characterValidation` is the name
- If the `contentType` is `EmailAddress`
  - `lineType` is `SingleLine`
  - The `inputType` is `Standard`
  - `keyboardType` is `EmailAddress`
  - `characterValidation` is an email address.
- If the `contentType` is `Password`
  - `lineType` is `SingleLine`
  - `inputType` is `Password`
  - The `keyboardType` is `Default`.
  - `characterValidation` is `None`
- If the `contentType` is `Pin`
  - `lineType` is `SingleLine`
  - `inputType` is `Password`
  - `keyboardType` is `NumberPad`
  - `characterValidation` is an `Integer`

## customCaretColor

---

```
public bool customCaretColor { get; set; }
```

Gets/Sets whether to use its own caret color or `textComponent.color`.

If this property is `true`, the color of the caret will be the color of `caretColor`; if it is `false`, the color of the caret will be `textComponent.color`.

## flexibleWidth

---

```
public virtual float flexibleWidth { get; }
```

Get the flexible width used for Auto Layout.

It always returns `-1`, so the width of flexible will be ignored. The details are explained in *Chapter 9 Auto Layout*.

## flexibleHeight

---

```
public virtual float flexibleHeight { get; }
```

Get the flexible height used for Auto Layout.

Since it always returns `-1`, the height of the flexible will be ignored. The details are explained in *Chapter 9 Auto Layout*.

## inputType

---

```
public InputField.InputType inputType { get; set; }
```

Gets/Sets the type of input.

The definition of `InputType` is as follows.

```
/// <summary>
/// Type of data input from keyboard.
/// </summary>
public enum InputType
{
    /// <summary>
    /// Standard keyboard
    /// </summary>
    Standard,

    /// <summary>
    /// With AutoCorrect
    /// </summary>
    AutoCorrect,

    /// <summary>
    /// Password
    /// </summary>
    Password,
}
```

## isFocused

```
public bool isFocused { get; }
```

Get whether or not this `InputField` is focused, i.e., whether or not it is ready for input.

For example, the process of changing the color depending on whether it is focused or not would be as follows.

```
var inputField = GetComponent<InputField>();
var image = inputField.GetComponent<Image>();
if (inputField.isFocused)
{
    image.color = Color.green;
}
else
{
```

```
    image.color = Color.white;  
}
```

## keyboardType

```
public TouchScreenKeyboardType keyboardType { get; set; }
```

Get/Sets the keyboard type for mobile devices such as iOS and Android.

The definition of `TouchScreenKeyboardType` is as follows. The definition of `TouchScreenKeyboardType` is as follows (in fact, it is almost identical to the definition of the `UIKeyboardType` enumeration type in Objective-C/Swift for iOS).

```
/// <summary>  
/// <para> Enumeration of supported touchscreen keyboard types</para>  
/// </summary>  
public enum TouchScreenKeyboardType  
{  
    /// <summary>  
    /// <para> The default keyboard layout on the targeted platform</para>  
    /// </summary>  
    Default,  
  
    /// <summary>  
    /// <para> Keyboard with standard ASCII keys</para>  
    /// </summary>  
    ASCIICapable,  
  
    /// <summary>  
    /// <para> Number and Punctuation Keyboard</para>  
    /// </summary>  
    NumbersAndPunctuation,  
  
    /// <summary>  
    /// <para> Keyboard for URL input. " .", "/" and ".com" are available. </para>  
    /// </summary>  
    URL,
```

```
/// <summary>
/// <para> numeric keyboard</para>
/// </summary>
NumberPad,
```

```
/// <summary>
/// <para> A keyboard with a layout suitable for entering phone numbers. Numbers and "*" and "#" are available. </para>
/// </summary>
PhonePad,
```

```
/// <summary>
/// <para> Alphabetic and numeric keyboard</para>
/// </summary>
NamePhonePad,
```

```
/// <summary>
/// <para> Keyboard suitable for entering email addresses." @" and "." and the spacebar are available. </para>
/// </summary>
EmailAddress,
```

```
/// <summary>
/// <para> Keyboard for Nintendo Network (Deprecated). </para>
/// </summary>
[Obsolete ("Wii U is no longer supported as of Unity 2018.1.")]
NintendoNetworkAccount,
```

```
/// <summary>
/// <para> Keyboard for social media such as Twitter. "@" is available; "#" is available on iOS and tvOS. </para>
/// </summary>
Social,
```

```
/// <summary>
/// <para> Keyboard with "." next to the spacebar. Good for searching. </para>
/// </summary>
Search,
```

```
/// <summary>
/// <para> Number and decimal point keyboard</para>
```

```
/// </summary>
DecimalPad,
```

```
/// <summary>
/// <para> Numeric-only keyboard, suitable for entering PIN numbers and one-time password
/// s. iOS 12 or later can autofill one-time authentication. </para>
/// </summary>
OneTimeCode
}
```

## layoutPriority

```
public virtual int layoutPriority { get; }
```

Get the layout priority of this component.

Always returns 1.

This property is an implementation of the [ILayoutElement](#) interface.

## lineType

```
public InputField.LineType lineType { get; set; }
```

Gets/Sets the row type of this [InputField](#).

The definition of [LineType](#) is as follows.

```
/// <summary>
/// Specify the behavior of the InputField row
/// </summary>
public enum LineType
{
    /// <summary>
    /// Only one line is allowed. Horizontal scrolling is allowed. horizontalOverflow of textCompo
    /// nent property is set to HorizontalWrapMode.Overflow. press Return key to submit.
```

```
/// </summary>
SingleLine,
```

```
/// <summary>
/// Multiple lines can be entered. Vertical scrolling is possible. horizontalOverflow of textCom
ponent is set to HorizontalWrapMode.Wrap. press Return key to submit.
```

```
/// </summary>
MultiLineSubmit,
```

```
/// <summary>
/// Multiple lines can be entered. Vertical scrolling is possible. horizontalOverflow of textCom
ponent is set to HorizontalWrapMode.Wrap. line feed is inserted when Return key is pressed.
```

```
/// </summary>
MultiLineNewline
}
```

The default value is [SingleLine](#).

Even if you only need one line, you will need to choose something other than [SingleLine](#) in the following situations.

- If you want to use Japanese input on MacOS X. In this case, you need to set the [lineType](#) to [MultiLineNewline](#).
- Wrap for BestFit or for your own layout calculations. In this case, you may need to change the [lineType](#) to [MultiLineSubmit](#).

## minHeight

```
public virtual float minHeight { get; }
```

Get the minimum height to be used for Auto Layout.

Always returns [0](#). The details are explained in *Chapter 9 Auto Layout*.

## minWidth

```
public virtual float minWidth { get; }
```

Get the minimum width to be used for Auto Layout.

Always returns 0. The details are explained in *Chapter 9 Auto Layout*.

### multiLine

```
public bool multiLine { get; }
```

Gets whether or not the [InputField](#) supports multiple lines.

If the [lineType](#) is [LineType.MultiLineNewline](#) or [LineType.MultiLineSubmit](#), otherwise (i.e., if [lineType](#) is [SingleLine](#)) return false.

### onEndEdit

```
public InputField.SubmitEvent onEndEdit { get; set; }
```

Gets/Sets the callback that will be called when the editing is completed.

[InputField.SubmitEvent](#) is a [UnityEvent<string>](#).

```
public class SubmitEvent : UnityEvent<string> {}
```

The timing when this callback is called is as follows.

- The Enter key was pressed.
- The focus is now outside the [InputField](#).
- The mobile on-screen keyboard has been hidden.
- [OnDeselect\(\)](#) of the [IDeselectHandler](#) interface was called by [StandAloneInputModule](#) and others when this object was deselected.
- [OnDisable\(\)](#) was called when a [GameObject](#) became inactive, etc.

In particular, note that [onEndEdit](#) may be called at the same time that [OnDisable\(\)](#) is called.

## onValidateInput

```
public InputField.onValidateInput onValidateInput { get; set; }
```

Gets/Sets a callback to validate the inputted characters.

The definition of `InputField.OnValidateInput` is as follows.

```
public delegate char OnValidateInput(string text, int charIndex, char addedChar);
```

The sample code for setting `onValidateInput` is shown below.

```
using UnityEngine;
using UnityEngine.UI;

public class MyInputFieldValidate : MonoBehaviour
{
    void Start()
    {
        var inputField = GetComponent<InputField>();
        inputField.onValidateInput = (text, charIndex, addedChar) =>
        {
            // If it's not a number, convert it to an empty string
            if (!char.IsDigit(addedChar))
            {
                addedChar = '\0';
            }

            // return the final character to be added
            return addedChar;
        };
    }
}
```

If the `onValidateInput` property is not set, then validation will be performed in `InputField.Validate()` based on the `characterValidation` setting.

## onValueChanged

---

```
public InputField.OnChangeEvent onValueChanged { get; set; }
```

Gets/Sets the callback that will be called when the text is changed.

`InputField.OnChangeEvent` is a `UnityEvent<string>`.

```
public class OnChangeEvent : UnityEvent<string> {}
```

This callback is called after the validation is finished. It is also called after deletion or Ctrl+X cut, not just after adding characters. This callback can be called from various sources, but note that it is called from its own `LateUpdate()`.

## placeholder

---

```
public Graphic placeholder { get; set; }
```

Gets/Sets the `Graphic` to be displayed when the `InputField` text is empty.

In the case of an `InputField` created from the Editor, it refers to a `Text` component with the text "Enter text...". You can set any component that inherits from `Graphic`, for example, you can specify an `Image` component.

## preferredWidth

---

```
public virtual float preferredWidth { get; }
```

Get the width that you want to set if there is enough space during Auto Layout.

If `textComponent` is not `null`, it will return a width wide enough to display the `text` (using the `cachedTextGeneratorForLayout` of `textComponent` to calculate the width needed to display the `text` using `TextGenerator.GetPreferredWidth()` to calculate and return the width needed to display the `text`).

If `textComponent` is `null`, 0 is returned.

The details of how `preferredWidth` is used are explained in *Auto Layout*.

### preferredHeight

```
public virtual float preferredHeight { get; }
```

Get the height to be set when there is enough space during Auto Layout.

If `textComponent` is not `null`, it returns a height that is sufficient to display the `text` (it uses the `cachedTextGeneratorForLayout` of `textComponent` to calculate and return the height needed to display the `text` using `TextGenerator.GetPreferredHeight()` to calculate and return the height needed to display the `text`).

If `textComponent` is `null`, 0 is returned.

The details of how `preferredHeight` is used are explained in *Auto Layout*.

### readOnly

```
public bool readOnly { get; set; }
```

Gets/Sets whether the text is read-only or not.

The default value is `false`.

Read-only means that it cannot be edited from the keyboard, but it can be edited from the script via the `text` property.

Even if it is read-only, it is possible to move the caret by keyboard operation.

### selectionAnchorPosition

```
public int selectionAnchorPosition { get; set; }
```

Gets/Sets the position of the first character of the selected range.

In fact, the value you get is the same as the value you get from the `caretPosition` property. Also, when you try to set a value, if the range is selected, nothing is done, but if it is not, the `caretPosition` is set to that value.

### selectionColor

---

```
public Color selectionColor { get; set; }
```

Gets/Sets the highlight color when selecting a range of text with the keyboard.

The highlight rectangle of the range selection is drawn by a mesh generated using [VertexHelper](#).

### selectionFocusPosition

---

```
public int selectionFocusPosition { get; set; }
```

Gets/Sets the position of the last character of a range selection.

If the range is not selected, the values of `selectionFocusPosition` and `selectionAnchorPosition` will match. When you try to set a value, if the range is selected, nothing will be done, but if it is not, the `caretPosition` will be set to that value.

### shouldHideMobileInput

---

```
public bool shouldHideMobileInput { get; set; }
```

Gets/Sets whether the keyboard input area of the mobile is hidden or not.

For iOS / tvOS / Android, the default value is `false`, which can be changed via this property. For all other platforms, it always returns `true`.

If this property is `false`, the caret will not be displayed (because the cursor is in the input area of the mobile device).

## text

---

```
public string text { get; set; }
```

Gets/Sets the current text value of the InputField.

When setting, "\0" will be replaced with `string.Empty`. `SingleLine`, "\n" and "\t" will be replaced with "".

Then, validation is performed by calling `onValidateInput` or `InputField.Validate()`. It also checks that the length of the text does not exceed the `characterLimit`.

Finally, if the text is changed, the `onValueChanged` callback is called.

## textComponent

---

```
public Text textComponent { get; set; }
```

Gets/Sets the `Text` component to be used for displaying on the screen.

When a `textComponent` is set, the `horizontalOverflow` property of the `textComponent` is set according to the value of `lineType`.

## touchScreenKeyboard

---

```
public TouchScreenKeyboard touchScreenKeyboard { get; }
```

Get a `TouchScreenKeyboard` object that represents a touchscreen keyboard used on a mobile platform.

The `TouchScreenKeyboard` object is created by calling `TouchScreenKeyboard.Open()` at the timing of `LateUpdate()` when it needs to be created.

wasCanceled

---

```
public bool wasCanceled { get; }
```

Get whether or not [DeactivateInputField\(\)](#) is called as a result of the keyboard being hidden or the Esc key being pressed to cancel the input and return to the original text.

The default value is [false](#).

This value, once set to [true](#), will remain [false](#) until the input is activated again.

## Public methods of InputField

### ActivateInputField

```
public void ActivateInputField();
```

Activate the input field so that events can be processed.

Activating the input field means that the following process will be performed

- Call `EventSystem.current.SetSelectedGameObject()` to register itself as the currently selected object.
- Call `TouchScreenKeyboard.Open()` to generate the touchscreen keyboard used by the mobile platform.
- Enable focus.
- Display the caret
- Set the `text` of `InputField` to the `text` of `textComponent`.

However, it will actually become active at the timing of the first `LateUpdate()` after this method is called.

### CalculateLayoutInputHorizontal

```
public virtual void CalculateLayoutInputHorizontal();
```

This method is an implementation of `CalculateLayoutInputHorizontal()` of the `ILayoutElement` interface, but it is empty.

### CalculateLayoutInputVertical

```
public virtual void CalculateLayoutInputVertical();
```

This method is an implementation of `CalculateLayoutInputVertical()` of the `ILayoutElement` interface, but it is empty.

## DeactivateInputField

---

```
public void DeactivateInputField();
```

Deactivates the [InputField](#) and stops the event processing.

If it is not cancelled, Submit will be performed at this time, and the [onEndEdit](#) callback will be called.

## ForceLabelUpdate

---

```
public void ForceLabelUpdate();
```

Forcibly and immediately update the [text](#) of [textComponent](#).

When this method is called, the position of the caret is also recalculated.

## GraphicUpdateComplete

---

```
public virtual void GraphicUpdateComplete();
```

Called by the [CanvasUpdateRegistry](#) (via the [ICanvasElement](#) interface) when the [Graphic](#) rebuild is complete, but empty.

## LayoutComplete

---

```
public virtual void LayoutComplete();
```

Called by the [CanvasUpdateRegistry](#) (via the [ICanvasElement](#) interface) when the layout rebuild is complete, but empty.

## MoveTextEnd

---

```
public void MoveTextEnd(bool shift);
```

Move the position of the caret to the end of the text.

If `shift` is `true`, the range from the current caret position to the end of the text will be selected as the range.

## MoveTextStart

---

```
public void MoveTextStart(bool shift);
```

Move the position of the caret to the beginning of the text.

If `shift` is `true`, the range from the current caret position to the beginning of the text will be selected as the range.

## OnBeginDrag

---

```
public virtual void OnBeginDrag(PointerEventData eventData);
```

This method is called at the timing when the dragging actually starts (i.e., the touch position moves).

This method is an implementation of the `IBeginDragHandler` interface.

Dragging in the input field is implemented for this purpose, as it allows you to make a range selection.

## OnDeselect

---

```
public override void OnDeselect(BaseEventData eventData);
```

Called by [StandAloneInputModule](#) and others when this object is deselected.

This method is an implementation of the [IDeselectHandler](#) interface.

When this method is called, [DeactivateInputField\(\)](#) will be called and the input field will be deactivated.

### OnDrag

---

```
public virtual void OnDrag(PointerEventData eventData);
```

Called by [StandAloneInputModule](#) when dragged.

This method is an implementation of the [IDragHandler](#) interface.

Dragging in the input field is implemented for this purpose, as it allows you to make a range selection.

### OnEndDrag

---

```
public virtual void OnEndDrag(PointerEventData eventData);
```

Called by [StandAloneInputModule](#) and others when the drag is completed.

This method is an implementation of the [IEndDragHandler](#) interface.

Dragging in the input field is implemented for this purpose, as it allows you to make a range selection.

### OnPointerClick

---

```
public virtual void OnPointerClick(PointerEventData eventData);
```

Called by [StandAloneInputModule](#) and others when a button is left or right clicked.

This method is an implementation of the [IPointerClickHandler](#) interface.

It is implemented to make the input field active when a left click is made.

### OnPointerDown

---

```
public override void OnPointerDown(PointerEventData eventData);
```

Called by [StandAloneInputModule](#) and others when a finger is touched or a mouse is pressed on a touch panel.

This method is defined in the [Selectable](#) component (which implements the [IPointerDownHandler](#) interface).

The process to be followed here is as follows.

- Call [ActivateInputField\(\)](#) to activate the input field.
- Call [EventSystem.current.SetSelectedGameObject\(\)](#) to register itself as the currently selected object.
- Set the caret position to the touched position.

### OnSubmit

---

```
public virtual void OnSubmit(BaseEventData eventData);
```

It is called by [StandAloneInputModule](#) and other modules when the Enter key is pressed on the keyboard.

This method is an implementation of the [ISubmitHandler](#) interface.

If the input field is inactive, it will be activated.

### OnUpdateSelected

---

```
public virtual void OnUpdateSelected(BaseEventData eventData);
```

Called every frame from [StandAloneInputModule](#) etc. when this object is selected.

This method is an implementation of the [IUpdateSelectedHandler](#) interface.

It is implemented to deactivate an input field or select all text in response to a keystroke.

### ProcessEvent

---

```
public void ProcessEvent(Event e);
```

Mainly handles key input events.

Backspace / Delete / Home / End / Insert / Enter(Return) / Esc / Ctrl + A / Ctrl + C / Ctrl + V / Ctrl + X / Up/Down key events are handled.

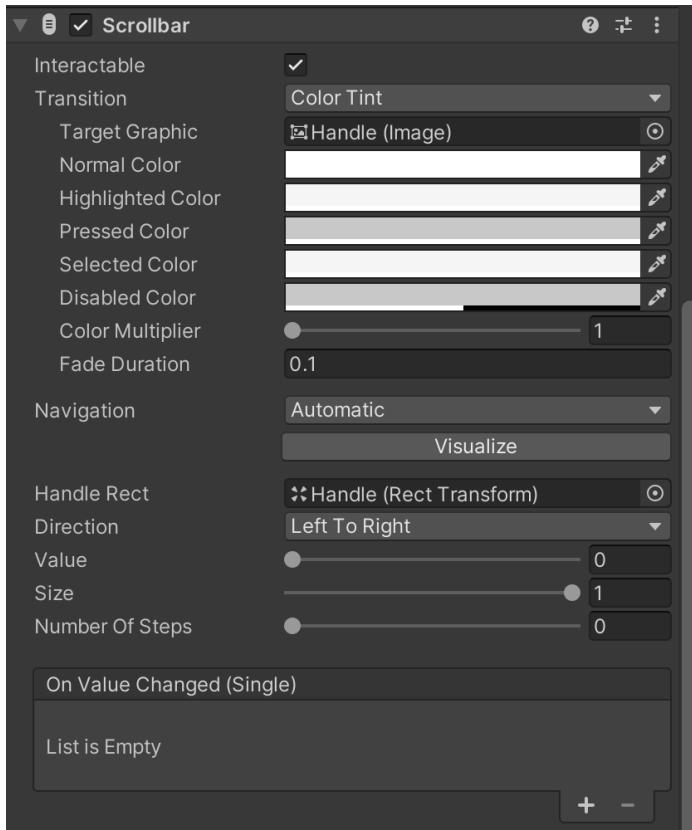
### Rebuild

---

```
public virtual void Rebuild(CanvasUpdate update);
```

Update the geometry (caret and highlight rectangles) generated by the [InputField](#) at the timing of the [LatePreRender](#) stage of the Canvas rebuild.

## Scrollbar component



```
[AddComponentMenu("UI/Scrollbar", 34)]
[ExecuteAlways]
[RequireComponent(typeof(RectTransform))]
public class Scrollbar : Selectable, IMoveHandler, IPointerDownHandler, IPointerUpHandler, IP
ointerEnterHandler, IPointerExitHandler, ISelectHandler, IDeselectHandler, IBEGINDragHandler,
IDragHandler, IInitializePotentialDragHandler, IEventSystemHandler, ICanvasElement
```

Scrollbar is a component to implement a scrollbar that can move between 0 and 1.

The **Scrollbar** component inherits from the **Selectable** component, which is a component copied from **Slider**, with the following differences

- The **Slider** is used to set the value, and the **Scrollbar** is used to move the display range.
- The **Scrollbar** has a fixed value range of 0 to 1.

- [Scrollbar](#) does not have a fill that covers from the minimum value to the current location.
- The size of the [Scrollbar](#) handle is variable.
- The [Scrollbar](#) implements the [IBeginDragHandler](#) interface. This makes it possible to know when a drag is actually initiated (i.e., the touch position is moved) via the [OnBeginDrag\(\)](#) callback. On the other hand, the [OnInitializePotentialDrag\(\)](#) callback of the [InitializePotentialDragHandler](#) interface is called just before the drag is detected and starts.

## Properties of Scrollbar

direction

```
public Scrollbar.Direction direction { get; set; }
```

Gets/Sets the scroll bar direction.

Direction is defined as a [Scrollbar.Direction](#) enumeration type with four directions: left to right, right to left, bottom to top, and top to bottom.

```
/// <summary>
/// Settings indicating one of the four directions
/// </summary>
public enum Direction
{
    /// <summary>
    /// Left to Right
    /// </summary>
    LeftToRight,

    /// <summary>
    /// Right to Left
    /// </summary>
    RightToLeft,

    /// <summary>
    /// Bottom to Top
    /// </summary>
    BottomToTop,

    /// <summary>
    /// Top to Bottom
    /// </summary>
    TopToBottom,
}
```

For example, in the case of [LeftToRight](#) (left to right), left is [0](#) and right is [1](#).

## handleRect

---

```
public RectTransform handleRect { get; set; }
```

Gets/Sets the [RectTransform](#) of the handle.

The handle is a UI element in the display, and the dragging decision is made by the [Scrollbar](#) itself.

## numberOfSteps

---

```
public int numberOfSteps { get; set; }
```

Gets/Sets the number of divisions for displaying the position of the handle in a discrete (i.e. stepwise) manner.

If the value of this property is less than or equal to [1](#), the handle will move smoothly; if it is greater than [1](#), the position of the handle will be equally divided between [0](#) and [1](#). The default value is [0](#).

Even if this value is set to a value greater than [1](#), the internal position will change smoothly with a float. Note that this property only indicates the position in the appearance field.

## onValueChanged

---

```
public Scrollbar.ScrollEvent onValueChanged { get; set; }
```

Gets/Sets the callback that will be called when the scroll bar is changed.

[Scrollbar](#).[SliderEvent](#) is a [UnityEvent<float>](#).

```
public class Scrollbar : UnityEvent<float> { }
```

The following is a sample code that uses [onValueChanged](#) to output a log to the console when the slider is changed.

```
[RequireComponent(typeof(Scrollbar))]
public class ScrollbarCallbackSample : MonoBehaviour
{
    private void Start()
    {
        var scrollbar = GetComponent<Scrollbar>();

        scrollbar.onValueChanged.AddListener((value) =>
        {
            Debug.Log("Scrollbar changed" + value);
        });
    }
}
```

## size

```
public float size { get; set; }
```

Gets/Sets the size of the handle in the range of [0](#) to [1](#).

This value is a percentage of the total slider area. The default value is [0.2](#).

The [ScrollRect](#) component, described below, changes the [size](#) of this scroll bar according to the size of the content to be scrolled. If the content size is large, the [size](#) of the scroll bar is adjusted to be small, and if the content size is small, the [size of](#) the scroll bar is adjusted to be large.

## value

```
public float value { get; set; }
```

Gets/Sets the value of the current scroll bar.

If [numberOfSteps](#) is greater than [1](#), then a value equally divided by [numberOfSteps](#) between [0](#) and [1](#) is returned.

```
public float value
{
    get
    {
        float val = m_Value;
        if (m_NumberOfSteps > 1)
            val = Mathf.Round(val * (m_NumberOfSteps - 1)) / (m_NumberOfSteps - 1);
        return val;
    }
    set
    {
        Set(value);
    }
}
```

## Public methods of Scrollbar

### OnMove

```
public override void OnMove(AxisEventData eventData);
```

Called by `StandAloneInputModule` and other modules when moving up, down, left, or right by key input.

This method is an override of `OnMove` defined in the `Selectable` component (which implements the `IMoveHandler` interface).

It receives the movement direction from `eventData` and changes the value of the scroll bar if it matches the direction of `direction`.

### FindSelectableOnDown

```
public override Selectable FindSelectableOnDown();
```

Locate and return the underlying `Selectable` object for navigation.

This method is an override of `FindSelectableOnDown` defined in the `Selectable` component.

If `navigation` is `Navigation.Mode.Automatic` and `direction` is vertical (`BottomToTop` or `TopToBottom`), `null` is returned (because we want to move the scroll bar up and down for up/down key operation). Otherwise, it returns the result of `Selectable.FindSelectableOnDown()`.

### FindSelectableOnLeft

```
public override Selectable FindSelectableOnLeft();
```

Locate and return the `Selectable` object to the left for navigation.

This method is an override of `FindSelectableOnLeft`, which is defined in the `Selectable` component.

If navigation is `Navigation.Mode.Automatic` and `direction` is horizontal (`LeftToRight` or `RightToLeft`) (because we want the scrollbar to move left and right when the left and right keys are pressed), `null` will be returned. Otherwise, it returns the result of `Selectable.FindSelectableOnLeft()`.

### FindSelectableOnRight

---

```
public override Selectable FindSelectableOnRight();
```

Locate and return the `Selectable` object to the right for navigation.

This method is an override of `FindSelectableOnRight`, which is defined in the `Selectable` component.

If navigation is `Navigation.Mode.Automatic` and `direction` is horizontal (`LeftToRight` or `RightToLeft`) (because we want the scrollbar to move left and right when the left and right keys are pressed), `null` will be returned. Otherwise, it returns the result of `Selectable.FindSelectableOnRight()`.

### FindSelectableOnUp

---

```
public override Selectable FindSelectableOnUp();
```

Locate and return the `Selectable` object above for navigation.

This method is an override of `FindSelectableOnDown` defined in the `Selectable` component.

If navigation is `Navigation.Mode.Automatic` and `direction` is vertical (`BottomToTop` or `TopToBottom`), `null` is returned (because we want to move the scroll bar up and down for up/down key operation). Otherwise, it returns the result of `Selectable.FindSelectableOnDown()`.

### SetDirection

---

```
public void SetDirection(Scrollbar.Direction direction, bool includeRectLayouts);
```

Set the direction of the scroll bar to the specified value.

If `includeRectLayouts` is `true`, the size and alignment of `RectTransform` will be changed when changing from horizontal to vertical (or vice versa), and the child elements will be changed as well. If `includeRectLayouts` is `false`, no special layout adjustments will be made.

### OnInitializePotentialDrag

```
public virtual void OnInitializePotentialDrag(PointerEventData eventData);
```

Called by `StandAloneInputModule` just before the drag is detected and starts.

This is a method that implements the `InitializePotentialDragHandler` interface.

Here, `useDragThreshold` in `eventData` is set to `false`. This means that even if the distance between the previous position of the mouse or finger and the current position is less than a certain value, it will be judged that dragging has started. If `useDragThreshold` is set to `true`, a movement of `10` pixels or less will not be judged as a drag start.

The threshold of `10` pixels to start dragging can be changed in `EventSystem.current.pixelDragThreshold`.

```
EventSystem.current.pixelDragThreshold = 20;
```

This setting affects the entire current `EventSystem`.

### OnBeginDrag

```
public virtual void OnBeginDrag(PointerEventData eventData);
```

It is called by `StandAloneInputModule` and other modules at the timing when the dragging actually starts (i.e., the touch position moves).

This method is an implementation of the `IBeginDragHandler` interface.

If you just touch the handle, the amount of movement is zero, and this method will not be called. If you want to detect the timing of immediate movement by touching the handle or touching outside the handle, you need to use `OnInitializePotentialDrag()` of the `InitializePotentialDragHandler` interface.

The reason why this is done in [OnBeginDrag\(\)](#) instead of [OnPointerDown\(\)](#), unlike [Slider](#), is that there is a possibility that the [Scrollbar](#) will be changed from the [ScrollRect](#) while the user is holding down the button.

## OnDrag

---

```
public virtual void OnDrag(PointerEventData eventData);
```

Called by [StandAloneInputModule](#) when dragged.

This method is an implementation of the [IDragHandler](#) interface.

Calculate and set the slider value based on the current mouse or finger position received from [eventData](#).

## OnPointerDown

---

```
public override void OnPointerDown(PointerEventData eventData);
```

Called by [StandAloneInputModule](#) and others when a finger is touched or a mouse is pressed on a touch panel.

This method is defined in the [Selectable](#) component (which implements the [IPointerDownHandler](#) interface).

[OnPointerDown\(\)](#) is called first when this method is called. [OnPointerDown\(\)](#) is called. Then, if the pressed position is inside the handle, it is used as the starting point of the drag. If the pressed position is outside the handle, the coroutine is started, and the process of moving the scroll bar to the pressed position immediately continues until the finger leaves or the dragging starts. This is done after waiting for [WaitForEndOfFrame](#) to finish processing the [ScrollRect](#)'s [LateUpdate](#). This is because the [ScrollRect](#) changes the [Scrollbar](#).

## OnPointerUp

---

```
public override void OnPointerUp(PointerEventData eventData);
```

Called by [StandAloneInputModule](#) when the finger is released from the touch screen or the mouse is released.

This method is an implementation of the [IPointerUpHandler](#) interface. It changes the current selection state [currentSelectionState](#) (so that [SelectionState.Pressed](#) is released) and then performs the transition. In addition, it clears the press-and-hold decision initiated by [OnPointerDown\(\)](#).

### Rebuild

---

```
public virtual void Rebuild(CanvasUpdate executing);
```

Only when the Editor is running, [onValueChanged](#) is called during the [PreLayout](#) stage of the Canvas rebuild.

### SetValueWithoutNotify

---

```
public virtual void SetValueWithoutNotify(float input);
```

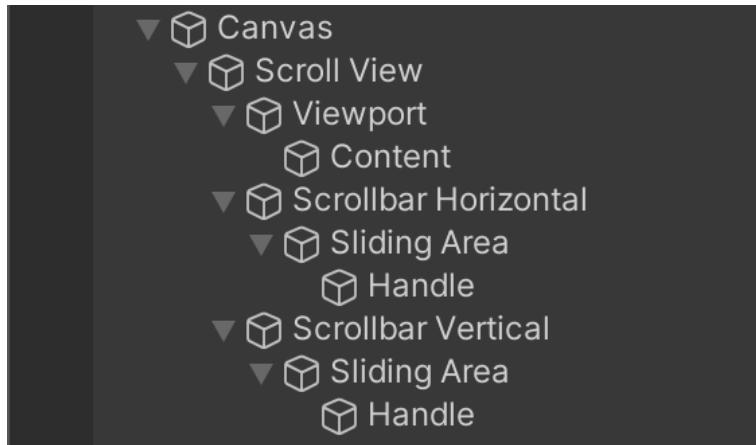
Change the value of the scroll bar without calling [onValueChanged](#).

## Chapter 8 Scroll View

A Scroll View is a scrollable UI element consisting of a [ScrollView](#) and its related components, which can be created most quickly by selecting *UI -> Scroll View* from the Editor.

The structure of the object when created from the Editor using *UI -> Scroll View* is as follows.

```
ScrollView (RectTransform/Image, Anchor:middle/center)
    Viewport (Mask/Image, Anchor:stretch/stretch)
        Content (Anchor:top/stretch)
    Scrollbar Horizontal (Scrollbar/Image, Anchor:bottom/stretch)
        Sliding Area (Anchor:stretch/stretch)
        Handle (Image, Anchor:stretch/stretch)
    Scrollbar Vertical (Scrollbar/Image, Anchor:stretch/right)
        Sliding Area (Anchor:stretch/stretch)
        Handle (Image, Anchor:custom/stretch)
```



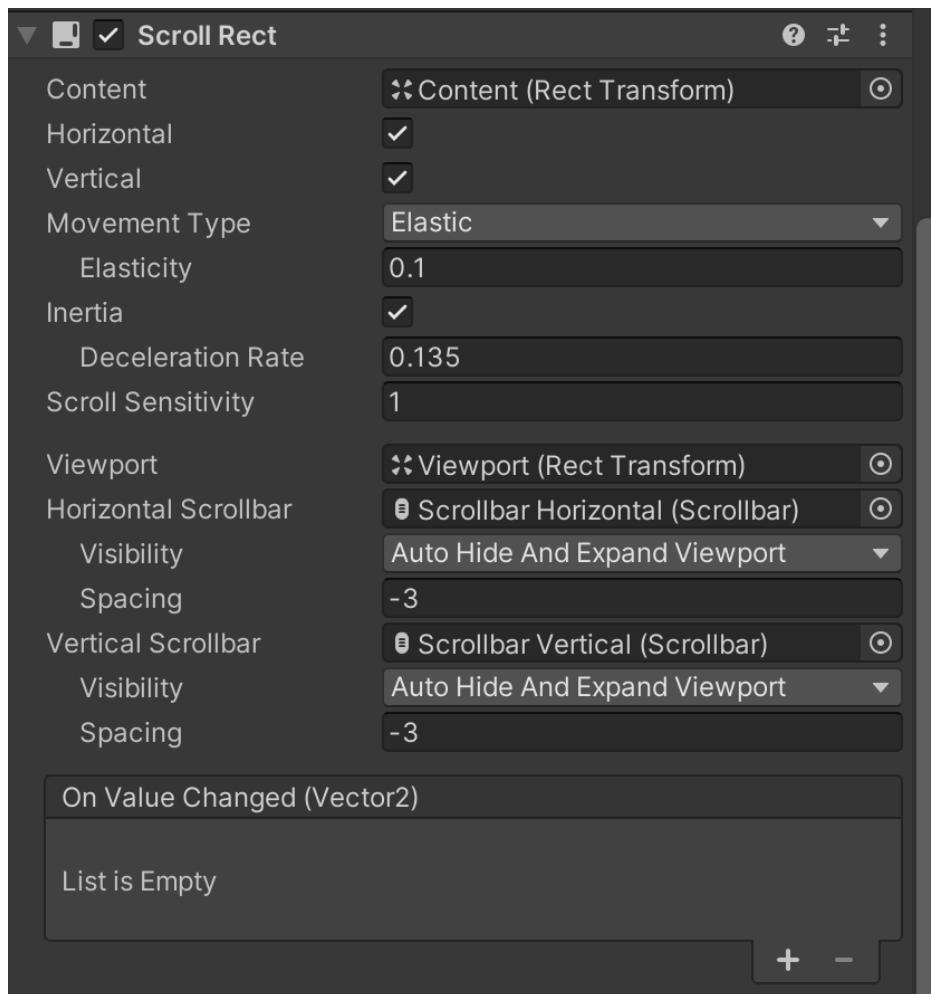
There is a [GameObject](#) with a [RectTransform](#) attached at the top, a *Viewport* (the content display area) and *Content* (the displayed content) below it, and horizontal and vertical [Scrollbars](#). [RectTransform](#) and these objects work together to create the scrolling view.

[RectTransform](#) is implemented under the assumption that the *Viewport*, *Scrollbar Horizontal*, and *Scrollbar Vertical* are directly under the *ScrollView*. Therefore, this hierarchical structure should not be changed unnecessarily.

Scroll View can cause major performance problems, so care should be taken when implementing it.

First, let's look at the [ScrollRect](#) component, which plays a central role in Scroll View.

## ScrollRect component



```
[AddComponentMenu("UI/Scroll Rect", 37)]
[SelectionBase]
[ExecuteAlways]
[DisallowMultipleComponent]
[RequireComponent(typeof(RectTransform))]
public class ScrollRect : UIBehaviour, InitializePotentialDragHandler, IBeginDragHandler, IEndDragHandler, IDragHandler, IScrollViewHandler, IEventSystemHandler, ICanvasElement, ILayoutElement, ILayoutGroup, ILayoutController
```

## Properties of ScrollRect

### content

```
public RectTransform content { get; set; }
```

Gets/Sets the [RectTransform](#) of the scrollable content.

The [GameObject](#) of [content](#) should exist in a hierarchy that is a child of [ScrollRect](#).

If the amount of content you want to display is small, you can create all the content in advance under this [content](#), but if not, you will have to change the [content](#) dynamically.

### viewport

```
public RectTransform viewport { get; set; }
```

Gets/Sets the [RectTransform](#) of the viewport of the content display area.

In the case of the object created by *UI -> Scroll View*, the Viewport with the [Mask](#) component attached is set. If you want to simply mask the object with a rectangle, it may be better to replace it with a [RectMask2D](#) component for better performance. Performance will vary depending on the platform and UI configuration, so be sure to measure and select the right one.

The [viewport](#) must reside directly under the [ScrollRect](#).

### horizontalScrollbar

```
public Scrollbar horizontalScrollbar { get; set; }
```

Sets/gets the [Scrollbar](#) object for horizontal scrolling.

The scroll bar must exist directly under the [ScrollRect](#).

## verticalScrollbar

---

```
public Scrollbar verticalScrollbar { get; set; }
```

Sets/gets the [Scrollbar](#) object for vertical scrolling.

The scroll bar must exist directly under the [ScrollRect](#).

## inertia

---

```
public bool inertia { get; set; }
```

Gets/Sets whether or not to keep the speed for a while after the dragging in the [RectScroll](#) area is finished.

The default value is [true](#). If this property is set to [true](#), the dragging will gradually decelerate after the dragging is finished, and the deceleration rate is defined by [decelerationRate](#). The current movement speed per second can be obtained/set in [velocity](#).

Note that when scrolling by dragging the [Scrollbar](#) instead of dragging within the [RectScroll](#) area, the [velocity](#) will remain at [0](#), and scrolling will stop immediately when the dragging of the [Scrollbar](#) is finished.

This property is common to both horizontal and vertical directions.

## decelerationRate

---

```
public float decelerationRate { get; set; }
```

Gets/Sets the percentage of deceleration if [inertia](#) is [true](#).

The default value is [0.135f](#). If the content is not currently being dragged and is not moving elastically beyond the scroll rectangle ([movementType](#) is not [Elastic](#)), velocity will be multiplied by [Pow\(decelerationRate, Time.unscaledDeltaTime\)](#). Therefore, if the value of this property is [0](#), it will stop immediately, if it is [1](#), it will maintain the velocity, and if it is greater than [1](#), it will

accelerate. If the value is set to a negative value, the content disappears in the direction of the day after tomorrow (the coordinates become `NaN`), so it must not be set to a negative value.

This property is common to both horizontal and vertical directions.

## elasticity

```
public float elasticity { get; set; }
```

Gets/Sets the amount of elasticity to be used when the content moves beyond the scroll rectangle.

The default value is `0.1f`. If the value of this property is less than or equal to `0`, the content will immediately return to the position inside the rectangle when the dragging is finished outside the scroll rectangle. The larger the value, the slower the return. Note that when scrolling with the mouse scroll wheel, the elasticity is tripled.

As with `inertia`, when scrolling by dragging the `Scrollbar` instead of dragging within the `RectScroll` area, `velocity` is kept at `0`, and scrolling stops immediately when the dragging of the `Scrollbar` is finished.

This property is common to both horizontal and vertical directions.

## horizontal

```
public bool horizontal { get; set; }
```

Gets/Sets whether horizontal scrolling is enabled or disabled.

If this property is `true`, horizontal scrolling is possible by dragging in the `ScrollRect` area. Even if this property is `false`, if `horizontalScrollbar` is set to horizontal scrollbar, left and right scrolling will be possible via the scrollbar.

If the `horizontalScrollbarVisibility` is set to `AutoHide` or `AutoHideAndExpandViewport`, and the horizontal width of the content is the same as the `ScrollRect`, the horizontal scrollbar will be invisible, but you can still scroll beyond the scroll rectangle by dragging inside the `ScrollRect`. If the horizontal width of the content is the same as the `ScrollRect`, the horizontal scrollbar will be invisible, but it is still possible to scroll beyond the scroll rectangle by dragging inside the `ScrollRect`.

Therefore, if you do not use horizontal scrolling, you should always explicitly set this property to `false`.

### vertical

---

```
public bool vertical { get; set; }
```

Gets/Sets whether vertical scrolling is enabled or disabled.

If this property is `true`, vertical scrolling is possible by dragging in the `ScrollRect` area. The behavior is the same as `horizontal`.

### horizontalNormalizedPosition

---

```
public float horizontalNormalizedPosition { get; set; }
```

Gets/Sets the numeric value that represents the horizontal scroll position between `0` and `1`.

`0` represents the left edge. The value will be within the range of `0` to `1` even when scrolling beyond the scroll rectangle.

### verticalNormalizedPosition

---

```
public float verticalNormalizedPosition { get; set; }
```

Gets/Sets the numeric value between `0` and `1` for the scroll position in the vertical direction.

`0` represents the bottom edge. The value will be within the range of `0` to `1` even when scrolling beyond the scroll rectangle.

### horizontalScrollbarSpacing

---

```
public float horizontalScrollbarSpacing { get; set; }
```

Gets/Sets the height of the space between the horizontal scroll bar and the viewport.

The horizontal scroll bar's touch detection will be extended inside the viewport by this height.

If a horizontal scroll bar is present, the anchor is automatically adjusted so that the height of the viewport and the length of the vertical scroll bar are reduced by this height. The default value is `0`, but if you create it from `UI -> ScrollView` in Editor, it will be set to `-3` (i.e., `3` pixels longer). It is better to adjust it according to your visual convenience.

Note that this property will only appear in the **Inspector** if `horizontalScrollbarVisibility` is set to `AutoHideAndExpandViewport`, but as long as the horizontal scrollbar exists, it will be enabled regardless of the value of `horizontalScrollbarVisibility`.

### verticalScrollbarSpacing

```
public float verticalScrollbarSpacing { get; set; }
```

Gets/Sets the height of the space between the vertical scroll bar and the viewport.

The behavior is the same as `verticalScrollbarSpacing`.

### horizontalScrollbarVisibility

```
public ScrollRect.ScrollbarVisibility horizontalScrollbarVisibility { get; set; }
```

Gets/Sets the display mode of the horizontal scroll bar.

The display modes of horizontal and vertical scroll bars are defined as the `ScrollRect.ScrollbarVisibility` enumeration type.

```
/// <summary>
/// Enumeration indicating the display mode of the scroll bar
/// </summary>
public enum ScrollbarVisibility
{
    /// <summary>
    /// Always show the scrollbar
```

```

/// </summary>
Permanent,

```

```

/// <summary>
/// Hide the scrollbars if they are not needed (the content fits in the viewport).
/// In this case, do not change the viewport rectangle size.
/// </summary>
AutoHide,

```

```

/// <summary>
/// Hide the scrollbars if they are not needed (the content will fit in the viewport).
/// In this case, the rectangle size of the viewport is increased by the amount of the scrollbar.
/// </summary>
/// <remarks>
/// If this setting is used, scrollbars and viewport rectangles will move in tandem.
/// The RectTransform of the scrollbar and viewport is automatically calculated and cannot be edited manually.
/// </remarks>
AutoHideAndExpandViewport,
}

```

`AutoHideAndExpandViewport` is not recommended in cases where the size of the content changes dynamically, because the viewport size is recalculated when the size of the content increases, causing complicated behavior. In such cases, you should use `Permanent` or `AutoHide`.

If you find scrollbars visually distracting, you can usually set the alpha value of the scrollbars to 0, and then fade the alpha when the scrolling starts.

## verticalScrollbarVisibility

```
public ScrollRect.ScrollbarVisibility verticalScrollbarVisibility { get; set; }
```

Gets/Sets the display mode of the horizontal scroll bar.

The behavior is the same as for `horizontalScrollbarVisibility`.

## movementType

```
public ScrollRect.MovementType movementType { get; set; }
```

Gets/Sets the behavior when the content is moved beyond the scroll rectangle.

The definition of the `ScrollRect.MovementType` enumeration type is as follows

```
public enum MovementType
{
    /// <summary>
    /// No restriction on movement. Content can be moved forever.
    /// </summary>
    Unrestricted,

    /// <summary>
    /// Elastic movement. Content can temporarily move beyond the bounds, but will be shrunk back to its original size.
    /// </summary>
    Elastic,

    /// <summary>
    /// Elastic movement. The content cannot be moved beyond the frame.
    /// </summary>
    Clamped,
}
```

The default value is `Elastic`.

## normalizedPosition

```
public Vector2 normalizedPosition { get; set; }
```

Gets/Sets the scroll position expressed as `Vector2` between `(0, 0)` and `(1, 1)`.

In `Vector2`, `x` represents the horizontal direction and `y` represents the vertical direction, with `(0, 0)` being the lower left and `(1, 1)` being the upper right.

### onValueChanged

---

```
public ScrollRect.ScrollRectEvent onValueChanged { get; set; }
```

Gets/Sets the callback to be called from `LateUpdate()` when the scroll position is changed or the display area or display boundary is changed.

`ScrollRect.ScrollRectEvent` is a `UnityEvent<Vector2>` with `normalizedPosition` passed as an argument.

It should be remembered that the timing of the call is `LateUpdate()`.

### preferredHeight

---

```
public virtual float preferredHeight { get; }
```

Get the preferred height used for Auto Layout.

Always returns `-1`. The details are explained in *Chapter 9 Auto Layout*.

### preferredWidth

---

```
public virtual float preferredWidth { get; }
```

Get the preferred height used for Auto Layout.

Always returns `-1`. The details are explained in *Chapter 9 Auto Layout*.

### scrollSensitivity

---

```
public float scrollSensitivity { get; set; }
```

Gets/Sets the scroll sensitivity of the mouse scroll wheel and trackpad.

The default value is [1](#).

If the value is [0](#), the mouse scroll wheel and trackpad will not work.

velocity

```
public Vector2 velocity { get; set; }
```

Gets/Sets the current speed of the content in horizontal and vertical directions.

[Velocity](#) is recalculated every [LateUpdate\(\)](#), so if you want to keep it moving at a constant speed manually, you need to re-set [velocity](#) repeatedly using [Update\(\)](#) or something similar. If you want to keep moving at a constant speed manually, you need to re-set [velocity](#) repeatedly using [Update\(\)](#).

flexibleHeight

```
public virtual float flexibleHeight { get; }
```

Get the flexible height used for Auto Layout.

It always returns [-1](#), so the height of [flexible](#) is ignored. The details are explained in [Chapter 9 Auto Layout](#).

flexibleWidth

```
public virtual float flexibleWidth { get; }
```

Get the flexible width used for Auto Layout.

It always returns [-1](#), so the width of [flexible](#) is ignored. The details are explained in [Chapter 9 Auto Layout](#).

## layoutPriority

---

```
public virtual int layoutPriority { get; }
```

Get the priority level used for Auto Layout.

Always returns [-1](#). The details are explained in *Auto Layout*.

## minHeight

---

```
public virtual float minHeight { get; }
```

Get the minimum height to be used for Auto Layout.

Always returns [-1](#). The details are explained in *Auto Layout*.

## minWidth

---

```
public virtual float minWidth { get; }
```

Get the minimum width to be used for Auto Layout.

Always returns [0](#). The details are explained in *Auto Layout*.

## Public methods of ScrollRect

### CalculateLayoutInputHorizontal

```
public virtual void CalculateLayoutInputHorizontal();
```

This method is an implementation of [CalculateLayoutInputHorizontal\(\)](#) of the [ILayoutElement](#) interface, but it is empty.

After this method is called, the properties of the horizontal input of the layout should return the latest values. Also, at the time this method is called, the child should always have the latest horizontal input of the layout.

### CalculateLayoutInputVertical

```
public virtual void CalculateLayoutInputVertical();
```

This method is an implementation of [CalculateLayoutInputVertical\(\)](#) of the [ILayoutElement](#) interface, but it is empty.

After this method is called, the property of the vertical input of the layout should return the latest value. Also, at the time this method is called, the child should always have the latest vertical input of the layout.

### GraphicUpdateComplete

```
public virtual void GraphicUpdateComplete();
```

Called by the [CanvasUpdateRegistry](#) (via the [ICanvasElement](#) interface) when the [Graphic](#) rebuild is complete, but empty.

## IsActive

```
public override bool IsActive();
```

Returns whether this object is active or not.

This is an override method of [UIBehaviour](#) defined in [UIBehaviour](#).

```
public abstract class UIBehaviour : MonoBehaviour
{
    ...
    /// <summary>
    /// Returns true if GameObject and Component are active
    /// </summary>
    public virtual bool IsActive()
    {
        return isActiveAndEnabled;
    }
}
```

In the case of [ScrollRect](#), it also checks if the content is set or not.

```
public override bool IsActive()
{
    return base.IsActive() && m_Content != null;
}
```

## LayoutComplete

```
public virtual void LayoutComplete();
```

Called by the [CanvasUpdateRegistry](#) (via the [ICanvasElement](#) interface) when the layout rebuild is complete, but empty.

## OnInitializePotentialDrag

---

```
public virtual void OnInitializePotentialDrag(PointerEventData eventData);
```

Called by [StandAloneInputModule](#) just before the drag is detected and starts.

This method is an implementation of the [InitializePotentialDragHandler](#) interface.

Here, [velocity](#) is set to [Vector2.zero](#).

## OnBeginDrag

---

```
public virtual void OnBeginDrag(PointerEventData eventData);
```

It is called at the timing when the dragging actually starts (i.e., the touch position moves).

This method is an implementation of the [IBeginDragHandler](#) interface.

Turn on the flag that is being dragged, adjust the position and size of the content, and record the starting point of the drag.

## OnDrag

---

```
public virtual void OnDrag(PointerEventData eventData);
```

Called by [StandAloneInputModule](#) when dragged.

This method is an implementation of the [IDragHandler](#) interface.

The difference from the starting point of the drag recorded by [OnBeginDrag](#) is produced, and the position of the content is changed.

## OnEndDrag

---

```
public virtual void OnEndDrag(PointerEventData eventData);
```

Called by [StandAloneInputModule](#) and others when the drag is completed.

This method is an implementation of the [IEndDragHandler](#) interface.

When this method is called, the flag indicating that a drag is in progress is turned off.

## OnScroll

---

```
public virtual void OnScroll(PointerEventData data);
```

Called by [StandAloneInputModule](#) and others when the mouse scroll wheel and trackpad are scrolled.

This method is an implementation of the [IScrollHandler](#) interface.

Change the position of the content by multiplying the amount of scrolling by [scrollSensitivity](#).

## Rebuild

---

```
public virtual void Rebuild(CanvasUpdate executing);
```

It is processed in the [PreLayout](#) and [PostLayout](#) stages of the Canvas rebuild.

In the [PreLayout](#) stage, the following processes are performed.

- Update the cache of [RectTransform](#) for horizontal/vertical scroll bars.
- Update the height and width cache of the horizontal and vertical scroll bars.
- Update the cache of flags for whether or not to expand the viewport when a horizontal/vertical scrollbar is hidden. This is determined not only by the [horizontalScrollbarVisibility](#) and [verticalScrollbarVisibility](#), but also by the proper hierarchy (i.e., whether the scrollbar and viewport are directly under the [ScrollRect](#)). The flags are determined by not only the

`verticalScrollbarVisibility`, but also the proper hierarchy (whether the scrollbar and viewport are directly under the `ScrollRect`).

In the `PostLayout` stage, the following processes are performed.

- Adjust the position and size of the content.
- Adjust the position of the scroll bars.
- Set the flag to cause a `Canvas` rebuild (`Canvas.ForceUpdateCanvases()`) in `LateUpdate()` (if Canvas rebuild has not already occurred).

### SetLayoutHorizontal

```
public virtual void SetLayoutHorizontal();
```

Originally, this method is called to set the horizontal layout during Auto Layout, but in the case of `ScrollRect`, it is used to adjust the position and size of the viewport and content.

This method implements the `ILayoutGroup` interface (which inherits from `ILayoutController`).

### SetLayoutVertical

```
public virtual void SetLayoutVertical();
```

Originally, this method is called to set the vertical layout in Auto Layout, but in the case of `ScrollRect`, it is used to adjust the position and size of the scroll bars.

This method implements the `ILayoutGroup` interface (which inherits from `ILayoutController`).

### StopMovement

```
public virtual void StopMovement();
```

Called to stop scrolling.

When this method is called, `velocity` is set to `Vector2.zero`.

## How to configure content for Scroll View

There are two main approaches to configuring content for Scroll View.

1. Prepare all the contents to be displayed under *Contents* in advance.
2. If necessary, place the UI elements you want to display in the *Contents*.

Challenges exist with both of these approaches.

The first approach may be useful if you only want to display fixed elements. However, as the number of items to be displayed increases, the time to instantiate all of the UI elements will take longer and the time to rebuild the Scroll View will also increase. If only a few elements are needed in the Scroll View, this approach is preferred because of its simplicity.

The second approach may be useful for lists of indefinite number of elements. However, it requires a lot of code to implement. Since instantiating UI elements to be displayed is a heavy process, it inevitably requires an object pool.

## Notes on content updates

In the above case of "placing the position of UI elements to be displayed in *Contents* as needed", it is necessary to update *Contents* dynamically according to scrolling. The following is a summary of the precautions to be taken when updating the contents.

- The `onValueChanged` callback for `ScrollRect` is called late in `LateUpdate()`. At this point, the position and size of the content and the position of the scroll bar are fixed. At this point, the position and size of the content and the position of the scroll bars are fixed, so it is fine to use `onValueChanged` to update the content display. If you want to change the content position, size, or scrollbar position yourself, you should do so before `LateUpdate()`, that is, in `Update()`.
- `OnTransformParentChanged()` is called not only when the `transform` parent is changed, but also when the sibling order in the hierarchy is changed, and `Canvas` rebuild is performed by `SetAllDirty()`. So, basically, do not change the sibling order of elements in the content.
- It is also possible to use a sub-`Canvas` as an element of the Scroll View. However, this is not recommended because it increases the number of draw calls and the load on the `Canvas` rebuild.
- Many components of UI elements (not only `ScrollRect` components) have the `[ExecuteAlways]` attribute, so if you change the object with `Awake()`, `Start()`, or `OnEnable()`, it will be reflected even when the Editor is running. If you change the object with `Awake()`, `Start()` or `OnEnable()`, it will be reflected in the Editor. Therefore, if you want to extend the functionality of `ScrollRect`, it is safer to attach it to the same `GameObject` as a separate component instead of a component that inherits from `ScrollRect`.
- The cost of batching a `Canvas` increases based on the number of `CanvasRenderers`, not the number of `RectTransforms` in the `Canvas`.

## ScrollView Sample

With the above caveats in mind, we can create a helper class like the following `VerticalScrollList`. This `VerticalScrollList` is a component to implement a vertical scroll list, and works by attaching it to the same `GameObject` as the `ScrollRect` component.

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

[RequireComponent(typeof(ScrollRect))]
public class VerticalScrollList : MonoBehaviour
{
    // Data for the UI elements in the content
    public class ContentsElementData
    {
        // Something to determine the type of UI element
        public int type;

        // Y-coordinate within the content (assuming 0 at the top and positive as you go down)
        public float y { set; get; }

        // Height
        public float height { set; get; }
    }

    // Object pool for each type of UI element
    protected class ElementObjectPool
    {
        // List of pooled objects
        private List<GameObject> list = new List<GameObject>();

        // Type of the object managed by this pool
        public int type { get; set; }

        // Get an object from the pool. If the pool is empty, create a new one and return it.
        public GameObject GetObject()
        {
            GameObject go;
```

```

if (list. Count == 0) // if the pool is empty.
{
    // Create a GameObject.
    go = new GameObject(string.Format("Contents Type:{0}", type), typeof(RawImage));

    // Attach the component in a nice way
    var rawImage = go.GetComponent<RawImage>();

    // change the content depending on the type
    switch (type)
    {
        case 0:
            rawImage.color = Color.red;
            break;

        case 1:
            rawImage.color = Color.blue;
            break;

        case 2:
            rawImage.color = Color.green;
            break;
    }
}

else // If it's in the pool.
{
    // get it out of the pool
    go = list[0];
    list.RemoveAt(0);
}

return go;
}

// return the object to the pool
public void ReleaseObject(GameObject go)
{
    list.Add(go);
}
}

// All data of UI elements in the content

```

```

protected List<ContentsElementData> contentsElementDatas = new List<ContentsElementData>();
// Cache RectTransform for each UI element
protected List<RectTransform> elementRectTransforms = new List<RectTransform>();

// Cache ScrollRects attached to the same GameObject
protected ScrollRect scrollRect;

// Object pool per type
protected Dictionary<int, ElementObjectPool> elementPoolDict = new Dictionary<int, ElementObjectPool>();

// Remember the scroll position
public static Dictionary<string, float> initialPositionDict = new Dictionary<string, float>();

protected void Start()
{
    scrollRect = GetComponent<ScrollRect>();

    // Replace Mask with RectMask2D
    // However, mask may be faster on PC or console (not mobile)
    if (GetComponent<RectMask2D>() == null)
    {
        scrollRect.viewport.gameObject.AddComponent<RectMask2D>();
    }

    var mask = scrollRect.viewport.gameObject.GetComponent<Mask>();
    if (mask != null)
    {
        Destroy(mask);
    }

    // Horizontal scrolling is disabled
    scrollRect.horizontal = false;

    // Changing the Handle anchorMin / anchorMax of the vertical scrollbar will cause
    // Graphic rebuild will occur, so hide the vertical scrollbar.
    // If you need scrollbars, it's better to implement them by yourself without anchor.
    scrollRect.verticalScrollbar.gameObject.SetActive(false);
    scrollRect.verticalScrollbar = null;
}

```

```

InitContentData();
InitContentSize();

// Register OnValueChanged (*don't forget to cancel it with OnDestroy())
scrollRect.onValueChanged.AddListener(OnValueChanged);

// Set the Y coordinate of each UI element
float totalY = 0.0f;

for (int i = 0; i < contentsElementDatas. Count; i++)
{
    // The Y coordinates of the UI elements are calculated here.
    contentsElementDatas[i].y = totalY;
    totalY += contentsElementDatas[i].height;

    elementRectTransforms.Add(null);
}

// Perform the initial UI placement
scrollRect.normalizedPosition = new Vector2(0, 1);
OnValueChanged(scrollRect.normalizedPosition);
}

protected void OnDestroy()
{
    // Cancel OnValueChanged
    scrollRect.onValueChanged.RemoveListener(OnValueChanged);
}

// Get the pool corresponding to type
protected ElementObjectPool GetElementPool(int type)
{
    // Create a new pool if it is not in the pool's Dirctionary
    if (!elementPoolDict. ContainsKey(type))
    {
        var pool = new ElementObjectPool();
        pool.type = type;
        elementPoolDict.Add(type, pool);
    }

    return elementPoolDict[type];
}

```

```

// Create the UI element
protected RectTransform CreateElement(int index)
{
    // Get the data that determines the contents of the UI element
    var data = contentsElementDatas[index];

    // Get the object pool
    var elementPool = GetElementPool(data.type);

    // Get the object
    var go = elementPool.GetObject();

    // Object's parent is ScrollRect
    go.transform.SetParent(scrollRect.content, false);

    // Set the position of the object
    var rectTransform = go.transform as RectTransform;
    rectTransform.sizeDelta = new Vector2(scrollRect.content.rect.width - 50, data.height);
    rectTransform.anchorMin = new Vector2(0.5f, 1.0f);
    rectTransform.anchorMax = new Vector2(0.5f, 1.0f);
    rectTransform.pivot = new Vector2(0.5f, 1.0f);

    // Y-coordinate gets smaller as we go down, so we'll reverse the sign
    rectTransform.anchoredPosition = new Vector3(0, -1 * data.y);

    elementRectTransforms[index] = rectTransform;

    return rectTransform;
}

// Return the object to the pool
protected void ReleaseElement(int index)
{
    var rectTransform = elementRectTransforms[index];

    var go = rectTransform.gameObject;

    var data = contentsElementDatas[index];
    var elementPool = GetElementPool(data.type);
    elementPool.ReleaseObject(go);
}

```

```

        elementRectTransforms[index] = null;
    }

    // Create the data that determines the content of the UI element
    protected void InitContentData()
    {
        // Determine the data for the UI elements in the content
        for (int i = 0; i < 20; i++)
        {
            contentsElementDatas.Add(new ContentsElementData
            {
                type = 0,
                height = 20,
            });
            contentsElementDatas.Add(new ContentsElementData
            {
                type = 1,
                height = 30,
            });
            contentsElementDatas.Add(new ContentsElementData
            {
                type = 2,
                height = 40,
            });
        }
    }

    // RectTransform of each UI element is initially set to null.
    // If it is visible in the Viewport, the RectTransform will be set.
    elementRectTransforms = new List<RectTransform>(contentsElementDatas.Count);
    for (int i = 0; i < elementRectTransforms.Count; i++)
    {
        elementRectTransforms.Add(null);
    }
}

// Initialize the RectTransform of the content
protected void InitContentSize()
{
    var contentRectTransform = scrollRect.content.transform as RectTransform;

    // Anchor is top/stretch
    contentRectTransform.anchorMin = new Vector2(0.0f, 1.0f);
}

```

```

contentRectTransform.anchorMax = new Vector2(1.0f, 1.0f);

// Position is top-aligned
contentRectTransform.pivot = new Vector2(0.5f, 1.0f);

// Initial position is top
contentRectTransform.anchoredPosition = Vector2.zero;

// width is stretch, so if we set it to 0, it will fill the Viewport width
// length is the length needed to fit all the content
contentRectTransform.sizeDelta = new Vector2(0, CalcFullContentHeight());
}

// Return the size needed for the entire content
protected float CalcFullContentHeight()
{
    float height = 0.0f;

    for (int i = 0; i < contentsElementDatas.Count; i++)
    {
        height += contentsElementDatas[i].height;
    }

    return height;
}

// Callback when the scrolling position is changed
protected void OnValueChanged(Vector2 position)
{
    float viewportHeight = scrollRect.viewport.rect.height;

    // Current scroll position
    float scrollY = (1.0f - position.y) * (scrollRect.content.rect.height - viewportHeight);

    // See if each UI element fits in the Viewport
    for (int i = 0; i < elementRectTransforms.Count; i++)
    {
        var data = contentsElementDatas[i];
        float elementY = data.y;
        float elementHeight = data.height;

        if (elementY + elementHeight < scrollY || elementY > scrollY + viewportHeight)
    }
}

```

```
{  
    // Since the UI element is outside the Viewport, return the object to the pool, if any.  
    // Use ReferenceEquals for comparison as it is faster.  
    // (Be careful with null after Destroy, but it's safe this time)  
    if (!ReferenceEquals(elementRectTransforms[i], null))  
    {  
        ReleaseElement(i);  
  
        // since it will be hidden by Mask or RectMask2D  
        // there is no need to deactivate the GameObject.  
    }  
}  
else  
{  
    // UI element is in the Viewport, so if it's not there, get the object from the pool.  
    if (ReferenceEquals(elementRectTransforms[i], null))  
    {  
        CreateElement(i);  
    }  
}  
}  
}
```

## Chapter 9 Auto Layout

### Auto Layout Overview

Auto Layout related components are used to control the size and position of `RectTransform`.

Auto Layout in a nutshell, it is

"A system that determines the position and size of a `RectTransform` using three types of properties: Minimum, Preferred, and Flexible."

Since Auto Layout-related components implement a specific interface, some of them, such as `Image` and `Text`, are subclasses of `Graphic` and are also Auto Layout-related components.

The Auto Layout interface is shown below.

- `ILayoutElement` : An interface to make an element eligible for Auto Layout; `Image`, `Text`, etc. also implement the `ILayoutElement` interface.
- `ILayoutGroup` : An interface for manipulating a child `RectTransform`. Examples of implementations are `HorizontalLayoutGroup`, `VerticalLayoutGroup`, `GridLayoutGroup`, and `ScrollView`.
- `ILayoutSelfController` : An interface for manipulating one's own `RectTransform`. Examples of implementations are `ContentSizeFitter` and `AspectRatioFitter`.
- `ILayoutController` : Parent interface for `ILayoutGroup` and `ILayoutSelfController`.
- `ILayoutIgnorer` : An interface to exclude itself from Auto Layout. It is implemented by the `LayoutElement` component, which is used to control the layout on its own.

`RectTransform`-based layout systems are flexible enough to handle many different kinds of layouts and allow for complete freedom in the placement of UI elements. However, there are times when a more structured system is needed.

The Auto Layout system provides a way to place UI elements in nested Layout Groups, such as horizontal or vertical groups or grids. It also allows elements to be automatically resized according to the content they contain. For example, it can dynamically resize to a size that perfectly fits text with padding.

The Auto Layout system is a system built on top of the basic `RectTransform` layout system and can be used as an option for some or all elements.

The Auto Layout system is based on the concept of Layout Element and Layout Controller. Layout Element is a `GameObject` with `RectTransform` and other components. A Layout Element does not

set its own size directly. The Layout Controller, on the other hand, uses the size information of the Layout Element to set the size of the [RectTransform](#) of the Layout Element.

The Layout Element has six properties: [minWidth](#), [minHeight](#), [preferredWidth](#), [preferredHeight](#), [flexibleWidth](#), and [flexibleHeight](#).

An example of a Layout Controller is the [ContentSizeFitter](#) and various [LayoutGroup](#) components.

## Rules for determining the size of the Layout Element

The basic rules for how the size of a Layout Element is determined within a Layout Group are as follows.

1. First, `minWidth` and `minHeight` are allocated.
2. The `preferredWidth` and `preferredHeight` are allocated if there is enough available space.
3. If there is more space available, `flexibleWidth` and `flexibleHeight` will be allocated.

All [GameObjects](#) in the Layout Group that have a [RectTransform](#) will act as Layout Elements.

Their min, preferred, and flexible sizes are set to `0` by default. When a specific component is assigned to a [GameObject](#), the values of these properties are changed.

The [Image](#) and [Text](#) components are examples of components that provide properties of the Layout Element. They match the `preferredWidth` and `preferredHeight` to the content of the [Sprite](#) or text.

## Manual sizing with the Layout Element component

If you want to adjust the sizes of min, preferred, and flexible yourself, you can attach the [LayoutElement](#) component to the [GameObject](#), enable the checkboxes for the properties you want to change, and specify the values.

The details of [LayoutElement](#) are described later.

## Layout Controller

A Layout Controller is a component that controls the size of one or more layout elements, where a layout element is a [GameObject](#) with a [RectTransform](#). The Layout Controller may control its own layout elements, or it may control the layout elements of its children.

A component that functions as a Layout Controller may also function as a layout element.

### ContentSizeFitter component to adjust its own size

The easiest way to see the Auto Layout system in action is to attach the [ContentSizeFitter](#) to a [GameObject](#) that has a [Text](#) component. The easiest way to see the Auto Layout system in action is to attach the [ContentSizeFitter](#) to a [GameObject](#) with a [Text](#) component.

If the [horizontalFit](#) or [verticalFit](#) of [ContentSizeFitter](#) is set to [PreferredSize](#), [RectTransform](#) will adjust the width or height to the contents of the [Text](#). The details are described later in the [ContentSizeFitter](#) section.

### AspectRatioFitter component

The [AspectRatioFitter](#) functions as a Layout Controller that controls the size of its own layout elements.

The [AspectRatioFitter](#) does not take into account layout information such as Minimum Size and Preferred Size.

## Layout Group

A Layout Group is a type of Layout Controller that controls the position and size of its child Layout elements. For example, a [HorizontalLayoutGroup](#) places its children next to each other, and a [GridLayoutGroup](#) places its children in a grid.

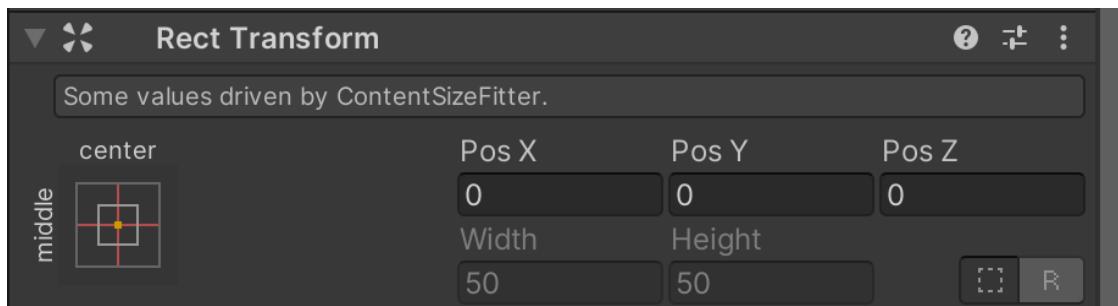
A Layout Group does not control its own size. Instead, it acts as a layout element that can be controlled by other Layout Controllers or controlled manually.

No matter what size is assigned to a Layout Group, it will most likely allocate the appropriate amount of space to each of its child layout elements based on the Minimum, Preferred, and Flexible sizes they report. Layout groups can be arbitrarily nested.

## Driven properties of RectTransform

Since the Layout Controller in the Auto Layout system automatically controls the size and placement of certain UI elements, their position and size should not be manually edited from the **Inspector** or **Scene View**. Such changed values will only end up being reset by the Layout Controller during the next layout calculation.

`RectTransform` has a concept of “driven properties” to solve this. For example, if the `horizontalFit` property of the `ContentSizeFitter` is set to `MinSize` or `PreferredSize`, then the width of the `RectTransform` is controlled by the `ContentSizeFitter`. In this case, the width will be displayed as read-only, and a small information box at the top of the `RectTransform` will indicate that the property is controlled by the `ContentSizeFitter`.



The driven property of `RectTransform` has a role other than preventing manual editing. Layouts can change simply by changing the screen resolution or the size of the **Game View**. This, in turn, changes the placement and size of the layout elements, which in turn changes the value of the driven property. However, it is not desirable for the Scene to be marked as having unsaved changes since the **Game View** has just been resized. To prevent this, the values of driven properties are not saved in the Scene. Also, even if the values of those properties are changed, it does not mean that the Scene has been changed.

If you want to set the driven properties yourself, you can do so using the `DrivenRectTransformTracker` class. The sample code is shown below.

```
using UnityEngine;

// Make RectTransform's properties uneditable from Inspector
[RequireComponent(typeof(RectTransform))]
[ExecuteAlways]
public class DrivenTransformPropertiesSample : MonoBehaviour
```

```

{
    // DrivenRectTransformTracker will work even if it is not assigned.
    #pragma warning disable 649
    private DrivenRectTransformTracker drivenRectTransformTracker;
    #pragma warning restore 649

    private void OnValidate()
    {
        #if UNITY_EDITOR
            var rectTransform = transform as RectTransform;

            // make Position and Size uneditable
            drivenRectTransformTracker.Add(this, rectTransform,
                DrivenTransformProperties.AnchoredPosition3D | DrivenTransformProperties.SizeDelta)
        ;
        #endif
    }

    private void OnDisable()
    {
        #if UNITY_EDITOR
            // we must always call Clear() in OnDisable()
            drivenRectTransformTracker.Clear();
        #endif
    }
}

```

## Create your own Auto Layout components

There are already several components in the Auto Layout system, but it is also possible to create new components that control the layout in your own way. To create your own component, you just need to implement a specific interface.

- Components that implement the [ILayoutElement](#) interface are treated as layout elements by the Auto Layout system.
- A component that implements the [ILayoutGroup](#) interface is expected to have driven control over its child [RectTransform](#).
- Components that implement the [ILayoutSelfController](#) interface are expected to control their own [RectTransform](#) as a driven property.

## Calculating the layout

The `minWidth`, `preferredWidth`, and `flexibleWidth` of a layout element are calculated by calling `CalculateLayoutInputHorizontal()` on the component that implements `ILayoutElement`. This calculation is done bottom-up. That is, the child is calculated before the parent, and the parent can take the child's information into account when calculating its own.

The actual width of the layout element is calculated and set by calling `SetLayoutHorizontal()` of the component that implements `ILayoutController`. This calculation is done in a top-down fashion. That is, the children are calculated after the parent, and the width of the children must be based on the width available to the parent. Once this step is completed, the width of the `RectTransform` of the layout element is determined.

The `minHeight`, `preferredHeight`, and `flexibleHeight` heights of a layout element are calculated by calling `CalculateLayoutInputVertical()` on a component that implements `ILayoutElement`. This calculation is done bottom-up. That is, the child is calculated before the parent, and the parent can take the child's information into account when calculating its own.

The actual width of the layout element is calculated and set by calling `SetLayoutVertical()` of the component that implements `ILayoutController`. This calculation is done in a top-down fashion. That is, the children are calculated after the parent, and the height of the children must be based on the full height available in the parent. Once this step is completed, the height of the `RectTransform` of the layout element is determined.

As we have seen, the Auto Layout system evaluates the width first, and then the height. So, the height may depend on the width, while the calculated width is never dependent on the height.

## Layout rebuild triggers

If the current layout is no longer valid due to changes in some properties of a component, a layout rebuild, or layout recalculation, is necessary. The layout rebuild can be done by calling

```
namespace UnityEngine.UI
{
    public class LayoutRebuilder : ICanvasElement
    {
        ...
        public static void MarkLayoutForRebuild(RectTransform rect)
```

`MarkLayoutForRebuild()` does not cause an immediate Layout rebuild; it occurs at the end of the current frame, just before rendering (see the Canvas Rebuild section of *Chapter 2 UI Element and Canvas Rebuild*). The reason why Layout rebuild does not occur immediately is that it may occur multiple times in the same frame, which is bad for performance.

Layout The timing at which a rebuild should be requested is as follows.

- In the setter of the property that changes the layout
- In the following callbacks
  - `OnEnable()`
  - `OnDisable()`
  - `OnRectTransformDimensionsChange()`
  - `OnValidate()` (only if required at Editor runtime)
  - `OnDidApplyAnimationProperties()`

## ILayoutElement interface

```
public interface ILayoutElement
```

A component that implements the [ILayoutElement](#) interface is a layout element that is subject to Auto Layout.

The layout system can precompute the values of [minWidth](#), [preferredWidth](#), and [flexibleWidth](#) by calling [CalculateLayoutInputHorizontal\(\)](#). This eliminates the need to calculate each property each time it is called.

Similarly, the layout system can precompute the values of [minHeight](#), [preferredHeight](#), and [flexibleHeight](#) by calling [CalculateLayoutInputVertical\(\)](#). This eliminates the need to calculate each property each time it is called.

The [minWidth](#), [preferredWidth](#), and [flexibleWidth](#) properties should not depend on any property of the [RectTransform](#) of the layout element. Otherwise, the behavior will be non-deterministic.

On the other hand, the [minHeight](#), [preferredHeight](#), and [flexibleHeight](#) properties may depend on the horizontal direction of the [RectTransform](#), such as the [width](#) and [position's x](#) elements. They may also depend on the [RectTransform](#) elements of the child layout elements.

## Properties of ILayoutElement

### minWidth

```
public float minWidth { get; }
```

Get the minimum width to be allocated by this layout element.

### minHeight

```
public float minHeight { get; }
```

Get the minimum height to be assigned by this layout element.

### preferredWidth

```
public float preferredWidth { get; }
```

Get the width that you want to set if there is enough space during Auto Layout.

If you set this property to -1, you can ignore the preferredWidth itself.

### preferredHeight

```
public float preferredHeight { get; }
```

Get the desired height to be set if there is enough space during Auto Layout.

If this property is set to -1, the preferredHeight itself can be ignored.

## flexibleWidth

---

```
public float flexibleWidth { get; }
```

Get the percentage of width that will be allocated to this layout element if there is extra space available.

## flexibleHeight

---

```
public float flexibleHeight { get; }
```

Get the percentage of height that this layout element would be allocated if there was extra space available.

## layoutPriority

---

```
public int layoutPriority { get; }
```

Get the layout priority of this component.

If more than one component that implements the [ILayoutElement](#) interface is attached to the same [GameObject](#), the component that returns the highest value for this property will be used. However, if each property is less than [0](#), it will be ignored, so it is possible to override a specific property.

## Public methods of ILayoutElement

### CalculateLayoutInputHorizontal

```
public void CalculateLayoutInputHorizontal();
```

Calculate `minWidth`, `preferredWidth`, and `flexibleWidth`.

After this method is called, the values of those properties should be up to date. In addition, the children `minWidth`, `preferredWidth`, and `flexibleWidth` should be up to date when this method is called.

### CalculateLayoutInputVertical

```
public void CalculateLayoutInputVertical();
```

Calculate the `minHeight`, `preferredHeight`, and `flexibleHeight`.

After this method is called, the values of those properties should be up to date. In addition, the children `minHeight`, `preferredHeight`, and `flexibleHeight` should be up to date when this method is called.

## ILayoutController interface

```
public interface ILayoutController
```

ILayoutController is an interface to control the layout of [RectTransform](#).

If you want to control your own [RectTransform](#), you should implement the [ILayoutSelfController](#) interface. If you want to control a child [RectTransform](#), implement the [ILayoutGroup](#) interface.

The layout system calls [SetLayoutHorizontal\(\)](#) first, and then [SetLayoutVertical\(\)](#).

It's no problem to call [LayoutUtility.GetMinWidth](#), [LayoutUtility.GetPreferredWidth](#), and [LayoutUtility.GetFlexibleWidth](#) for itself or its children inside [SetLayoutHorizontal](#).

And also, it's no problem to call [LayoutUtility.GetMinHeight](#), [LayoutUtility.GetPreferredHeight](#), and [LayoutUtility.GetFlexibleHeight](#) for itself or its children inside [SetLayoutVertical](#).

## Public method of ILayoutController

### SetLayoutHorizontal

```
public void SetLayoutHorizontal();
```

Called by the Auto Layout system to handle the horizontal direction of the layout.

### SetLayoutVertical

```
public void SetLayoutVertical();
```

Called by the Auto Layout system to handle the vertical direction of the layout.

## ILayoutGroup interface

[ILayoutGroup](#) is an [ILayoutController](#) that controls its child [RectTransform](#).

Since [ILayoutGroup](#) does not add anything specific to [ILayoutController](#), the contents are identical.

## ILayoutSelfController interface

[ILayoutSelfController](#) is an [ILayoutController](#) that controls its own [RectTransform](#).

Since [ILayoutSelfController](#) does not add anything special to [ILayoutController](#), the contents are identical.

If you want to control the [RectTransform](#) of the [GameObject](#) itself, use [ILayoutSelfController](#), and if you want to control the [RectTransform](#) of the [GameObject](#)'s children, use [ILayoutGroup](#).

[SetLayoutHorizontal](#) if you want to control the horizontal direction of the layout, and call [ILayoutController.SetLayoutVertical](#) if you want to control the vertical direction of the layout. It is possible to change the width, height, position, and rotation of [RectTransform](#).

## ILayoutIgnorer interface

```
public interface ILayoutIgnorer
```

[RectTransform](#) is ignored by the layout system if you have a component that implements [ILayoutIgnorer](#).

A component that implements [ILayoutIgnorer](#) can make its parent Layout Group component not treat this [RectTransform](#) as part of the group. In this way, the [RectTransform](#) can set its own layout even though it is a child [GameObject](#) of the Layout Group.

## Properties of ILayoutIgnorer

### ignoreLayout

```
bool ignoreLayout { get; }
```

Determines whether the [RectTransform](#) should be ignored by the layout system. If this property returns [true](#), the parent Layout Group component will not consider this [RectTransform](#) to be part of the group. This [RectTransform](#) can then be placed manually, even though it is also a child of the layout group.

## LayoutElement component

```
[AddComponentMenu("Layout/Layout Element", 140)]
[RequireComponent(typeof(RectTransform))]
[ExecuteAlways]
public class LayoutElement : UIBehaviour, ILayoutElement, ILayoutIgnorer
```

LayoutElement is a component used to override the `minWidth`, `minHeight`, `preferredWidth`, `preferredHeight`, `flexibleWidth`, and `flexibleHeight` of an existing layout element.

Set a value greater than or equal to `0` for the value you want to override, and leave the value you do not want to override at `-1`. Alternatively, you can set `ignoreLayout` to `true` to ignore the Auto Layout placement.

Here are some examples of usage

- When `ContentSizeFitter` is attached to `Text` and set to `PreferredSize`, if the text string is empty, the size of `RectTransform` will be `(0, 0)`. In this case, if you attach `LayoutElement` and set `minWidth` and `minHeight`, you can keep the `RectTransform` rectangle even if the text string is empty.
- Attach `LayoutElement` to a `GameObject` that has an `Image` that is a child of `VerticalLayoutGroup`. You can then manually adjust the position by turning on `ignoreLayout` in `LayoutElement`.

The description of properties and methods is omitted since they are identical to those of the `ILayoutElement` interface.

## LayoutGroup component

```
[DisallowMultipleComponent]  
[ExecuteAlways]  
[RequireComponent(typeof(RectTransform))]  
public abstract class LayoutGroup : UIBehaviour, ILayoutElement, ILayoutGroup
```

LayoutGroup is the parent class of [HorizontalLayoutGroup](#), [VerticalLayoutGroup](#), [HorizontalOrVerticalLayoutGroup](#), and [GridLayoutGroup](#).

LayoutGroup is an abstract class, so it cannot be attached to a [GameObject](#).

## Properties of LayoutGroup

### padding

```
public RectOffset padding { get; set; }
```

Gets/Sets the padding to be added around the child layout element.

This property is used by the GridLayoutGroup component.

### childAlignment

```
public TextAnchor childAlignment { get; set; }
```

Gets/Sets the alignment to be used for the layout elements of the Layout Group's children.

We have already discussed [TextAnchor](#) in the section on the [alignment](#) property of the [Text](#) component, but let's reiterate the definition.

```
namespace UnityEngine
{
    public enum TextAnchor
    {
        UpperLeft,
        UpperCenter,
        UpperRight,
        MiddleLeft,
        MiddleCenter,
        MiddleRight,
        LowerLeft,
        LowerCenter,
        LowerRight
    }
}
```

If a layout element does not have `flexibleWidth` or `flexibleHeight` set, then its child elements may not be able to use the space available in the Layout Group. In this case, this alignment setting is used to indicate how the children should be aligned in the Layout Group.

### minWidth

---

```
public virtual float minWidth { get; }
```

Get the minimum width to be allocated by this layout element.

This property is an implementation of the [ILayoutElement](#) interface.

[0](#) is returned except for [GridLayoutGroup](#).

### minHeight

---

```
public virtual float minHeight { get; }
```

Get the minimum width to be allocated by this layout element.

This property is an implementation of the [ILayoutElement](#) interface.

[0](#) is returned except for [GridLayoutGroup](#).

### preferredWidth

---

```
public virtual float preferredWidth { get; }
```

Get the width you want to set if there is enough space.

This property is an implementation of the [ILayoutElement](#) interface.

[0](#) is returned except for [GridLayoutGroup](#).

## preferredHeight

---

```
public virtual float preferredHeight { get; }
```

Get the height that you want to set if there is enough space.

This property is an implementation of the [ILayoutElement](#) interface.

0 is returned except for [GridLayoutGroup](#).

## flexibleWidth

---

```
public virtual float flexibleWidth { get; }
```

Get the percentage of width that will be allocated to this layout element if there is extra space available.

This property is an implementation of the [ILayoutElement](#) interface.

Returns -1 for [GridLayoutGroup](#), 0 for other components.

## flexibleHeight

---

```
public virtual float flexibleHeight { get; }
```

Gets the percentage of height that this layout element will be allocated if there is extra space available. This property implements the [ILayoutElement](#) interface.

Returns -1 for [GridLayoutGroup](#), 0 for other components.

## layoutPriority

---

```
public virtual int layoutPriority { get; }
```

Get the layout priority of this component.

This property is an implementation of the [ILayoutElement](#) interface.

If more than one component that implements the [ILayoutElement](#) interface is attached to the same [GameObject](#), the component that returns the highest value for this property will be used. However, if each property is less than zero, it will be ignored, so it is possible to override a specific property.

Always returns 0.

## Public methods of LayoutGroup

### CalculateLayoutInputHorizontal

```
public virtual void CalculateLayoutInputHorizontal();
```

It is supposed to be a method to calculate `minWidth`, `preferredWidth`, and `flexibleWidth`, but in fact it just enumerates `RectTransforms` that do not have `ILayoutIgnorer`. This method is an implementation of the `ILayoutElement` interface.

### CalculateLayoutInputHorizontal

```
public abstract void CalculateLayoutInputVertical();
```

It is supposed to be a method to calculate `minHeight`, `preferredHeight`, and `flexibleHeight`, but this method is not implemented in `LayoutGroup`.

## HorizontalOrVerticalLayoutGroup component

[ExecuteAlways]

```
public abstract class HorizontalOrVerticalLayoutGroup : LayoutGroup, ILayoutElement, ILayoutGroup, ILayoutController
```

HorizontalOrVerticalLayoutGroup is a class that implements the functions common to [HorizontalLayoutGroup](#) and [VerticalLayoutGroup](#), and is their parent class.

## Properties of HorizontalOrVerticalLayoutGroup

### childControlWidth

```
public bool childControlWidth { get; set; }
```

Gets/Sets whether or not this Layout Group controls the width of its children.

The default value returns [true](#).

If this property is set to [false](#), then this Layout Group will only affect the width of its children. In this case, the width of the child can be set via [RectTransform](#).

If this property is set to [true](#), then the width of the child is automatically set according to [minWidth](#), [flexibleWidth](#), and [preferredWidth](#). This is useful when the width of a child varies depending on the available space. In this case, the width of each child cannot be set via [RectTransform](#). However, [minWidth](#), [flexibleWidth](#), and [preferredWidth](#) can be controlled by adding a [LayoutElement](#) component.

### childControlHeight

```
public bool childControlHeight { get; set; }
```

Gets/Sets whether or not this Layout Group controls the height of its children.

The default value is [true](#).

If this property is set to [false](#), then this Layout Group only affects the height of its children. In that case, the height of the child can be set via [RectTransform](#).

If this property is set to [true](#), the height of the child will be automatically set according to [minHeight](#), [flexibleHeight](#), and [preferredHeight](#). This is useful when the height of the child changes depending on the available space. In this case, the height of each child cannot be set via [RectTransform](#). However, the [minHeight](#), [flexibleHeight](#), and [preferredHeight](#) can be controlled by adding a [LayoutElement](#) component.

## childForceExpandWidth

---

```
public bool childForceExpandWidth { get; set; }
```

Gets/Sets whether any extra horizontal space will force the child to fill it.

The default value is [true](#).

## childForceExpandHeight

---

```
public bool childForceExpandHeight { get; set; }
```

Gets/Sets whether extra vertical space forces the child to fill it or not.

The default value is [true](#).

## childScaleWidth

---

```
public bool childScaleWidth { get; set; }
```

Gets/Sets whether to use the [localScale](#) of the child [RectTransform](#) when calculating the width of the child.

The default value is [false](#).

## childScaleHeight

---

```
public bool childScaleHeight { get; set; }
```

Gets/Sets whether to use the [localScale](#) of the child [RectTransform](#) when calculating the height of the child.

The default value is [false](#).

## reverseArrangement

---

```
public bool reverseArrangement { get; set; }
```

Gets/Sets whether the order of the children should be sorted in reverse.

The default value is [false](#).

If this property is [false](#), the first child object will be placed first. If this property is [true](#), the last child object will be placed first.

## spacing

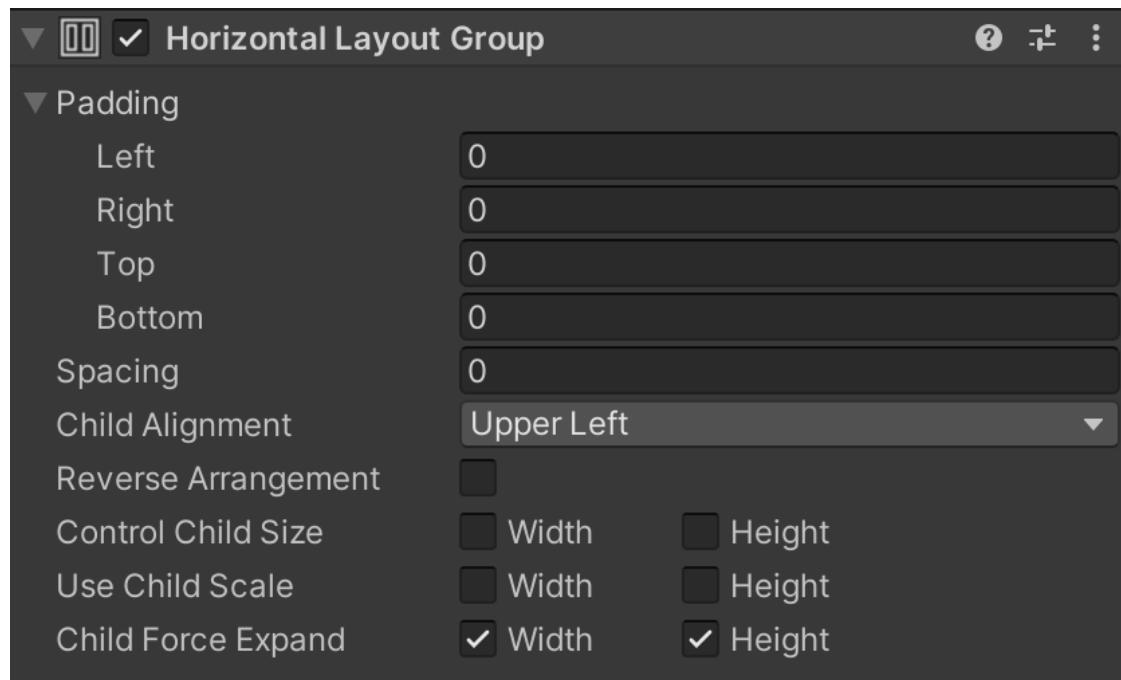
---

```
public float spacing { get; set; }
```

Gets/Sets the spacing between layout elements in a Layout Group.

The default value is [0](#).

## HorizontalLayoutGroup component



```
[AddComponentMenu("Layout/Horizontal Layout Group", 150)]
```

```
public class HorizontalLayoutGroup : HorizontalOrVerticalLayoutGroup, ILayoutElement, ILayoutGroup, ILayoutController
```

HorizontalLayoutGroup is a component for arranging child layout elements in the horizontal direction.

Almost all of the implementation is in the form of using the functions implemented in the parent class, [HorizontalOrVerticalLayoutGroup](#).

The width of each layout element is calculated as follows using [minWidth](#), [preferredWidth](#), and [flexibleWidth](#).

1. The result of adding all the [minWidths](#) of all the child layout elements (and the spaces between them) shall be the [minWidth](#) of the [HorizontalLayoutGroup](#).
2. The [preferredHeight](#) of the [HorizontalLayoutGroup](#) shall be the result of adding all the [preferredHeights](#) of all the child layout elements (and the spaces between them).

3. If the width of `HorizontalLayoutGroup` is less than or equal to `minWidth`, then the width of the child layout elements will be their respective `minWidths`.
4. If the width of the `HorizontalLayoutGroup` is greater than or equal to the `minWidth` and less than or equal to the `preferredWidth`, then the closer the width of the `HorizontalLayoutGroup` is to the `preferredWidth`, the closer the widths of the child layout elements will be to their respective `preferredWidth`.
5. If the width of `HorizontalLayoutGroup` is larger than the `preferredWidth`, the extra width is distributed to the child layout elements according to the `flexibleWidth`.

## Public methods of HorizontalLayoutGroup

### CalculateLayoutInputHorizontal

```
public virtual void CalculateLayoutInputHorizontal();
```

This method implements [CalculateLayoutInputHorizontal\(\)](#) of the [ILayoutElement](#) interface.

### CalculateLayoutInputVertical

```
public virtual void CalculateLayoutInputVertical();
```

This method implements [CalculateLayoutInputVertical\(\)](#) of the [ILayoutElement](#) interface.

### SetLayoutHorizontal

```
public override void SetLayoutHorizontal();
```

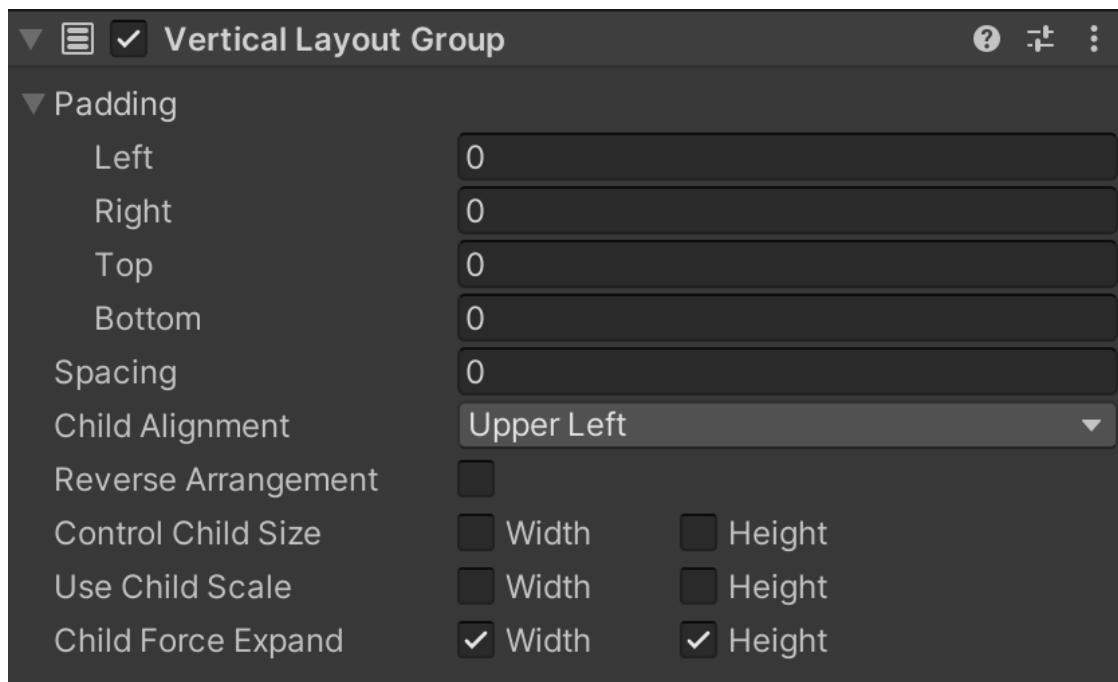
Called to handle the horizontal direction of the layout.

### SetLayoutVertical

```
public override void SetLayoutVertical();
```

Called to handle the vertical direction of the layout.

## VerticalLayoutGroup component



```
[AddComponentMenu("Layout/Vertical Layout Group", 151)]  
public class VerticalLayoutGroup : HorizontalOrVerticalLayoutGroup, ILayoutElement, ILayout  
Group, ILayoutController
```

`VerticalLayoutGroup` is a component for vertically arranging child layout elements.

Almost all of the implementation is in the form of using the functions implemented in the parent class, `HorizontalOrVerticalLayoutGroup`.

The width of each layout element is calculated as follows using `minWidth`, `preferredWidth`, and `flexibleWidth`.

1. The result of adding all the `minHeights` of all the child layout elements (and the spaces between them) shall be the `minHeight` of the `VerticalLayoutGroup`.
2. The `preferredHeight` of `VerticalLayoutGroup` shall be the result of adding all the `preferredHeights` of all the child layout elements (and the spaces between them).
3. If the height of `VerticalLayoutGroup` is less than or equal to `minHeight`, then the height of the child layout elements will be their respective `minHeights`.

4. If the width of the `VerticalLayoutGroup` is greater than or equal to the `minHeight` and less than or equal to the `preferredHeight`, then the closer the width of the `VerticalLayoutGroup` is to the `preferredHeight`, the closer the width of the child layout elements will be to their respective `preferredHeight`.
5. If the width of `VerticalLayoutGroup` is greater than the `preferredHeight`, the excess width is distributed to the child layout elements according to the `flexibleHeight`.

## VerticalLayoutGroup public method

### CalculateLayoutInputHorizontal

```
public virtual void CalculateLayoutInputHorizontal();
```

This method implements [CalculateLayoutInputHorizontal\(\)](#) of the [ILayoutElement](#) interface.

### CalculateLayoutInputVertical

```
public virtual void CalculateLayoutInputVertical();
```

This method implements [CalculateLayoutInputVertical\(\)](#) of the [ILayoutElement](#) interface.

### SetLayoutHorizontal

```
public override void SetLayoutHorizontal();
```

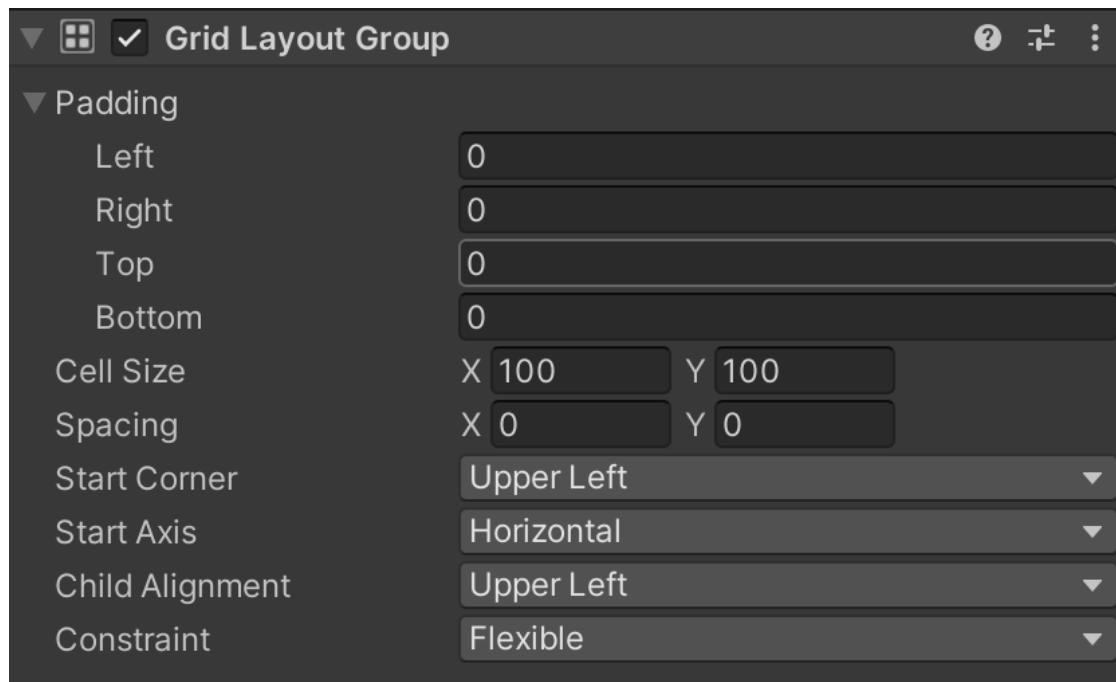
This method is called to handle the horizontal direction of the layout.

### SetLayoutVertical

```
public override void SetLayoutVertical();
```

This method is called to handle the vertical direction of the layout.

## GridLayoutGroup component



[AddComponentMenu("Layout/Grid Layout Group", 152)]

```
public class GridLayoutGroup : LayoutGroup, ILayoutElement, ILayoutGroup, ILayoutController
```

GridLayoutGroup is a [LayoutGroup](#) that aligns the layout elements under it horizontally and vertically, and arranges them like a grid.

Unlike other [LayoutGroups](#), GridLayoutGroup ignores the min / preferred / flexible sizes of the internal layout elements and applies the [cellSize](#) property of [LayoutGroup](#), which is a fixed size, to all layout elements.

When using [GridLayoutGroup](#) as part of Auto Layout such as [ContentSizeFitter](#), there are a few points that need to be considered.

The Auto Layout system calculates the horizontal and vertical sizes separately. However, this does not mesh with the [GridLayoutGroup](#) mechanism, where the number of rows and the number of columns depend on each other.

For a given number of cells, there can be several combinations of the number of rows and columns that fill the grid. To help the layout system, the `constraint` property can be used to fix the number of rows or columns.

We suggest several ways to use the layout system together with `ContentSizeFitter`.

### Combination of `flexibleWidth` and `fixed height`

If you want the grid to expand horizontally as more elements are added, you can use a combination of `flexibleWidth` and fixed height, in which case, set each property as follows

- Set constraint of `GridLayoutGroup` to `FixedRowCount`
- Set `horizontalFit` of `ContentSizeFitter` to `PreferredSize`
- Set `verticalFit` of `ContentSizeFitter` to `PreferredSize` or `Unconstrained`

However, if `verticalFit` is `Unconstrained`, you will have to calculate and prepare the sufficient height for the specified number of cell columns by yourself.

### Combination of `fixed width` and `flexibleHeight`

If you want the grid to expand vertically as the number of elements increases, you can use a combination of `fixed width` and `flexibleHeight`, in which case, set each property as follows

- Set constraint of `GridLayoutGroup` to `FixedColumnCount`
- Set the `horizontalFit` of `ContentSizeFitter` to `PreferredSize` or `Unconstrained`.
- Set `verticalFit` of `ContentSizeFitter` to `Preferred Size`

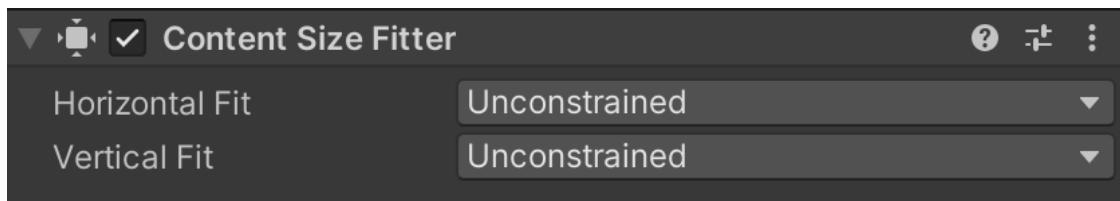
However, if `horizontalFit` is `Unconstrained`, you must calculate and prepare a sufficient width to fit the number of rows in the specified cell by yourself.

## Combination of flexibleWidth and flexibleHeight

It is possible to use a grid with a combination of `flexibleWidth` and `flexibleHeight`, but it is not possible to fix the number of rows and columns. The grid will try to have approximately the same number of rows and columns. In this case, set each property as follows.

- Set constraint of `GridLayoutGroup` to `Flexible`
- Set `horizontalFit` of `ContentSizeFitter` to `PreferredSize`
- Set `verticalFit` of `ContentSizeFitter` to `Preferred Size`

## ContentSizeFitter component



```
[AddComponentMenu("Layout/Content Size Fitter", 141)]
[ExecuteAlways]
[RequireComponent(typeof(RectTransform))]
public class ContentSizeFitter : UIBehaviour, ILayoutSelfController, ILayoutController
```

ContentSizeFitter is a component to fit RectTransform to the size of its content.

ContentSizeFitter can be used for GameObjects that have one or more ILayoutElement(s), such as Text, Image, HorizontalLayoutGroup, VerticalLayoutGroup, GridLayoutGroup, etc. such as HorizontalLayoutGroup, VerticalLayoutGroup, or GridLayoutGroup.

The ContentSizeFitter functions as a layout controller to control the size of its own layout elements. The size is determined by the minimum and preferred layout elements attached to the GameObject. Such layout elements can be Image, Text, Layout Group, or Layout Element components.

It should be noted that when a RectTransform is resized (whether by the ContentSizeFitter or not), the resizing is done around the pivot. This means that the direction of the resize can be controlled using the pivot.

For example, if the pivot was in the center, the ContentSizeFitter would scale the RectTransform equally in all directions. If the pivot was at the top left, ContentSizeFitter will scale RectTransform to the bottom right.

## Properties of ContentSizeFitter

### horizontalFit

```
public ContentSizeFitter. FitMode horizontalFit { get; set; }
```

Get/Sets the Fit mode for determining the width.

The Fit mode is defined as follows.

```
/// <summary>
/// Available Fit Modes
/// </summary>
public enum FitMode
{
    /// <summary>
    /// No resizing will be done
    /// </summary>
    Unconstrained,

    /// <summary>
    /// Resize to minimum size
    /// </summary>
    MinSize,

    /// <summary>
    /// Resize to preferred size
    /// </summary>
    PreferredSize
}
```

The default value is **Unconstrained**, which means that no resizing will be performed.

### verticalFit

```
public ContentSizeFitter. FitMode verticalFit { get; set; }
```

Get/Sets the Fit mode for determining the height.

The default value is [Unconstrained](#), which means that no resizing will be performed.

## Public methods of ContentSizeFitter

### SetLayoutHorizontal

```
public virtual void SetLayoutHorizontal();
```

This is the method that the Auto Layout system calls to handle the horizontal direction of the layout.

This method is a method that implements the [ILayoutController](#) interface.

Depending on the [horizontalFit](#) value, set the [minWidth](#) or [preferredWidth](#) to the width of [RectTransform](#). To do so, use [RectTransform's SetSizeWithCurrentAnchors\(\)](#) method.

### SetLayoutVertical

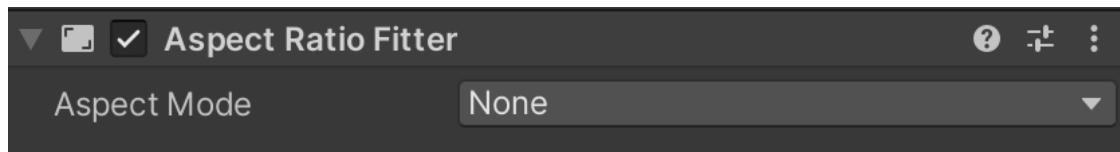
```
public virtual void SetLayoutVertical();
```

This is the method that the Auto Layout system calls to handle the vertical direction of the layout.

This method is a method that implements the [ILayoutController](#) interface.

Depending on the [verticalFit](#) value, set the [minHeight](#) or [preferredHeight](#) to the width of [RectTransform](#). To do so, use [RectTransform's SetSizeWithCurrentAnchors\(\)](#) method.

## AspectRatioFitter component



```
[AddComponentMenu("Layout/Aspect Ratio Fitter", 142)]
[ExecuteAlways]
[RequireComponent(typeof(RectTransform))]
[DisallowMultipleComponent]
public class AspectRatioFitter : UIBehaviour, ILayoutSelfController, ILayoutController
```

AspectRatioFitter is a component to resize `RectTransform` based on the specified aspect ratio and control method.

## Properties of AspectRatioFitter

### aspectMode

```
public AspectRatioFitter.AspectMode aspectMode { get; set; }
```

Gets/Sets the aspect ratio control method.

The definition of `AspectRatioFitter.AspectMode` is as follows.

```
/// <summary>
/// Specifies the mode to be used to enforce the aspect ratio.
/// </summary>
public enum AspectMode
{
    /// <summary>
    /// Do not fix the aspect ratio.
    /// </summary>
    None,

    /// <summary>
    /// Automatically set the height according to the aspect ratio.
    /// </summary>
    WidthControlsHeight,

    /// <summary>
    /// Auto-set the width according to the aspect ratio.
    /// </summary>
    HeightControlsWidth,

    /// <summary>
    /// Automatically set the height and width so that it is fully contained in (not overflowing into)
    /// the parent rectangle while maintaining the aspect ratio.
    /// </summary>
    FitInParent,
}

/// <summary>
/// Automatically set the height and width so that the parent rectangle is completely contained i
```

n this rectangle, while maintaining the aspect ratio.

```
/// </summary>
EnvelopeParent
}
```

The default value is [None](#).

[aspectRatio](#)

```
public float aspectRatio { get; set; }
```

Get/Sets the aspect ratio (width divided by height).

For example, if the width is [200](#) and the height is [100](#), the aspect ratio would be [2](#).

The default value is [1](#).

## Public methods of AspectRatioFitter

### IsAspectModeValid

```
public bool IsAspectModeValid()
```

Returns whether this aspect ratio control is successful or not.

In practice, `false` is returned when the `aspectMode` is `FitInParent` or `EnvelopeParent`, but the parent does not exist. Otherwise, `true` is returned.

### IsComponentValidOnObject

```
public bool IsComponentValidOnObject()
```

Returns whether this component is valid or not.

In fact, it returns `false` if a `Canvas` is attached to the same `GameObject` and the `renderMode` of the `Canvas` is not `WorldSpace`.

### SetLayoutHorizontal

```
public virtual void SetLayoutHorizontal()
```

It is called by the Auto Layout system to handle the horizontal direction of the layout, but its contents are empty.

### SetLayoutVertical

```
public virtual void SetLayoutVertical()
```

It is called by the Auto Layout system to handle the vertical direction of the layout, but its contents are empty.

## Use Anchor instead of Auto Layout

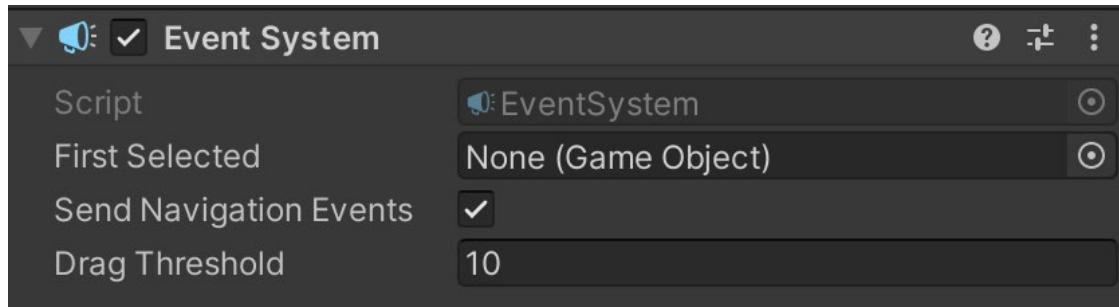
In this chapter, we have described the Auto Layout system, but Auto Layout is not always a panacea. For example, by properly assigning [RectTransform](#) anchors, the position and size of a [RectTransform](#) can be adjusted according to its parent.

The calculation of the size and position of [RectTransform](#) is done in native code by the Transform system itself. This is generally more performant than relying on the Layout system. It is also possible to write a [MonoBehaviour](#) that sets up a [RectTransform](#) based layout, so before using Auto Layout, think about whether you can use anchors to achieve the desired layout.

## Chapter 10 EventSystem

### EventSystem Components

```
[AddComponentMenu("Event/Event System")]
[DisallowMultipleComponent]
public class EventSystem : UIBehaviour
```



The [EventSystem](#) is a mechanism for sending events to objects in applications that support keyboard, mouse, screen touch, etc. The [EventSystem](#) consists of several modules for sending events.

Even if you attach an [EventSystem](#) to a [GameObject](#), the [Inspector](#) will not show most of its functions. This is because [EventSystem](#) is designed to be a manager for interfacing between various modules.

The main roles of the [EventSystem](#) are as follows.

- Manage the currently selected [GameObject](#).
- Manage the currently used [InputModule](#).
- Update the [InputModule](#) every frame.
- Manage raycasts.

[InputModule](#) will be described later, but it is mainly responsible for processing input, managing the state of events, and sending events to objects.

An [EventSystem](#) is responsible for handling events in a scene, and only one [EventSystem](#) should be placed in a scene. If you have more than one [EventSystem](#) in a scene (including an additive scene), you will get a warning, so be careful when using multiple scenes.

The [EventSystem](#) looks for and holds the [InputModule](#)(s) attached to the same [GameObject](#)(s).

## Update() of EventSystem

The [EventSystem](#) performs the following processing during [Update\(\)](#).

- [UpdateModule\(\)](#) of an InputModule is used to update the mouse position and to handle click and drag events. For example, [OnPointerClick\(\)](#) of the [IPointerClickHandler](#) interface is called from [UpdateModule\(\)](#) when clicking.
- Call [Process\(\)](#) of the currently active InputModule to perform the following process.
  - Call [OnUpdateSelected\(\)](#) of the [IUpdateSelectedHandler](#) interface on the selected object.
  - Check the status of the mouse buttons.
  - Call [OnMove\(\)](#) of the [IMoveHandler](#) interface if necessary.
  - Call [OnSubmit\(\)](#) of the [ISubmitHandler](#) interface if necessary.
  - Call [OnCancel\(\)](#) of [ICancelHandler](#) interface if necessary.

### Note on disabling EventSystem.

You may want to disable the currently active `EventSystem` in order to temporarily disable all input.

```
EventSystem.current.enabled = false;
```

This in itself is not a problem, but in most cases, `EventSystem.current` will become `null` immediately after this setting is made. So, if you try to enable the input again, the

```
EventSystem.current.enabled = true;
```

will result in a `NullReferenceException`. To avoid this, you need to have a separate reference to `EventSystem` as follows.

```
EventSystem es = EventSystem.current;  
EventSystem.current.enabled = false;  
es.enabled = true;
```

However, if you want to enable/disable all inputs, it may be more straightforward to create your own `InputModule` and control it there. How to create your own `InputModule` will be described later.

## Static properties of EventSystem

current

```
public static EventSystem current { get; set; }
```

Gets/Sets the currently active [EventSystem](#).

An [EventSystem](#) registers itself in the list of static [EventSystem](#) when [OnEnable\(\)](#) is called. This is the active [EventSystem](#).

Conversely, when [OnDisable\(\)](#) is called, it removes itself from the list of static [EventSystem](#) objects, so that [current](#) becomes another object in the list or [null](#). Since there is essentially only one [EventSystem](#) object at a time, [current](#) will be [null](#) after [OnDisable\(\)](#) is called.

## Properties of EventSystem

### alreadySelecting

```
public bool alreadySelecting { get; }
```

Get whether or not the currently selected object is already set.

Make sure that this property is not `true` when `Selectable` tries to register itself as the currently selected object.

### currentInputModule

```
public BaseInputModule currentInputModule { get; }
```

Get the currently active InputModule.

### currentSelectedGameObject

```
public GameObject currentSelectedGameObject { get; }
```

Gets the currently selected `GameObject`.

The currently selected `GameObject` is set by the `SetSelectedGameObject()` method.

### firstSelectedGameObject

```
public GameObject firstSelectedGameObject { get; set; }
```

Gets/Sets the `GameObject` that will be the first currently selected object.

The default value is `null`, so initially no object is selected.

## isFocused

---

```
public bool isFocused { get; }
```

Gets whether or not [EventSystem](#) is in the focus state.

The default value is [true](#).

[OnApplicationFocus\(bool\)](#). The value of this property is the focus status of the application as notified by [MonoBehaviour](#).

If the application does not have the focus, the InputModule will not process every frame.

## pixelDragThreshold

---

```
public int pixelDragThreshold { get; set; }
```

Gets/Sets the threshold for judging the start of a drag.

The default value is [10](#).

If the distance between the previous position of the mouse or finger and the current position is less than or equal to this property, the dragging is not considered to have started.

However, for [Slider](#) and [Scrollbar](#), this threshold is not used because the [useDragThreshold](#) of [eventData](#) is set to [false](#) in [OnInitializePotentialDrag\(\)](#).

## sendNavigationEvents

---

```
public bool sendNavigationEvents { get; set; }
```

Gets/Sets whether navigation events such as Move, Submit, and Cancel are allowed.

The default value is [true](#).

If [true](#), [InputManager](#) will handle the navigation event in [Process\(\)](#).

## Public methods of EventSystem

### IsPointerOverGameObject

```
public bool IsPointerOverGameObject();
public bool IsPointerOverGameObject(int pointerId);
```

Returns whether a pointer with the ID specified by the argument exists on any [GameObject](#) (or, in other words, whether an object that receives [OnPointerEnter](#) exists). If the argument is omitted, a normal mouse pointer will be judged.

The sample code for this method is shown below.

```
void Update()
{
    // Check if the mouse button has been pressed.
    if (Input.GetMouseButton(0))
    {
        // check if the mouse is over the UI
        if (EventSystem.current.IsPointerOverGameObject())
        {
            Debug.Log("Mouse pressed on UI");
        }
    }
}
```

The following Pointer IDs are defined.

Value	Definition.	Description
Any integer greater than or equal to 0	Touch.fingerId	Index of the finger being touched.
-1	PointerInputModule.kMouseLeftId	Left Mouse Pointer Event
-2	PointerInputModule.kMouseRightId	Right Mouse Pointer Event
-3	PointerInputModule.kMouseMiddleId	Naka Mousse Pointer Event

-4	PointerInputModule.kFakeTouchesId	Touch simulation events on non-touch devices
----	-----------------------------------	--

## RaycastAll

---

```
public void RaycastAll(PointerEventData eventData, List<RaycastResult> raycastResults);
```

Raycast the scene using all currently existing Raycasters and store the results in `raycastResults`.

Raycasting is discussed later in the *Raycaster* section.

## SetSelectedGameObject

---

```
public void SetSelectedGameObject(GameObject selected);
public void SetSelectedGameObject(GameObject selected, BaseEventData pointer);
```

Sets the currently selected `GameObject`.

The `GameObject` set here can be retrieved with the `currentSelectedGameObject` property. However, if `alreadySelecting` property is set to `true`, it will not be set and an error will be output.

`OnDeselect()` is called for a `GameObject` that has already been set, and `OnSelect()` is called for a newly set `GameObject`.

When `Selectable` calls this method from `OnPointerDown(PointerEventData eventData)`, it passes `eventData` as a pointer.

## UpdateModules

---

```
public void UpdateModules();
```

Find and hold one or more `InputModule`(s) attached to the same `GameObject`.

This method is called at the timing of `OnEnable()` and `OnDisable()` of `BaseInputModule`.

## Events supported by EventSystem

The [EventSystem](#) supports a large number of events, which can be customized with your own [InputModule](#).

The events supported by [StandaloneInputModule](#) are provided by an interface, which can be implemented by a [MonoBehaviour](#) that implements that interface. If it has a valid [EventSystem](#), the event will be called at the appropriate time.

The following is the interface of the event and the methods to be implemented in that interface.

- [IPointerEnterHandler](#) : Implements [OnPointerEnter\(\)](#). This method is called when a pointer overlaps an object.
- [IPointerExitHandler](#) : Implements [OnPointerExit\(\)](#). This method is called when the pointer leaves the object.
- [IPointerDownHandler](#) : Implement [OnPointerDown\(\)](#). This method is called when a pointer is pressed.
- [IPointerUpHandler](#) : Implements [OnPointerUp\(\)](#). This method is called when the pointer is released from the tamper.
- Implement '[IPointerClickHandler](#) : [OnPointerClick\(\)](#)'. This method is called when the click is confirmed as a result of pressing and speaking the pointer.
- [InitiallyInitializePotentialDragHandler](#) : Implement [OnInitializePotentialDrag\(\)](#). This method is called just before the dragging process starts.
- [IBeginDragHandler](#) : Implement [OnBeginDrag\(\)](#). This method is called when the dragging actually starts (i.e., when the touch position moves).
- [IDragHandler](#) : Implement [OnDrag\(\)](#). This method is called when there is a movement while dragging.
- [IEndDragHandler](#) : Implement [OnEndDrag\(\)](#). This method is called when the dragging is finished.
- [IDropHandler](#) : Implement [OnDrop\(\)](#). This method is called when a drop is performed.
- [IScrollHandler](#) : Implement [OnScroll\(\)](#). This method is called when scrolling with the mouse wheel occurs.
- [IUpdateSelectedHandler](#) : Implement [OnUpdateSelected\(\)](#). This method is called every frame when a game object is currently selected.
- [ISelectHandler](#) : Implement [OnSelect\(\)](#). This method is called when a game object is selected.
- [IDeselectHandler](#) : Implement [OnDeselect\(\)](#). This method is called when a game object is deselected.
- [IMoveHandler](#) : Implement [OnMove\(\)](#). This method is called when the Move event is triggered by navigation.
- [ISubmitHandler](#) : Implement [OnSubmit\(\)](#). This method is called when the Submit button is pressed.

- **ICancelHandler** : Implement [OnCancel\(\)](#). This method is called when the Cancel button is pressed.

## Messaging System

The uGUI uses a new Messaging System designed to replace the existing [SendMessage](#). This Messaging System is written in pure C# and is designed to solve the problems of [SendMessage](#).

This Messaging System is capable of sending its own data. You can also specify how far down the hierarchy you want to send events. They can be sent only to specific [GameObjects](#), or to children or parents. There is also a helper function to find [GameObjects](#) that implement the messaging interface.

The Messaging System is designed to be used not only for uGUI, but also for general purpose. It is relatively easy to add your own messaging events, and it works using the same framework that uGUI uses for all its event handling.

## Send your own messages using the Messaging System

It is relatively easy to send your own messages using the Messaging System; there is an interface called [IEventSystemHandler](#) in the [UnityEngine](#). [IEventSystemHandler](#) exists in the [UnityEngine.EventSystems](#) namespace, and any interface that inherits from it will be the target for receiving events via the Messaging System.

```
using UnityEngine.EventSystems;

// Interface for receiving custom messages
public interface ICustomMessageTarget : IEventSystemHandler
{
    void Message();
    void MessageWithData(BaseEventData eventData);
    void MessageWithCustomData(string customData);
    ...
}
```

After defining the interface as described above, inherit it with the [MonoBehaviour](#). Then, when a message is issued to the [GameObject](#) of this [MonoBehaviour](#), the method will be executed.

```
using UnityEngine;
using UnityEngine.EventSystems;

// An object that can receive custom messages
public class CustomMessageTarget : MonoBehaviour, ICustomMessageTarget
{
    public void Message()
    {
        Debug.LogFormat("Message() of {0} was called", this.name);
    }

    public void MessageWithData(BaseEventData eventData)
    {
        Debug.LogFormat("MessageWithData() of {0} was called", this.name);
    }

    public void MessageWithCustomData(string customData)
    {
        Debug.LogFormat("MessageWithCustomData() for {0} was called : {1}", this.name, custo
```

```
mData);
    }
}
```

Now we have a script to receive messages. Next, we need to send a message. Usually, this is for loosely coupled events. For example, in uGUI, we issue events such as [PointerEnter](#) and [PointerExit](#). This is similar to other things that happen to the user's input to the application.

There is a helper class called [ExecuteEvents](#) for sending messages. You can use the [Execute\(\)](#) method of [ExecuteEvents](#) to send messages.

```
// the object to send the message to
GameObject targetObject;
...

// The simplest message sending pattern
ExecuteEvents.Execute<ICustomMessageTarget>(targetObject, null, (target, eventData) =>
{
    target.Message();
});

// Pattern to check if the target can receive a message
if (ExecuteEvents.CanHandleEvent<ICustomMessageTarget>(targetObject))
{
    ExecuteEvents.Execute<ICustomMessageTarget>(targetObject, null, (target, eventData) =>
    {
        target.Message();
    });
}
else
{
    Debug.LogErrorFormat("{0} cannot process ICustomMessageTarget's event", targetObject. name);
}

// Pattern for passing BaseEventData
BaseEventData sendingEventData = new BaseEventData(EventSystem.current);
ExecuteEvents.Execute<ICustomMessageTarget>(targetObject, sendingEventData, (target, eventData) =>
{
```

```

        target.MessageWithEventData(eventData);
    });

// Pattern for passing your own arguments
ExecuteEvents.Execute<ICustomMessageTarget>(targetObject, null, (target, eventData) =>
{
    string customData = "custom data";
    target.MessageWithCustomData(customData);
});

// If the message was not received, the pattern climbs to its parent
ExecuteEvents.ExecuteHierarchy<ICustomMessageTarget>(gameObject, sendingEventData, (target, eventData) =>
{
    target.Message();
});

```

The above code calls the [Message\(\)](#) method on a [GameObject](#) component that implements the [ICustomMessageTarget](#) interface, and the [ExecuteEvents](#) class reference describes the [Execute\(\)](#) method for sending messages to children, parents, etc. The [ExecuteEvents](#) class reference describes the [Execute\(\)](#) method for sending messages to children, parents, etc.

## Advantages and disadvantages of Messaging System

The advantages of the Messaging System are as follows.

- Unlike `SendMessage()`, it can be checked by the interface.
- Unlike `SendMessage()`, multiple arguments can be passed.
- Better performance than `SendMessage()`.
- The message that can be received in the **Inspector's Intercepted Event** is displayed.

On the other hand, the disadvantages of the Messaging System are as follows

- Implementation is somewhat cumbersome.
- It is only a mechanism to send messages to specific `GameObjects`, not a mechanism to broadcast messages to an unspecified number of people.

It is a good idea to use a Messaging System after considering these advantages and disadvantages.

## InputModule

InputModule is a class that is responsible for the main logic of [EventSystem](#).

The process performed by InputModule is as follows.

- Process the input.
- Manage the state of the event.
- Send the event to an object in the scene.

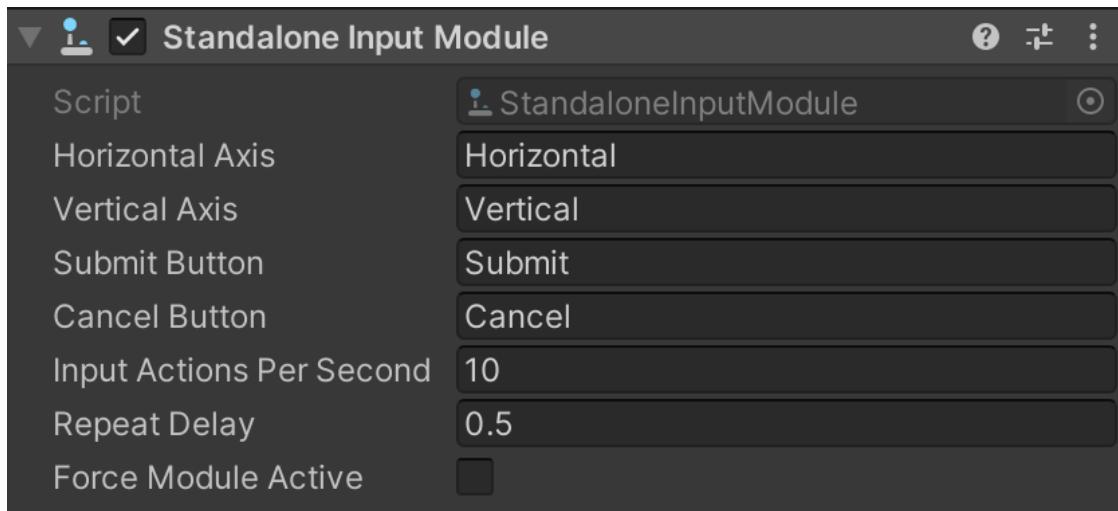
Only one InputModule can be active inside an [EventSystem](#), and the InputModule must be attached to the same [GameObject](#) as the [EventSystem](#) component.

If you want to write your own InputModule, you will need to send events that are supported by Unity's existing UI components. To create your own events, please refer to the *Messaging System* section above.

InputModule constitutes the main logic of the event system, but it can be customized by us. In the past, there were two types of InputModules: [StandaloneInputModule](#) for game controllers/keyboards/mouse and [TouchInputModule](#) for touch panels such as smartphones. Currently, all platforms are supported by [StandaloneInputModule](#), so it is safe to assume that InputModule refers to [StandaloneInputModule](#).

The purpose of InputModule is to map hardware-specific inputs (such as touch, joystick, mouse, motion controller, etc.) to events sent via the message system. The purpose of InputModule is to map hardware-specific inputs (such as touch, joystick, mouse, motion controller, etc.) to events sent via the message system.

## StandaloneInputModule component



```
[AddComponentMenu("Event/Standalone Input Module")]
public class StandaloneInputModule : PointerInputModule
```

[StandaloneInputModule](#) is a component for handling controller and mouse input.

Events such as button presses and drags are sent in response to the input.

[StandaloneInputModule](#) inherits from [PointerInputModule](#), but [PointerInputModule](#) is mainly an [InputModule](#) for mouse input and touch panel. Furthermore, [PointerInputModule](#) inherits from [BaseInputModule](#), which is a basic [InputModule](#).

The [StandaloneInputModule](#) sends pointer events to the component when the mouse or input device is moved. It also uses [GraphicsRaycaster](#) and [PhysicsRaycaster](#) to find out which element is being pointed to by the pointer device.

In addition, [StandaloneInputModule](#) sends Move, Submit, and Cancel events depending on the [Input](#) set in the *Input* window. This works for both keyboard and controller input.

### Items that can be set in StandaloneInputModule

The following items can be set in [StandaloneInputModule](#).

- Vertical and Horizontal axes used for keyboard and controller navigation.
- Submit and Cancel event buttons
- A timeout to determine the maximum number of events sent per second.

## Process flow of StandaloneInputModule

The process flow of [StandaloneInputModule](#) is as follows.

- If the axis set in the **Input** window is input, a Move event is sent to the currently selected object.
- When the Submit or Cancel button is pressed, the Submit or Cancel event is sent to the currently selected object.
- Process mouse input.
  - If it is at the start of a button press
    - Send a [PointerEnter](#) event (to all objects in the hierarchy that can receive it).
    - Send a [PointerPress](#) event.
    - Find an appropriate drag handler (the first element in the hierarchy that can handle dragging).
    - Send the [BeginDrag](#) event to the found drag handler.
    - The object marked as Pressed is the currently selected object of [EventSystem](#).
  - If you are continuing to hold down
    - Process the Move.
    - Send a [DragEvent](#) to the found drag handler.
    - If the touch moves between objects, [PointerEnter](#) and [PointerExit](#) are processed.
  - If they let go...
    - Send a [PointerUp](#) event to the object that received the [PointerPress](#).
    - If the current hover object is an object that received a [PointerPress](#), it will send a [PointerClick](#) event.
    - If there is a found drag handler, send a Drop event.
    - If there was a drag handler, send [EndDrag](#).
  - Handles scroll wheel events.

## Static variables of StandaloneInputModule

### kMouseLeftId

```
public const int kMouseLeftId = -1;
```

Pointer ID representing the left mouse pointer event.

It is used to indicate the default pointer position, not only for left-clicking on the mouse, but also for touch on touch panels.

### kMouseRightId

```
public const int kMouseRightId = -2;
```

Pointer ID representing the mouse right pointer event.

### kMouseMiddleId

```
public const int kMouseMiddleId = -3;
```

Pointer ID representing the middle pointer event of the mouse.

### kFakeTouchesId

```
public const int kFakeTouchesId = -4;
```

It is supposed to be a pointer ID used to simulate touch on devices that do not support touch, but it is not actually used.

## Properties of StandaloneInputModule

### cancelButton

```
public string cancelButton { get; set; }
```

Gets/Sets the character string that represents the Cancel button in [Input](#).

The default value is "Cancel".

Open *Project Settings*, select *Input Manager*, open *Cancel* (there may be more than one), and you can set the keys and buttons corresponding to Cancel on the actual keyboard or controller.

The following is a sample code that detects that the Cancel button has been pressed.

```
using UnityEngine;
using UnityEngine.Events;

public class StandaloneInputModuleCancelSample : MonoBehaviour
{
    void Update()
    {
        var inputModule = EventSystem.current.currentInputModule as StandaloneInputModule;

        if (inputModule != null)
        {
            string cancelButtonString = inputModule.cancelButton;

            if (Input.GetButton(cancelButtonString))
            {
                Debug.LogFormat("cancel button pressed : {0}", cancelButtonString);
            }
        }
    }
}
```

## submitButton

```
public string submitButton { get; set; }
```

Gets/Sets the character string representing the Submit button in [Input](#).

The default value is "Submit".

Open *Project Settings*, select *Input Manager*, open *Submit* (there may be more than one), and you can set the keys and buttons for Submit on the actual keyboard or controller.

The following is a sample code to detect when the Submit button is pressed.

```
using UnityEngine;
using UnityEngine.Events;

public class StandaloneInputModuleSubmitSample : MonoBehaviour
{
    void Update()
    {
        var inputModule = EventSystem.current.currentInputModule as StandaloneInputModule;

        if (inputModule != null)
        {
            string submitButtonString = inputModule.submitButton;

            if (Input.GetButton(submitButtonString))
            {
                Debug.LogFormat("submit button was pressed : {0}", submitButtonString);
            }
        }
    }
}
```

## horizontalAxis

```
public string horizontalAxis { get; set; }
```

Gets/Sets the string that represents the horizontal axis in [Input](#).

The default value is "Horizontal".

Open *Project Settings*, select *Input Manager*, and open *Horizontal* (there may be more than one) to set the keys and buttons that correspond to the horizontal axis on the actual keyboard or controller.

The following is a sample code that detects when a horizontal axis is pressed.

```
using UnityEngine;
using UnityEngine.EventSystems;

public class StandaloneInputModuleHorizontalSample : MonoBehaviour
{
    void Update()
    {
        var inputModule = EventSystem.current.currentInputModule as StandaloneInputModule;
        if (inputModule != null)
        {
            string horizontalAxis = inputModule.horizontalAxis;

            float axisValue = Input.GetAxis(horizontalAxis);

            if (axisValue > 0)
            {
                Debug.LogFormat("Right pressed : {0} {1}", horizontalAxis, axisValue);
            }
            else if (axisValue < 0)
            {
                Debug.LogFormat("Left pressed : {0} {1}", horizontalAxis, axisValue);
            }
        }
    }
}
```

## verticalAxis

```
public string verticalAxis { get; set; }
```

Gets/Sets the string representing the vertical axis in [Input](#).

The default value is "Vertical".

Open *Project Settings*, select *Input Manager*, and open *Vertical* (there may be more than one) to set the keys and buttons that correspond to the vertical axis on a real keyboard or controller.

The following is a sample code that detects when a vertical axis is pressed.

```
using UnityEngine;
using UnityEngine.Events;

public class StandaloneInputModuleVerticalSample : MonoBehaviour
{
    void Update()
    {
        var inputModule = EventSystem.current.currentInputModule as StandaloneInputModule;
        if (inputModule != null)
        {
            string verticalAxis = inputModule.verticalAxis;

            float axisValue = Input.GetAxis(verticalAxis);

            if (axisValue > 0)
            {
                Debug.LogFormat("Top pressed : {0} {1}", verticalAxis, axisValue);
            }
            else if (axisValue < 0)
            {
                Debug.LogFormat("Bottom pressed : {0} {1}", verticalAxis, axisValue);
            }
        }
    }
}
```

## forceModuleActive

---

```
public bool forceModuleActive { get; set; }
```

Force this InputModule to be active.

In fact, this InputModule's [ActivateModule\(\)](#) is called at the timing of [EventSystem's Update\(\)](#).

## repeatDelay

---

```
public float repeatDelay { get; set; }
```

Gets/Sets the threshold number of seconds to send a Move event when the input direction of the keyboard/controller has not changed. If the input direction has not changed, the Move event will not be sent until a time longer than [repeatDelay](#) has passed. The default value is [0.5f](#).

Note that the name sounds like it is for general-purpose input control, but it is not.

## inputActionsPerSecond

---

```
public float inputActionsPerSecond { get; set; }
```

Gets/Sets the threshold number of seconds to send a move event when the input direction of the keyboard/controller is changed. If the input direction is changed, the move event will not be sent until a time longer than [1.0f / inputActionsPerSecond](#) has elapsed. The default value is [10](#).

Note that the name sounds like a generic maximum number of inputs per second, but it is not.

## input

---

```
public BaseInput input { get; }
```

Get the currently used [BaseInput](#).

In a nutshell, [BaseInput](#) is a wrapper for [Input](#). The details of [BaseInput](#) are described later in the [BaseInput](#) section.

This property returns the [BaseInput](#) class that is attached to the same [GameObject](#), but if it does not exist, it adds a component by calling [AddComponent\(\)](#) on the [BaseInput](#).

In Play mode, you can see that the [BaseInput](#) component is attached to the same [GameObject](#) as the [StandaloneInputModule](#).

### inputOverride

---

```
public BaseInput inputOverride { get; set; }
```

Gets/Sets the [BaseInput](#) to override the default [BaseInput](#) of this [InputModule](#).

By using this property, you can replace the input while using the same [InputModule](#). For example, you could create a component that inherits from [BaseInput](#) for automatic debugging, and pass a [StandaloneInputModule](#) as the fake input.

## Public methods of StandaloneInputModule

### ActivateModule

```
public override void ActivateModule();
```

Called when this InputModule becomes active.

The mouse position will be recorded and the currently selected object in [EventSystem](#) will be set.

### DeactivateModule

```
public override void DeactivateModule();
```

Called when this InputModule becomes inactive.

Called at the timing of [EventSystem.OnDisable\(\)](#); the currently selected object of [EventSystem](#) is set to [null](#).

### IsModuleSupported

```
public override bool IsModuleSupported();
```

Returns whether or not this InputModule is supported.

Always returns [true](#).

You can use this method if you have inherited from this class and want to change the support availability depending on the platform. If this method returns [false](#), [EventSystem](#) will not activate the InputModule.

## IsPointerOverGameObject

---

```
public override bool IsPointerOverGameObject(int pointerId);
```

Returns whether a pointer with the ID specified in the argument exists on any [GameObject](#) (or, in other words, whether an object that receives [OnPointerEnter](#) exists).

Called by [EventSystem.IsPointerOverGameObject\(\)](#).

## UpdateModule

---

```
public override void UpdateModule();
```

Called every frame to update the internal state.

This method is called even if it is not the current [InputModule](#) of the [EventSystem](#).

## Process

---

```
public override void Process();
```

Process every frame.

This method is called if it is the current [InputModule](#) of the [EventSystem](#).

The actual process is as follows.

- Call [OnUpdateSelected](#) in the [IUpdateSelectedHandler](#) interface for the selected object.
- Check the status of the mouse buttons.
- Call [OnMove\(\)](#) of the [IMoveHandler](#) interface if necessary.
- If necessary, call [OnSubmit\(\)](#) of the [ISubmitHandler](#) interface.
- If necessary, call [OnCancel\(\)](#) of [ICancelHandler](#) interface.

## ShouldActivateModule

---

```
public override bool ShouldActivateModule();
```

Returns whether or not this InputModule can be activated.

If this method returns **true**, this InputModule will be activated by [EventSystem](#). The conditions for this method to return **true** are as follows.

- [GameObject](#) is active.
- [enabled](#) value of this component is **true**.
- Either [forceModuleActive](#) is set to **true**, or some input has been made.

## BaseInput

[BaseInput](#) is an interface to the input system used by [BaseInputModule](#). In a nutshell, [BaseInput](#) is a wrapper for [Input](#).

By creating a component that inherits from [BaseInput](#), it is possible to replace the input while using the same [InputModule](#). For example, by creating a component that inherits from [BaseInput](#) for automatic debugging and overriding each property and method, you can pass artificial input to the [StandaloneInputModule](#).

## Properties of BaseInput

### compositionString

```
public virtual string compositionString { get; }
```

Get the string composed by the IME currently being input by the user (e.g. Japanese).

In fact, get the [Input.compositionString](#).

### compositionCursorPos

```
public virtual Vector2 compositionCursorPos { get; set; }
```

Gets/Sets the current text input position in the IME window (e.g. Japanese).

Actually, get/set the [Input.compositionCursorPos](#).

### imeCompositionMode

```
public virtual IMECompositionMode imeCompositionMode { get; set; }
```

Gets/Sets whether to allow input by IME (e.g. Japanese).

Actually, get/set the [Input.imeCompositionMode](#).

The default behavior of Unity is to enable IME when an [InputField](#) is selected, but you can leave IME disabled by setting this property to [IMECompositionMode.Off](#).

### mousePresent

```
public virtual bool mousePresent { get; }
```

Gets whether or not the mouse is present.

Actually, it gets [Input.mousePosition](#).

## mousePosition

---

```
public virtual Vector2 mousePosition { get; }
```

Get the mouse position.

The lower left corner of the screen is [\(0, 0\)](#), and the upper right corner is [\(Screen.width, Screen.height\)](#).

[GetMousePositionRelativeToMainDisplayResolution\(\)](#) instead of [Input.mousePosition](#).

[MultipleDisplayUtilities](#) is a utility class of [UnityEngine.UI](#). The value of [Input.mousePosition](#) returns the coordinates in the rendering area, while [MultipleDisplayUtilities](#).

[GetMousePosiToMainDisplayResolution\(\)](#) converts the coordinates to system coordinates and returns them.

## mouseScrollDelta

---

```
public virtual Vector2 mouseScrollDelta { get; }
```

Get the amount of mouse scroll movement.

Actually, it gets [Input.mouseScrollDelta](#).

## touchCount

---

```
public virtual int touchCount { get; }
```

Get the current number of simultaneous taps.

In practice, [Input.touchCount](#) is obtained. Note that [Input.touchCount](#) is guaranteed not to be changed in the current frame.

## touchSupported

---

```
public virtual bool touchSupported { get; }
```

Get whether the current device supports touch.

`Input.touchSupported` checks for support on a narrower scale than the current platform, so if you want to support touch, it is recommended to check this property rather than checking the system platform. property rather than checking the system platform.

## BaseInput public methods

### GetAxisRaw

```
public virtual float GetAxisRaw(string axisName);
```

Get the value of the axis in the state where it is not passed through the smoothing filter.

GetAxisRaw(), which actually returns Input.

Note that BaseInput does not have a GetAxis() method.

### GetButtonDown

```
public virtual bool GetButtonDown(string buttonName);
```

Returns whether the button is pressed or not.

GetButtonDown() is actually returned.

### GetMouseButton

```
public virtual bool GetMouseButton(int button);
```

Returns whether or not the mouse button is being pressed.

GetMouseButtonUp() is actually returned.

### GetMouseButtonDown

```
public virtual bool GetMouseButtonDown(int button);
```

Returns whether the mouse button has been pressed (from the released state) or not.

[GetMouseButton\(\)](#) is actually returned.

## GetMouseButtonUp

---

```
public virtual bool GetMouseButtonUp(int button);
```

Returns whether or not the mouse button has been released.

[GetMouseButtonUp\(\)](#) is actually returned.

## GetTouch

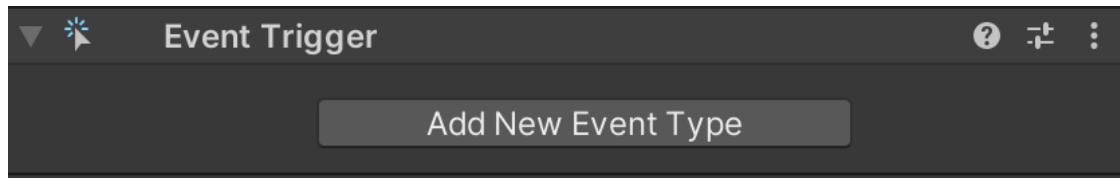
---

```
public virtual Touch GetTouch(int index)
```

Get the current touch status.

In the case of multi-touch, [index](#) should be 1 or more. [GetTouch\(\)](#) is actually returned.

## Event Trigger component



[EventTrigger](#) is a component used when you want to receive an arbitrary event from [EventSystem](#) and call a method corresponding to the event.

For example, the [Image](#) component does not implement [IPointerEnterHandler](#), so [OnPointerEnter\(\)](#) will not be called when the mouse cursor hovers over an image. It would be too much to create a new component that inherits from the [Image](#) component just to implement the processing when the mouse cursor hovers over an image. In such a case, this [EventTrigger](#) is useful.

If you want to perform some kind of processing when the mouse cursor hovers over an [Image](#), the procedure is as follows.

1. Attach the [EventTrigger](#) component to the same [GameObject](#) as the [Image](#) component.
2. Click the *Add New Event Type* button in [EventTrigger](#) and select *PointerEnter*.
3. Make sure that the callback corresponding to *PointerEnter* can now be set. This callback has one argument of type [BaseEventData](#). Note that the type of the argument is [BaseEventData](#), not [PointerEventData](#).
4. Create a [TriggeredOnPointerEnter](#) script as shown below, and create a method to receive [BaseEventData](#).

```
using UnityEngine;
using UnityEngine.Events;

public class TriggeredOnPointerEnter : MonoBehaviour
{
    public void MyOnPointerEnter(BaseEventData baseEventData)
    {
        Debug.Log("Mouse cursor overlapped");
    }
}
```

5. Attach the above [TriggeredOnPointerEnter](#) script to the same [GameObject](#) as the [Image](#) component.

6. Drag your `GameObject` to `PointerEnter` in `EventTrigger` and specify `MyOnPointerEnter()` as the callback.

The above will output a log when the mouse cursor hovers over this [image](#).

In this case, we set the callback from the **Inspector** of `EventTrigger`, but if you want to set it from a script, you can do so as follows

```
using UnityEngine;
using UnityEngine.Events;

[RequireComponent(typeof(EventTrigger))]
public class TriggereadOnPointerEnterFromScript : MonoBehaviour
{
    public void Start()
    {
        var trigger = GetComponent<EventTrigger>();

        // Item to register with EventTrigger
        var entry = new EventTrigger.Entry();
        entry.eventID = EventTriggerType.PointerEnter;
        entry.callback.AddListener(MyOnPointerEnter);

        // register with EventTrigger.
        trigger.triggers.Add(entry);
    }

    public void MyOnPointerEnter(BaseEventData baseEventData)
    {
        Debug.Log("Mouse cursor overlapped");
    }
}
```

There used to be a bug that prevented you from setting the proper callback from the **Inspector** in `EventTrigger`, but this has been fixed in Unity 2018.4, Unity 2019.2, Unity 2020.1 and later.

[https://issuetracker.unity3d.com/issues/events-generated-by-the-player-input-component-do-not-have-callbackcontext-set-as-their-parameter-type?\\_ga=2.42604581.127594340.1571029466-241225422.1516255303](https://issuetracker.unity3d.com/issues/events-generated-by-the-player-input-component-do-not-have-callbackcontext-set-as-their-parameter-type?_ga=2.42604581.127594340.1571029466-241225422.1516255303)

If you inadvertently encounter this bug, try upgrading your Unity version or setting the callback from a script.

Note that once an event has been captured by an [EventTrigger](#), it will not propagate to its parent any further.

## Types of events that can be specified in EventTrigger

The event types that can be specified in `EventTrigger` are defined as `EventTriggerType`.

```
namespace UnityEngine.EventSystems
{
    public enum EventTriggerType
    {
        /// <summary>
        /// Capture IPointerEnterHandler.Enter
        /// </summary>
        PointerEnter = 0,

        /// <summary>
        /// Capture IPointerExitHandler.OnPointerExit.
        /// </summary>
        PointerExit = 1,

        /// <summary>
        /// Capture IPointerDownHandler.OnPointerDown.
        /// </summary>
        PointerDown = 2,

        /// <summary>
        /// Capture IPointerUpHandler.OnPointerUp.
        /// </summary>
        PointerUp = 3,

        /// <summary>
        /// Capture IPointerClickHandler.OnPointerClick.
        /// </summary>
        PointerClick = 4,

        /// <summary>
        /// Capture IDragHandler.OnDrag.
        /// </summary>
        Drag = 5,

        /// <summary>
        /// Capture IDropHandler.OnDrop.
    }
}
```

```
/// </summary>
Drop = 6,  
  
/// <summary>
/// Capture IScrollHandler.OnScroll.
/// </summary>
Scroll = 7,  
  
/// <summary>
/// Capture IUpdateSelectedHandler.OnUpdateSelected
/// </summary>
UpdateSelected = 8,  
  
/// <summary>
/// Capture ISelectHandler.OnSelect.
/// </summary>
Select = 9,  
  
/// <summary>
/// Capture IDeselectHandler.OnDeselect.
/// </summary>
Deselect = 10,  
  
/// <summary>
/// Capture IMoveHandler.OnMove.
/// </summary>
Move = 11,  
  
/// <summary>
/// Capture IInitializePotentialDrag.InitializePotentialDrag.
/// </summary>
InitializePotentialDrag = 12,  
  
/// <summary>
/// Capture IBeginDragHandler.OnBeginDrag.
/// </summary>
BeginDrag = 13,  
  
/// <summary>
/// Captures IEndDragHandler.OnEndDrag.
/// </summary>
EndDrag = 14,
```

```
/// <summary>
/// Capture ISubmitHandler.Submit.
/// </summary>
Submit = 15,
```

```
/// <summary>
/// Capture ICancelHandler.OnCancel
/// </summary>
Cancel = 16
}
```

Regardless of the [EventTriggerType](#), the argument of the callback set to [EventTrigger](#) is of type [BaseEventData](#).

## Raycaster

Raycaster is used to find out where the pointers are overlapping.

It is common for InputModule to use a Raycaster set up in the scene to check for pointer overlap.

There are three Raycasters by default.

- [GraphicRaycaster](#) : Used for UI
- [PhysicsRaycaster](#) : Used for 3D physics.
- [Physics2DRaycaster](#) : used for 2D physics

Both of these inherit from [BaseRaycaster](#).

```
public abstract class BaseRaycaster : UIBehaviour

[AddComponentMenu("Event/Graphic Raycaster")]
[RequireComponent(typeof(Canvas))]
public class GraphicRaycaster : BaseRaycaster

[AddComponentMenu("Event/Physics Raycaster")]
[RequireComponent(typeof(Camera))]
public class PhysicsRaycaster : BaseRaycaster

[AddComponentMenu("Event/Physics 2D Raycaster")]
[RequireComponent(typeof(Camera))]
public class Physics2DRaycaster : PhysicsRaycaster
```

### Note

[EventSystems](#) namespace for the [BaseRaycaster](#), [Physics2DRaycaster](#), and [PhysicsRaycaster](#), while the namespace for the [GraphicRaycaster](#) is [UnityEngine.UI](#).

If a [Physics2DRaycaster](#) or [PhysicsRaycaster](#) is set up in the scene, it is easy for a non-UI element to receive messages from the InputModule, simply by attaching a script that implements the event interface. An example of how to use the [PhysicsRaycaster](#) is shown below.

1. Attach a [PhysicsRaycaster](#) to the *Main Camera*.
2. Create a Cube using the Editor menu *GameObject -> 3D Object -> Cube*.
3. Adjust the position of the Cube so that it appears in the *Main Camera*.

4. Create the following script so that the Cube can receive the `OnPointerClick()` callback.

```
using UnityEngine;
using UnityEngine.EventSystems;

public class InterClickHandlerComponent : MonoBehaviour, IPointerClickHandler
{
    public void OnPointerClick(PointerEventData eventData)
    {
        Debug.Log("Clicked");
    }
}
```

5. Attach the created `PointerClickHandlerComponent` to the Cube's `GameObject`.
6. Start Play mode, and make sure you see the log output when you click on the Cube in the **Game View**.
7. To test this, disable `BoxCollider` in Cube, and make sure that no logs are output when you click on Cube.

The `EventSystem` needs to figure out where to send the current input event, which is done by the Raycaster. Given a position in screen space, Raycaster will collect all potential targets, check if they are under the given position, and return the object closest to the screen.

If a Raycaster is valid in the scene, it will be used by the `EventSystem` when it is queried by the Input Module.

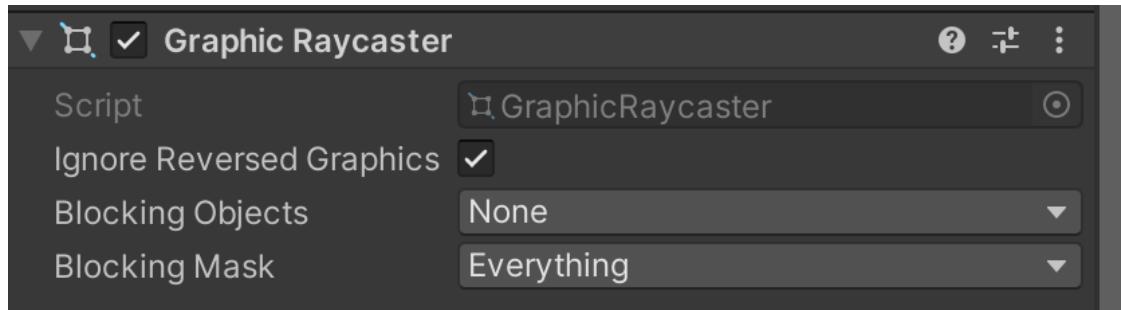
If more than one Raycaster is used, they will all be raycasted and the results will be sorted according to the distance to the element

In uGUI, the `GraphicRaycaster` component is basically used to handle input events.

Prior to Unity 5.4, every active Canvas to which the `GraphicRaycaster` was attached would throw a raycast every frame to check the pointer position. This happened regardless of the platform, and even iOS and Android devices without a mouse would look for the mouse position. This was a pure waste of CPU time, consuming more than 5% of the CPU frame time of the Unity application. This problem was fixed in Unity 5.4, and since then, devices without a mouse will not look for the mouse position and will not perform unnecessary raycasts.

`GraphicRaycaster` is a relatively naive implementation that traverses all `Graphic` components that have the `raycastTarget` property set to `true`. For each Raycast Target, the Raycaster will perform multiple tests. If the Raycast Target passes all the tests, it is added to the hit list.

## GraphicRaycaster



```
[AddComponentMenu("Event/Graphic Raycaster")]
[RequireComponent(typeof(Canvas))]
public class GraphicRaycaster : BaseRaycaster
```

**GraphicRaycaster** is a component that is used to raycast to the [Canvas](#).

The [GraphicRaycaster](#) checks all [Graphic](#) components under the [Canvas](#) to see if any of them have been hit at a particular position.

In addition to [Canvas](#) and [CanvasScaler](#), [GraphicRaycaster](#) is attached to the [GameObject](#) created by going to *GameObject -> UI -> Canvas* from the Editor.

The [GraphicRaycaster](#) can be configured to ignore the [Graphic](#) in the back or front, or to ignore 2D/3D objects. You can also manually set the priority if you want to process certain elements before or after the raycast.

## GraphicRaycaster Properties

### eventCamera

```
public override Camera eventCamera { get; }
```

Get the camera that generates the ray for this Raycaster.

This property returns different values depending on the settings of the [Canvas](#) attached to the same [GameObject](#). This property returns the result of evaluation in the following order.

1. If the [renderMode](#) of [Canvas](#) is [ScreenSpaceOverlay](#), return [null](#).
2. If the [Canvas renderMode](#) is [ScreenSpaceCamera](#) and the [Canvas worldCamera](#) is [null](#), return [null](#).
3. Returns the value of the [worldCamera](#) of the [Canvas](#) attached to the [GameObject](#) if it is not [null](#).
4. Return [Camera.main](#).

This property can be used to get the position in camera space with [Camera.ScreenToViewportPoint\(\)](#) or as [Graphic.Raycast\(\)](#).

### renderOrderPriority

```
public virtual int renderOrderPriority { get; }
```

Gets the priority level when multiple Raycaster exist.

This value is basically the [renderOrder](#) of the [Canvas](#).

If the [renderMode](#) of the [Canvas](#) attached to the same [GameObject](#) is [ScreenSpaceOverlay](#), it returns the [renderOrder](#) of the [rootCanvas](#) of the [Canvas](#), otherwise it returns [int.MinValue](#).

### rootRaycaster

```
public BaseRaycaster rootRaycaster { get; }
```

Get the [BaseRaycaster](#) at the top of this hierarchy.

This property was introduced in Unity 2019.1 for a Bug Fix.

### sortOrderPriority

---

```
public virtual int sortOrderPriority { get; }
```

Gets the priority according to the [sortingOrder](#) of [Canvas](#).

The higher the value, the higher the priority; returns the [Canvas](#)'s [sortingOrder](#) if the [Canvas](#)'s [renderMode](#) is [ScreenSpaceOverlay](#), otherwise returns [int.MinValue](#).

### blockingObjects

---

```
public GraphicRaycaster.BlockingObjects blockingObjects { get; set; }
```

Gets/Sets whether the 2D or 3D object blocks the [GraphicRaycaster](#).

The default value is [BlockingObjects.None](#), which means that no 2D or 3D objects will block the [GraphicRaycaster](#).

This property is valid only if the [renderMode](#) of the [Canvas](#) is other than [ScreenSpaceOverlay](#). This property is only valid if the [renderMode](#) of the [Canvas](#) is other than [ScreenSpaceOverlay](#), because if it is [ScreenSpaceOverlay](#), the UI will be displayed on the front page and will not be blocked by 2D/3D objects.

The definition of [BlockingObjects](#) is as follows.

```
/// <summary>
/// Type of Raycaster used to check for elements blocking the Canvas
/// </summary>
public enum BlockingObjects
{
    /// <summary>
    /// Does not raycast anything.
    /// </summary>
```

```

None = 0,

/// Do a 2D raycast.

TwoD = 1,

/// Perform a 3D raycast.

ThreeD = 2,

/// Performs 2D and 3D raycasts.

All = 3,
}

```

`RaycastAll()` will be called if 2D raycasting is enabled, or `Physics.RaycastAll()` if 3D raycasting is enabled, and objects hit by those raycasts will block raycast hits in the `GraphicRaycaster`. in the `GraphicRaycaster`.

The following are the steps to understand the behavior of `blockingObjects`.

1. Create a Cube using `GameObject -> 3D Object -> Cube`.
2. Adjust the Cube so that it appears in the **Main Camera**.
3. Create a Button in `GameObject -> UI -> Button`.
4. Adjust the Button so that it overlaps the Cube by half.
5. Put it in Play mode.
6. Make sure that the Button reacts when you click on the part of the Button hidden by the Cube.
7. Set the **Blocking Objects** of the `GraphicRaycaster` attached to the `Canvas` to *Three D*.
8. Make sure that the Button does not react when you click on the part of the Button hidden by the Cube.

As you can see, the setting of `blockingObjects` causes the `GraphicRaycaster`'s raycast hits to be blocked by 3D objects.

## blockingMask

```
public LayerMask blockingMask { get; set; }
```

Gets/Sets the [LayerMask](#) of the object that blocks the [GraphicRaycaster](#) if [blockingObjects](#) is not [None](#).

By default, all [LayerMask](#) objects will block the [GraphicRaycaster](#).

Note

This property used to be protected, but has been made public since Unity 2019.2.

The following are the steps to understand the behavior of [blockingMask](#).

1. Create a Cube using *GameObject -> 3D Object -> Cube*.
2. Adjust the Cube so that it appears in the *Main Camera*.
3. Create a Button in *GameObject -> UI -> Button*.
4. Adjust the Button so that it overlaps the Cube by half.
5. Set the **Blocking Objects** of the [GraphicRaycaster](#) attached to the [Canvas](#) to *Three D*.
6. Set the **Layer** of the Cube to **Water**
7. Put it in Play mode.
8. Make sure that the Button does not react when you click on the part of the Button hidden by the Cube.
9. Uncheck **Water** in [GraphicRaycaster's Blocking Mask](#).
10. Make sure that the Button reacts when you click on the part of the Button hidden by the Cube.

Thus, by setting the [blockingMask](#), you can set the **Layer** of the object that is the target of the [blockingObjects](#).

## ignoreReversedGraphics

```
public bool ignoreReversedGraphics { get; set; }
```

Gets/Sets whether UI elements that are turned inside out are ignored by raycast.

The default value is [true](#), so UI elements that are turned inside out will not be raycast hit.

To determine whether or not it is facing backward, the inner product of the front direction (of [eventCamera](#)) and the orientation of the UI element is used.

You may think that this property setting is irrelevant if the UI is not in world space, but it is not. For example, if the [Button's rotation](#) is 180 degrees even in screen space, it will not respond to touch by default. In such a case, if you want the button to respond to touch, you need to set this property to [false](#).

## Public methods of GraphicRaycaster

### Raycast

```
public override void Raycast(PointerEventData eventData, List<RaycastResult> resultAppendList);
```

It checks whether or not the [Graphic](#) under [Canvas](#) exists at the position of the pointer passed in [eventData](#), and stores the result in [resultAppendList](#).

This method is called every frame from [EventSystem.Update\(\)](#) through [PointerInputModule](#) (which is the parent class of [StandaloneInputModule](#)).

The flow of the process performed by [Raycast\(\)](#) is as follows.

1. If the pointer position is outside of the [Canvas](#), do nothing.
2. [RaycastAll\(\)](#) and [Physics.RaycastAll\(\)](#) are called to check if 2D / 3D objects block this raycast, depending on the setting of [blockingObjects](#). [RaycastAll\(\)](#) and [Physics.RaycastAll\(\)](#) will be called, and [Graphics](#) that are farther away from the hit will be ignored.
3. Do the following for all [Graphics](#) under [Canvas](#).
  - If the [raycastTarget](#) of [Graphic](#) is [false](#), skip the rest of the process.
  - If the [cull](#) of the [Graphic](#)'s [canvasRenderer](#) is [false](#), skip the rest of the process.
  - If the [depth](#) of [Graphic](#) is [-1](#), skip the rest of the process.
  - If the pointer position is outside the [RectTransform](#) rectangle, skip the rest of the process.
  - If [eventCamera](#) is not [null](#), skip further processing if [Graphic](#) is far from [farClipPlane](#).
  - Call [Raycast\(\)](#) on the [Graphic](#), and if a hit is found, add the [Graphic](#) to the list.
4. Sorts the list of hit [Graphics](#) by depth.
5. Do the following for each hit [Graphic](#).
  - If [ignoreReversedGraphics](#) is [true](#), skip the back-facing [Graphic](#).
  - If [Graphic](#) is behind the camera, skip it.
  - If [blockingObjects](#) is not [None](#), skip if the distance to the [Graphic](#) is farther than the distance from the camera to the 2D/3D object.
  - Store the information of the [Graphic](#) object in [RaycastResult](#) and add it to [resultAppendList](#).

## PhysicsRaycaster component

The **PhysicsRaycaster** uses [Physics.RaycastAll\(\)](#) or [Physics.RaycastNonAlloc\(\)](#) to determine the collision between the pointer and the **Collider** (within the camera's view).

When the **PhysicsRaycaster** is attached to a **GameObject** to which the **Camera** is attached, the collision between the pointer and the **Collider** is judged, and if necessary, a callback is called on the object to which the **Collider** is attached.

## Properties of PhysicsRaycaster

### depth

```
public virtual int depth { get; }
```

Get the depth to determine the priority of the event... but it's not actually used.

Returns `eventCamera.depth` if `eventCamera` is not `null`, returns `0xFFFFFFF` if `eventCamera` is `null`.

### eventCamera

```
public override Camera eventCamera { get; }
```

Get the camera that generates the ray for this Raycaster.

If `Camera` is attached to the same `GameObject`, return it; otherwise, return `Camera.main`.

### eventMask

```
public LayerMask eventMask { get; set; }
```

Gets/Sets the `LayerMask` for filtering the raycast target.

The default value is `-1`, i.e. `Everything`, and all Layers are subject to raycasting.

`RaycastAll()` and `Physics.RaycastNonAlloc()` will be the values obtained from `finalEventMask`.

### finalEventMask

```
public int finalEventMask { get; }
```

Get the value of the [LayerMask](#) that will be passed to [Physics.RaycastAll\(\)](#) and [Physics.RaycastNonAlloc\(\)](#).

If [eventCamera](#) is not [null](#), then the logical product of [eventCamera.cullingMask](#) and [eventMask](#) is returned. If [eventCamera](#) is [null](#), then -1 i.e. [Everything](#) is returned.

### maxRayIntersections

---

```
public int maxRayIntersections { get; set; }
```

Gets/Sets the maximum number of Ray hits to be detected.

The default value is [0](#).

[Physics.RaycastAll\(\)](#) is used to detect raycast hits with no limit on the number of hits. If the value is greater than [0](#), [Physics.RaycastNonAlloc\(\)](#) is used to detect raycast hits with a limit on the maximum number of hits.

[Physics.RaycastAll\(\)](#) is called by default, so GC Alloc is generated for every raycast decision. If you are concerned about performance, it would be better to set the maximum number of hits according to your game design.

## Public methods of PhysicsRaycaster

### Raycast

```
public override void Raycast(PointerEventData eventData, List<RaycastResult> resultAppendList);
```

It checks if the [Collider](#) exists at the pointer position passed in [eventData](#), and stores the result in [resultAppendList](#).

This method is called every frame from [EventSystem.Update\(\)](#) through [PointerInputModule](#) (which is the parent class of [StandaloneInputModule](#)).

The flow of the process performed by [Raycast\(\)](#) is as follows.

1. If [eventCamera](#) is [null](#), do nothing.
2. If the pointer position is outside the [pixelRect](#) of [eventCamera](#), do nothing.
3. Create a [Ray](#) by passing the pointer position to [eventCamera.ScreenPointToRay\(\)](#).
4. If [maxRayIntersections](#) is [0](#), call [Physics.RaycastAll\(\)](#) to find the [Collider](#) that hits the [Ray](#).  
If [maxRayIntersections](#) is non-zero, call [Physics.RaycastNonAlloc\(\)](#) and detects the [Collider](#) that hits the [Ray](#).
5. Sort the hit [Colliders](#) and add them to the [resultAppendList](#).

## Physics2DRaycaster component

```
[AddComponentMenu("Event/Physics 2D Raycaster")]
[RequireComponent(typeof(Camera))]
public class Physics2DRaycaster : PhysicsRaycaster
```

The [Physics2DRaycaster](#) uses [Physics2D.RaycastAll\(\)](#) or [Physics2D.RaycastNonAlloc\(\)](#) to determine the collision between the pointer and the [Collider](#) (within the area captured by the camera).

When the [Physics2DRaycaster](#) is attached to a [GameObject](#) to which the [Camera](#) is attached, the collision between the pointer and the [Collider](#) is judged, and if necessary, a callback is called on the object to which the [Collider](#) is attached.

[Physics2DRaycaster](#) inherits from [PhysicsRaycaster](#) and shares the same functions as [PhysicsRaycaster](#) except for the [Raycast\(\)](#) method. Therefore, please refer to [PhysicsRaycaster](#) for methods and properties other than [Raycast\(\)](#).

## Public methods of Physics2DRaycaster

```
public override void Raycast(PointerEventData eventData, List<RaycastResult> resultAppendList);
```

It checks whether `Collider2D` exists at the pointer position passed in `eventData`, and stores the result in `resultAppendList`.

This method is called every frame from `EventSystem.Update()` through `PointerInputModule` (which is the parent class of `StandaloneInputModule`).

The flow of the process performed by `Raycast()` is as follows.

1. If `eventCamera` is `null`, do nothing.
2. If the pointer position is outside the `pixelRect` of `eventCamera`, do nothing.
3. Create a `Ray` by passing the pointer position to `eventCamera.ScreenPointToRay()`.
4. If `maxRayIntersections` is `0`, `Physics2D.GetRayIntersectionAll()` is called to find the `Collider2D` that hits the `Ray`. If `maxRayIntersections` is non-zero, `Physics2D.GetRayIntersectionNonAlloc()` is called to find the `Collider2D` that hits the `Ray`.
5. Sort the `Collider2D` hits and add them to the `resultAppendList`.

### Note

`Physics2D.GetRayIntersectionAll()` and `Physics2D.GetRayIntersectionNonAlloc()` are called instead of `Physics2D.RaycastAll()` and `Physics2D.Get.RaycastNonAlloc()`. But in both cases, `Physics2D.OverlapPointAll()` and `Physics2D.OverlapPointNonAlloc()` will eventually be called.

## Extending StandaloneInputModule

By extending the [StandaloneInputModule](#)

1. Prevent continuous touch/click within a certain period of time.
2. Provide a flag to disable touch/click all together.
3. Disable taps in SafeArea.
4. Improve the Click judgment in the Scroll View.

This can be a useful feature.

However, the [StandaloneInputModule](#) component is not designed to be extended. Therefore, we need to create a class that is a copy of [StandaloneInputModule](#) and modify that class.

```
using System;
using UnityEngine;
using UnityEngine.Serialization;

namespace UnityEngine.EventSystems
{
    [AddComponentMenu("Event/Custom Standalone Input Module")]
    /// <summary>
    /// Custom Input Module to handle mouse/keyboard/controller input
    /// </summary>
    public class CustomStandaloneInputModule : PointerInputModule
    {
        // Here is a copy of the StandaloneInputModule
        ...
    }
}
```

Prevent continuous touch/click within a certain period of time.

It is often the case that you want to prevent continuous tapping within a certain period of time in order to avoid erroneous operation or problems. This can be prevented on the UI element side, but it is more reliable to cancel the tap event itself on the InputModule side.

Specifically, in the [StandaloneInputModule](#), the [ProcessTouchPress\(\)](#) method, which is responsible for touch processing of the touch panel, and [ProcessMousePress\(\)](#), which is responsible for mouse click processing, should be changed to disable touch/click within a certain period of time. In the [StandaloneInputModule](#).

Prepare a flag to disable touch/click at once.

This can also be done by [ProcessTouchPress\(\)](#) and [ProcessMousePress\(\)](#) above to disable touch/click depending on the flag.

Disable taps in a SafeArea

In iOS, if a touch/click occurs in a SafeArea, there is a possibility that it will not be reviewed by the AppStore. It is quite tricky to handle this in each UI element, but it is easier to handle it in the InputModule side.

Improving the Click judgment in ScrollView

In the default state, when the [Button](#) is placed in the Scroll View, the dragging judgment of the scroll takes precedence, and the [Button](#)'s clicking judgment becomes severe. To solve this problem, there are several solutions.

1. Increase the value of [pixelDragThreshold](#).

If the distance between the previous position of the mouse or finger and the current position is less than the value of [EventSystem.current.pixelDragThreshold](#), the drag is not considered to have started. The default value of [EventSystem.current.pixelDragThreshold](#) is 10, but increasing this value will make the dragging judgment more severe, resulting in the [Button](#) being judged as clicked.

```
EventSystem.current.pixelDragThreshold = 20;
```

However, be aware that this setting affects the entire current [EventSystem](#).

2. Customize [StandaloneInputModule](#) so that click judgment is also performed when dragging is finished.

You can use the [ReleaseMouse\(\)](#) method to do the appropriate processing. However, it is better to decide whether or not to perform the click judgment based on the dragging distance, since always performing the click judgment at the end of dragging may increase the number of false clicks.

## Create the CustomStandaloneInputModule

An extension of [StandaloneInputModule](#) that implements the functions described so far would be the following source code.

```
//#define DebugCustomStandaloneInputModule // For outputting debug logs

using System;
using UnityEngine;
using UnityEngine.Serialization;

namespace UnityEngine.EventSystems
{
    [AddComponentMenu("Event/Custom Standalone Input Module")]
    /// <summary>
    /// Unique Input Module to handle mouse/keyboard/controller input
    /// </summary>
    public class CustomStandaloneInputModule : PointerInputModule
    {
        private float m_PrevActionTime;
        private Vector2 m_LastMoveVector;
        private int m_ConsecutiveMoveCount = 0;

        private Vector2 m_LastMousePosition;
        private Vector2 m_MousePosition;

        private GameObject m_CurrentFocusedGameObject;

        private PointerEventData m_InputPointerEvent;

        /// <summary>
        /// Effective time for double-click judgment (seconds)
        /// </summary>
        public float multiClickPeriod = 0.3f;

        /// <summary>
        /// The time (in seconds) between the last Pointer Down and when Pointer Down is enabled again.
        /// Can be used as a countermeasure against so-called continuous hits.
        /// </summary>
```

```

public float validPressInterval = 0.0f;

/// <summary>
/// The distance (in pixels) at which dragging enables the click detection. Depends on the screen resolution.
/// </summary>
public float dragThreshold = 50f;

/// <summary>
/// It is possible to specify whether or not to ignore the push event processing.
/// </summary>
public bool ignorePressEvent { get; set; } = false;

/// <summary>
/// Time of last Pointer Down
/// </summary>
private float lastPointerDownTime = 0;

protected CustomStandaloneInputModule()
{
}

protected override void Awake()
{
    base.Awake();
}

/// <summary>
/// String representing the horizontal axis.
/// </summary>
[SerializeField]
private string m_HorizontalAxis = "Horizontal";

/// <summary>
/// String representing the vertical axis.
/// </summary>
[SerializeField]
private string m_VerticalAxis = "Vertical";

/// <summary>
/// A string representing the Submit button.

```

```

/// </summary>
[SerializeField]
private string m_SubmitButton = "Submit";

/// <summary>
/// String representing the Cancel button
/// </summary>
[SerializeField]
private string m_CancelButton = "Cancel";

/// <summary>
/// The threshold number of seconds to send a Move event when the input direction of the keyboard/controller changes.
/// </summary>
[SerializeField]
private float m_InputActionsPerSecond = 10;

/// <summary>
/// The threshold number of seconds to send a Move event if the keyboard/controller input direction has not changed.
/// </summary>
[SerializeField]
private float m_RepeatDelay = 0.5f;

[SerializeField]
[FormerlySerializedAs("m_AllowActivationOnMobileDevice")]
private bool m_ForceModuleActive;

/// <summary>
/// Whether to force this InputModule to be active or not.
/// </summary>
/// <remarks>
/// If there was no module with a higher priority than this, it will be forced to become active even if you try to deactivate it.
/// </remarks>
public bool forceModuleActive
{
    get { return m_ForceModuleActive; }
    set { m_ForceModuleActive = value; }
}

/// <summary>

```

```

    /// The threshold number of seconds to send a Move event when the input direction of the keyboard/controller changes.
    /// </summary>
    public float inputActionsPerSecond
    {
        get { return m_InputActionsPerSecond; }
        set { m_InputActionsPerSecond = value; }
    }

    /// <summary>
    /// The threshold number of seconds to send a Move event when the input direction of the keyboard/controller changes.
    /// </summary>
    /// <remarks>
    /// If the same direction is maintained, the inputActionsPerSecond property can be used to control the event firing rate. However, it is preferable to delay the first repetition so that the user does not inadvertently receive the same action.
    /// </remarks>
    public float repeatDelay
    {
        get { return m_RepeatDelay; }
        set { m_RepeatDelay = value; }
    }

    /// <summary>
    /// String representing the horizontal axis.
    /// </summary>
    public string horizontalAxis
    {
        get { return m_HorizontalAxis; }
        set { m_HorizontalAxis = value; }
    }

    /// <summary>
    /// String representing the vertical axis.
    /// </summary>
    public string verticalAxis
    {
        get { return m_VerticalAxis; }
        set { m_VerticalAxis = value; }
    }

```

```

/// <summary>
/// A string representing the Submit button.
/// </summary>
public string submitButton
{
    get { return m_SubmitButton; }
    set { m_SubmitButton = value; }
}

/// <summary>
/// String representing the Cancel button
/// </summary>
public string cancelButton
{
    get { return m_CancelButton; }
    set { m_CancelButton = value; }
}

private bool ShouldIgnoreEventsOnNoFocus()
{
#if UNITY_EDITOR
    return !UnityEditor.EditorApplication.isRemoteConnected;
#else
    return true;
#endif
}

public override void UpdateModule()
{
    if (!eventSystem.isFocused && ShouldIgnoreEventsOnNoFocus())
    {
        if (m_InputPointerEvent != null && m_InputPointerEvent.pointerDrag != null && m_
InputPointerEvent.dragging)
        {
            ReleaseMouse(m_InputPointerEvent, m_InputPointerEvent.pointerCurrentRaycast.g
ameObject);
        }

        m_InputPointerEvent = null;
    }

    return;
}

```

```

        m_LastMousePosition = m_MousePosition;
        m_MousePosition = input.mousePosition;
    }

public override bool IsModuleSupported()
{
    return m_ForceModuleActive || input.mousePresent || input.touchSupported;
}

public override bool ShouldActivateModule()
{
    if (!base.ShouldActivateModule())
        return false;

    var shouldActivate = m_ForceModuleActive;
    shouldActivate |= input.GetButtonDown(m_SubmitButton);
    shouldActivate |= input.GetButtonDown(m_CancelButton);
    shouldActivate |= !Mathf.Approximately(input.GetAxisRaw(m_HorizontalAxis), 0.0f);
    shouldActivate |= !Mathf.Approximately(input.GetAxisRaw(m_VerticalAxis), 0.0f);
    shouldActivate |= (m_MousePosition - m_LastMousePosition).sqrMagnitude > 0.0f;
    shouldActivate |= input.GetMouseButtonDown(0);

    if (input.touchCount > 0)
        shouldActivate = true;

    return shouldActivate;
}

/// <summary>
/// Called when this InputModule becomes active.
/// </summary>
public override void ActivateModule()
{
    if (!eventSystem.isFocused && ShouldIgnoreEventsOnNoFocus())
        return;

    base.ActivateModule();
    m_MousePosition = input.mousePosition;
    m_LastMousePosition = input.mousePosition;

    var toSelect = eventSystem.currentSelectedGameObject;
}

```

```

if (toSelect == null)
    toSelect = eventSystem.firstSelectedGameObject;

    eventSystem.SetSelectedGameObject(toSelect, GetBaseEventData());
}

/// <summary>
/// Called when this InputModule becomes inactive.
/// </summary>
public override void DeactivateModule()
{
    base.DeactivateModule();
    ClearSelection();
}

public override void Process()
{
    if (!eventSystem.isFocused && ShouldIgnoreEventsOnNoFocus())
        return;

    bool usedEvent = SendUpdateEventToSelectedObject();

    // case 1004066
    // If the navigation event changes the currently selected GameObject, and the Submit button
    // is a touch/mouse button
    // Touch/mouse events should be processed before navigation events.

    // Need to prioritize touch for the mouse emulation layer.
    if (!ProcessTouchEvent() && input.mousePresent)
        ProcessMouseEvent();

    if (eventSystem.sendNavigationEvents)
    {
        if (!usedEvent)
            usedEvent |= SendMoveEventToSelectedObject();

        if (!usedEvent)
            SendSubmitEventToSelectedObject();
    }
}

private bool ProcessTouchEvent()

```

```

{
    for (int i = 0; i < input.touchCount; ++i)
    {
        Touch touch = input.GetTouch(i);

        if (touch.type == TouchType.Indirect)
            continue;

        bool released;
        bool pressed;
        var pointer = GetTouchPointerEventData(touch, out pressed, out released);

        ProcessTouchPress(pointer, pressed, released);

        if (!released)
        {
            ProcessMove(pointer);
            ProcessDrag(pointer);
        }
        else
            RemovePointerData(pointer);
    }

    return input.touchCount > 0;
}

/// <summary>
/// This method is called by Unity when handling touch events. If you want to implement your own way of handling touch events, you should override this method.
/// </summary>
/// <param name="pointerEvent">Event data for this touch event, such as position and ID, sent to the target object of the touch event.</param>
/// <param name="pressed">If it is the first frame of the touch event, true will be passed, and if it is a later frame, false will be passed. Therefore, it can be used to determine if the touch event occurred at a certain moment or not.</param>
/// <param name="released">If it is the last frame of the touch event, true will be passed.</param>
/// <remarks>
/// This method can be overridden in the derived class to change the processing of touch events.
/// </remarks>
protected void ProcessTouchPress(PointerEventData pointerEvent, bool pressed, bool released)

```

```

{
    if (pressed)
    {
        ProcessTouchPressPressed(pointerEvent);
    }

    if (released)
    {
        ProcessTouchPressReleased(pointerEvent);
    }

    m_InputPointerEvent = pointerEvent;
}

/// <summary>
/// Handle finger presses on the touch screen.
/// </summary>
protected void ProcessTouchPressPressed(PointerEventData pointerEvent)
{
    // Exit if push events are specified to be ignored.
    if (ignorePressEvent)
    {
        return;
    }

    // Click judgment is not affected by TimeScale.
    float time = Time.unscaledTime;

    // If a certain amount of time has not passed since the last press, exit.
    if (time - lastPointerDownTime < validPressInterval)
    {
        #if DebugCustomStandaloneInputModule
            Debug.LogFormat("No time has passed since the last press. : {0} - {1} < {2}", time, lastPointerDownTime, validPressInterval);
        #endif
        return;
    }

    lastPointerDownTime = time;

    // Exit if pressed outside SafeArea
    if (!Screen.safeArea.Contains(pointerEvent.position))
}

```

```

    {
#if DebugCustomStandaloneInputModule
    Debug.LogFormat("Outside SafeArea : {0}", pointerEvent.position);
#endif
    return;
}

// The GameObject on which the finger is currently hovering
var currentOverGo = pointerEvent.pointerCurrentRaycast.gameObject;

pointerEvent.eligibleForClick = true;
pointerEvent.delta = Vector2.zero;
pointerEvent.dragging = false;
pointerEvent.useDragThreshold = true;
pointerEvent.pressPosition = pointerEvent.position;
pointerEvent.pointerPressRaycast = pointerEvent.pointerCurrentRaycast;

// If the GameObject on which the finger is hovering has changed,
// release the selected object.
DeselectIfSelectionChanged(currentOverGo, pointerEvent);

// If the GameObject on which the finger is hovering has changed,
if (pointerEvent.pointerEnter != currentOverGo)
{
    // Send IPointerExitHandler's OnPointerExit event and IPointerEnterHandler's OnPoint
erEnter event.
    HandlePointerExitAndEnter(pointerEvent, currentOverGo);
    pointerEvent.pointerEnter = currentOverGo;
}

// Search for a handler that can handle the Pointer Down event (i.e., a GameObject that im
plements IPointerDownHandler) on the GameObject and its parent or higher levels where the fin
ger is overlapping, and send an OnPointerDown event to that GameObject if found.
var newPressed = ExecuteEvents.ExecuteHierarchy(currentOverGo, pointerEvent, Execut
eEvents.pointerDownHandler);

// Look for a handler that can handle the Click event (i.e., a GameObject that implements
IPointerDownHandler) on the GameObject and its parent or higher level where the finger is overl
apping. However, do not send the Click event yet.
var newClick = ExecuteEvents.GetEventHandler<IPointerClickHandler>(currentOverGo)
;

```

```

// If a handler that can handle the Pointer Down event is not found, the object that handles
the Click event is regarded as the handler that handles the Pointer Down event for convenience.
if (newPressed == null)
{
    newPressed = newClick;
}

#if DebugCustomStandaloneInputModule
    Debug.Log("Tapped : " + newPressed);
#endif

// If the same GameObject was pressed as last time,
if (newPressed == pointerEvent.lastPress)
{
    var diffTime = time - pointerEvent.clickTime;

#if DebugCustomStandaloneInputModule
    Debug.Log("Number of seconds elapsed since the last tap : " + diffTime);
#endif

// If the click interval is short, it is assumed to be a double tap (or more).
if (diffTime < multiClickPeriod)
{
    ++pointerEvent.clickCount;
}
else
{
    pointerEvent.clickCount = 1;
}

pointerEvent.clickTime = time;
}
else
{
    // A different GameObject was pressed than last time, so this is a single tap.
    pointerEvent.clickCount = 1;
}

#if DebugCustomStandaloneInputModule
    switch (pointerEvent.clickCount)
    {
        case 1:

```

```

        Debug.Log("Single tap"); ;
break;

case 2:
    Debug.Log("Double tap"); ;
break;

default:
    Debug.LogFormat("{0} tap", pointerEvent.clickCount);
break;
}

#endif

pointerEvent.pointerPress = newPressed;
pointerEvent.rawPointerPress = currentOverGo;
pointerEvent.pointerClick = newClick;
pointerEvent.clickTime = time;

// Look for a GameObject with an overlapping finger and a handler (i.e., a GameObject that implements IDragHandler) that can handle dragging in the hierarchy above its parent.
pointerEvent.pointerDrag = ExecuteEvents.GetEventHandler<IDragHandler>(currentOverGo);

// Send the OnInitializePotentialDrag event of the InitializePotentialDragHandler (if implemented for the GameObject) if there is a handler that can handle the drag.
if (pointerEvent.pointerDrag != null)
{
    ExecuteEvents.Execute(pointerEvent.pointerDrag, pointerEvent, ExecuteEvents.initializePotentialDrag);
}
}

/// <summary>
/// Handle the touch screen finger being released.
/// </summary>
protected void ProcessTouchPressReleased(PointerEventData pointerEvent)
{
    var currentOverGo = pointerEvent.pointerCurrentRaycast.gameObject;

    // Send PointerUp Event
    ExecuteEvents.Execute(pointerEvent.pointerPress, pointerEvent, ExecuteEvents.pointerUpHandler);
}

```

```

#if DebugCustomStandaloneInputModule
    Debug.Log("fingers have been released.");
#endif

    // If the currently overlapping GameObject is a Click handler,
    var pointerClickHandler = ExecuteEvents.GetEventHandler<IPointerClickHandler>(currentOverGo);

        if (pointerEvent.pointerClick == pointerClickHandler && pointerEvent.eligibleForClick)
        {
            // Send OnPointerClick.
            ExecuteEvents.Execute(pointerEvent.pointerClick, pointerEvent, ExecuteEvents.pointerClickHandler);
        }
        else if (pointerEvent.pointerDrag != null && pointerEvent.dragging)
        {
            Debug.Log("Distance " + (pointerEvent.pressPosition - pointerEvent.position).magnitude);

            // Enable clicking when dragging and the distance traveled is less than a certain value (
            The distance traveled depends on the resolution)
            if ((pointerEvent.pressPosition - pointerEvent.position).sqrMagnitude < dragThreshold
            * dragThreshold)
            {
                Debug.Log("Additional Click");
                ExecuteEvents.Execute(pointerEvent.pointerClick, pointerEvent, ExecuteEvents.pointerClickHandler);
            }

            // Send OnDrop
            ExecuteEvents.ExecuteHierarchy(currentOverGo, pointerEvent, ExecuteEvents.dropHandler);
        }

        pointerEvent.eligibleForClick = false;
        pointerEvent.pointerPress = null;
        pointerEvent.rawPointerPress = null;
        pointerEvent.pointerClick = null;

        // Send OnEndDrag.

```

```

if (pointerEvent.pointerDrag != null && pointerEvent.dragging)
{
    ExecuteEvents.Execute(pointerEvent.pointerDrag, pointerEvent, ExecuteEvents.endDr
agHandler);
}

pointerEvent.dragging = false;
pointerEvent.pointerDrag = null;

// Send OnPointerExit.
ExecuteEvents.ExecuteHierarchy(pointerEvent.pointerEnter, pointerEvent, ExecuteEvent
s.pointerExitHandler);
pointerEvent.pointerEnter = null;
}

/// <summary>
/// Send Submit to the currently selected object.
/// </summary>
/// <returns>Returns true if the Submit event was used by the currently selected object.</re
turns>
protected bool SendSubmitEventToSelectedObject()
{
    if (eventSystem.currentSelectedGameObject == null)
        return false;

    var data = GetBaseEventData();
    if (input.GetButtonDown(m_SubmitButton))
        ExecuteEvents.Execute(eventSystem.currentSelectedGameObject, data, ExecuteEvents
.submitHandler);

    if (input.GetButtonDown(m_CancelButton))
        ExecuteEvents.Execute(eventSystem.currentSelectedGameObject, data, ExecuteEvents
.cancelHandler);
    return data.used;
}

private Vector2 GetRawMoveVector()
{
    Vector2 move = Vector2.zero;
    move.x = input.GetAxisRaw(m_HorizontalAxis);
    move.y = input.GetAxisRaw(m_VerticalAxis);
}

```

```

if (input.GetButtonDown(m_HorizontalAxis))
{
    if (move.x < 0)
        move.x = -1f;
    if (move.x > 0)
        move.x = 1f;
}
if (input.GetButtonDown(m_VerticalAxis))
{
    if (move.y < 0)
        move.y = -1f;
    if (move.y > 0)
        move.y = 1f;
}
return move;
}

/// <summary>
/// Calculate and send a move event to the current selected object.
/// </summary>
/// <returns>Return true if the Move event was used by the currently selected object.</retu
rns>
protected bool SendMoveEventToSelectedObject()
{
    float time = Time.unscaledTime;

    Vector2 movement = GetRawMoveVector();
    if (Mathf.Approximately(movement.x, 0f) && Mathf.Approximately(movement.y, 0f))
    {
        m_ConsecutiveMoveCount = 0;
        return false;
    }

    bool similarDir = (Vector2.Dot(movement, m_LastMoveVector) > 0);

    // If the direction hasn't changed by more than 90 degrees, wait until the next event.
    if (similarDir && m_ConsecutiveMoveCount == 1)
    {
        if (time <= m_PrevActionTime + m_RepeatDelay)
            return false;
    }
    // If direction changed at least 90 degree, or we already had the delay, repeat at repeat rate
}

```

```

else
{
    if (time <= m_PrevActionTime + 1f / m_InputActionsPerSecond)
        return false;
}

var axisEventData = GetAxisEventData(movement.x, movement.y, 0.6f);

if (axisEventData.moveDir != MoveDirection.None)
{
    ExecuteEvents.Execute(eventSystem.currentSelectedGameObject, axisEventData, ExecuteEvents.moveHandler);
    if (!similarDir)
        m_ConsecutiveMoveCount = 0;
    m_ConsecutiveMoveCount++;
    m_PrevActionTime = time;
    m_LastMoveVector = movement;
}
else
{
    m_ConsecutiveMoveCount = 0;
}

return axisEventData.used;
}

protected void ProcessMouseEvent()
{
    ProcessMouseEvent(0);
}

[Obsolete("This method is no longer checked, overriding it with return true does nothing!")]
protected virtual bool ForceAutoSelect()
{
    return false;
}

/// <summary>
/// Handles all mouse events.
/// </summary>
protected void ProcessMouseEvent(int id)

```

```

{
    var mouseData = GetMousePointerEventData(id);
    var leftButtonData = mouseData.GetButtonState(PointerEventData.InputButton.Left).eventData;
    m_CurrentFocusedGameObject = leftButtonData.buttonData.pointerCurrentRaycast.gameObject;

    // Handle all left mouse buttons.
    ProcessMousePress(leftButtonData);
    ProcessMove(leftButtonData.buttonData);
    ProcessDrag(leftButtonData.buttonData);

    // Handle right and center clicks.
    ProcessMousePress(mouseData.GetButtonState(PointerEventData.InputButton.Right).eventData);
    ProcessDrag(mouseData.GetButtonState(PointerEventData.InputButton.Right).eventData.buttonData);
    ProcessMousePress(mouseData.GetButtonState(PointerEventData.InputButton.Middle).eventData);
    ProcessDrag(mouseData.GetButtonState(PointerEventData.InputButton.Middle).eventData.buttonData);

    if (!Mathf.Approximately(leftButtonData.buttonData.scrollDelta.sqrMagnitude, 0.0f))
    {
        var scrollHandler = ExecuteEvents.GetEventHandler<IScrollHandler>(leftButtonData.buttonData.pointerCurrentRaycast.gameObject);
        ExecuteEvents.ExecuteHierarchy(scrollHandler, leftButtonData.buttonData, ExecuteEvents.scrollHandler);
    }
}

protected bool SendUpdateEventToSelectedObject()
{
    if (eventSystem.currentSelectedGameObject == null)
        return false;

    var data = GetBaseEventData();
    ExecuteEvents.Execute(eventSystem.currentSelectedGameObject, data, ExecuteEvents.updateSelectedHandler);
    return data.used;
}

```

```

/// <summary>
/// Handle all mouse button state changes.
/// </summary>
protected void ProcessMousePress(MouseButtonEventData data)
{
    ProcessMousePressed(data);

    ProcessMouseReleased(data);
}

/// <summary>
/// Handle mouse button presses.
/// </summary>
protected void ProcessMousePressed(MouseButtonEventData data)
{
    // If not pressed, exit.
    if (!data.PressedThisFrame())
    {
        return;
    }

    // Exit if press events are specified to be ignored.
    if (ignorePressEvent)
    {
        return;
    }

    // Click judgment is not affected by TimeScale.
    float time = Time.unscaledTime;

    // If a certain amount of time has not passed since the last press, exit.
    if (time - lastPointerDownTime < validPressInterval)
    {
#if DebugCustomStandaloneInputModule
        Debug.LogFormat("No time has passed since the last press. : {0} - {1} < {2}", time, lastPointerDownTime, validPressInterval);
#endif
        return;
    }

    lastPointerDownTime = time;
}

```

```

var pointerEvent = data.buttonData;

// Exit if pressed outside SafeArea
if (!Screen.safeArea.Contains(pointerEvent.position))
{
    #if DebugCustomStandaloneInputModule
        Debug.LogFormat("Outside SafeArea: {0}", pointerEvent.position);
    #endif
    return;
}

// GameObject that the mouse pointer is currently hovering over
var currentOverGo = pointerEvent.pointerCurrentRaycast.gameObject;

pointerEvent.eligibleForClick = true;
pointerEvent.delta = Vector2.zero;
pointerEvent.dragging = false;
pointerEvent.useDragThreshold = true;
pointerEvent.pressPosition = pointerEvent.position;
pointerEvent.pointerPressRaycast = pointerEvent.pointerCurrentRaycast;

// If the GameObject that the mouse pointer is hovering over has changed, release the selected object.
DeselectIfSelectionChanged(currentOverGo, pointerEvent);

// Search for a handler that can handle the Pointer Down event (i.e., a GameObject that implements IPointerDownHandler) in the GameObject and its parent or higher levels where the mouse pointer is overlapping, and send an OnPointerDown event to that GameObject if found.
GameObject newPressed = ExecuteEvents.ExecuteHierarchy(currentOverGo, pointerEvent, ExecuteEvents.pointerDownHandler);

// Look for a handler that can handle the Click event (i.e., a GameObject that implements IPointerDownHandler) on the GameObject and its parent or higher level where the mouse pointer overlaps. However, do not send the Click event yet.
GameObject newClick = ExecuteEvents.GetEventHandler<IPointerClickHandler>(currentOverGo);

// If no handler was found that could handle the Pointer Down event,
// the object that handles the Click event is regarded as the handler that handles the Pointer Down event.
if (newPressed == null)

```

```

    {
        newPressed = newClick;
    }

#if DebugCustomStandaloneInputModule
    Debug.Log("The mouse button was pressed. : " + newPressed);
#endif

// If the same GameObject is pressed as before
if (newPressed == pointerEvent.lastPress)
{
    var diffTime = time - pointerEvent.clickTime;

#if DebugCustomStandaloneInputModule
    Debug.Log("Number of seconds elapsed since the last click : " + diffTime);
#endif

// If the click interval is short, it will be a double click (or more).
if (diffTime < multiClickPeriod)
{
    ++pointerEvent.clickCount;
}
else
{
    pointerEvent.clickCount = 1;
}

pointerEvent.clickTime = time;
}
else
{
    // A different GameObject was pressed than last time, so this is a single click
    pointerEvent.clickCount = 1;
}

#if DebugCustomStandaloneInputModule
    switch (pointerEvent.clickCount)
    {
        case 1:
            Debug.Log("Single click"); ;
            break;
    }
}

```

```

case 2:
    Debug.Log("Double click"); ;
    break;

default:
    Debug.LogFormat("{0} click", pointerEvent.clickCount);
    break;
}

#endif

pointerEvent.pointerPress = newPressed;
pointerEvent.rawPointerPress = currentOverGo;
pointerEvent.pointerClick = newClick;
pointerEvent.clickTime = time;

// Look for GameObjects with overlapping mouse pointers and handlers that can handle dragging (i.e., GameObjects that implement IDragHandler) in the hierarchy above their parents.
pointerEvent.pointerDrag = ExecuteEvents.GetEventHandler<IDragHandler>(currentOverGo);

// If you have a handler that can handle dragging, send the OnInitializePotentialDrag event of the InitializePotentialDragHandler to that GameObject (if you have implemented it).
if (pointerEvent.pointerDrag != null)
{
    ExecuteEvents.Execute(pointerEvent.pointerDrag, pointerEvent, ExecuteEvents.initializePotentialDrag);
}

m_InputPointerEvent = pointerEvent;
}

/// <summary>
/// Handle the mouse button being released.
/// </summary>
protected void ProcessMousePressReleased(MouseButtonEventData data)
{
    var pointerEvent = data.buttonData;

    // GameObject that the mouse pointer is currently hovering over
    var currentOverGo = pointerEvent.pointerCurrentRaycast.gameObject;

    // If the mouse button is released
}

```

```

if (data.ReleasedThisFrame())
{
    ReleaseMouse(pointerEvent, currentOverGo);
}
}

/// <summary>
/// Handle the mouse button being released.
/// </summary>
private void ReleaseMouse(PointerEventData pointerEvent, GameObject currentOverGo)
{
    ExecuteEvents.Execute(pointerEvent.pointerPress, pointerEvent, ExecuteEvents.pointerU
pHandler);

    var pointerClickHandler = ExecuteEvents.GetEventHandler<IPointerClickHandler>(curre
ntOverGo);

    // PointerClick and Drop events
    if (pointerEvent.pointerClick == pointerClickHandler && pointerEvent.eligibleForClick)
    {
        ExecuteEvents.Execute(pointerEvent.pointerClick, pointerEvent, ExecuteEvents.poin
terClickHandler);
    }

    if (pointerEvent.pointerDrag != null && pointerEvent.dragging)
    {
        // Enable clicking when dragging and moving less than a certain distance.
        if ((pointerEvent.pressPosition - pointerEvent.position).sqrMagnitude < dragThreshold
* dragThreshold)
        {
            ExecuteEvents.Execute(pointerEvent.pointerClick, pointerEvent, ExecuteEvents.poi
nterClickHandler);
        }

        ExecuteEvents.ExecuteHierarchy(currentOverGo, pointerEvent, ExecuteEvents.dropHa
ndler);
    }

    pointerEvent.eligibleForClick = false;
    pointerEvent.pointerPress = null;
    pointerEvent.rawPointerPress = null;
    pointerEvent.pointerClick = null;
}

```

```
if (pointerEvent.pointerDrag != null && pointerEvent.dragging)
{
    ExecuteEvents.Execute(pointerEvent.pointerDrag, pointerEvent, ExecuteEvents.endDragHandler);
}

pointerEvent.dragging = false;
pointerEvent.pointerDrag = null;

// Do PointerEnter and PointerExit again.
// This will allow us to handle mouseovers that were previously ignored because other objects were being pressed.

if (currentOverGo != pointerEvent.pointerEnter)
{
    HandlePointerExitAndEnter(pointerEvent, null);
    HandlePointerExitAndEnter(pointerEvent, currentOverGo);
}

m_InputPointerEvent = pointerEvent;
}

protected GameObject GetCurrentFocusedGameObject()
{
    return m_CurrentFocusedGameObject;
}
}
```

## Chapter 11 Profiling

### Profiling tools in uGUI

As a profiling tool for performance analysis of uGUI, **Frame Debugger** is available in addition to Unity's standard **Profiler**. For iOS, you can also use **Instruments** and **Frame Debugger** included in Xcode.

### Unity Profiler

For basic information on how to use Unity's Profiler, please refer to the official Unity manual or the Unity Game Optimization.

Unity Official Manual Profiler Overview

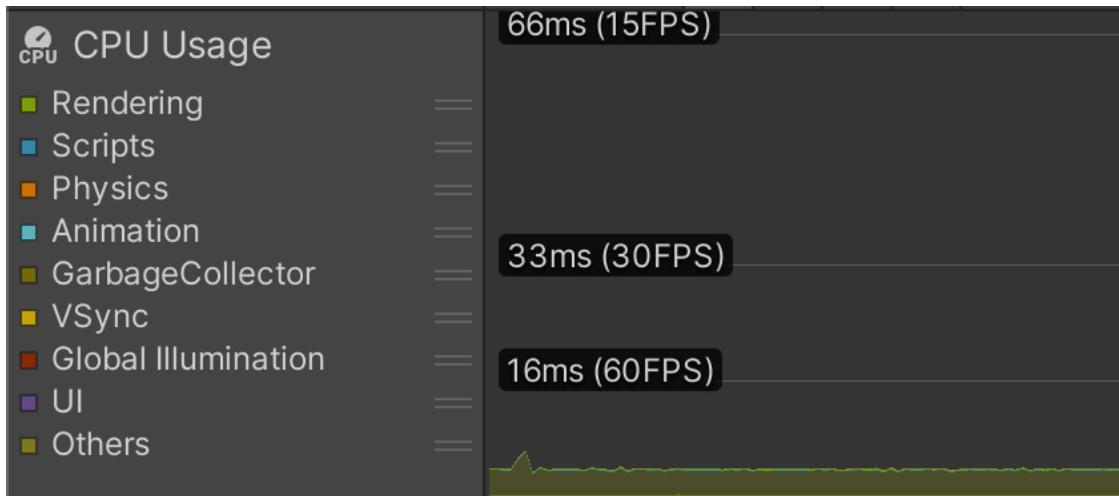
<https://docs.unity3d.com/Manual/Profiler.html>

Unity Game Optimization

<https://www.packtpub.com/product/unity-game-optimization-third-edition/9781838556518>

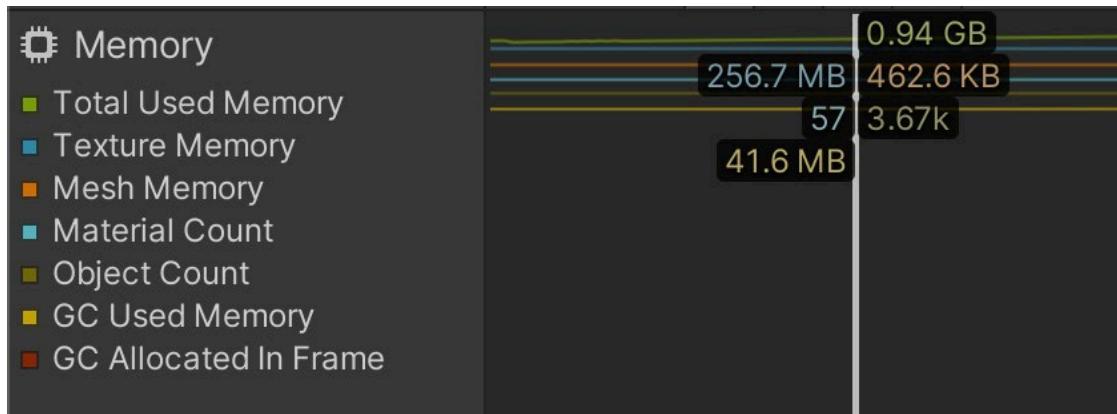
In the following, we will look at the functions related to uGUI.

## CPU Usage area



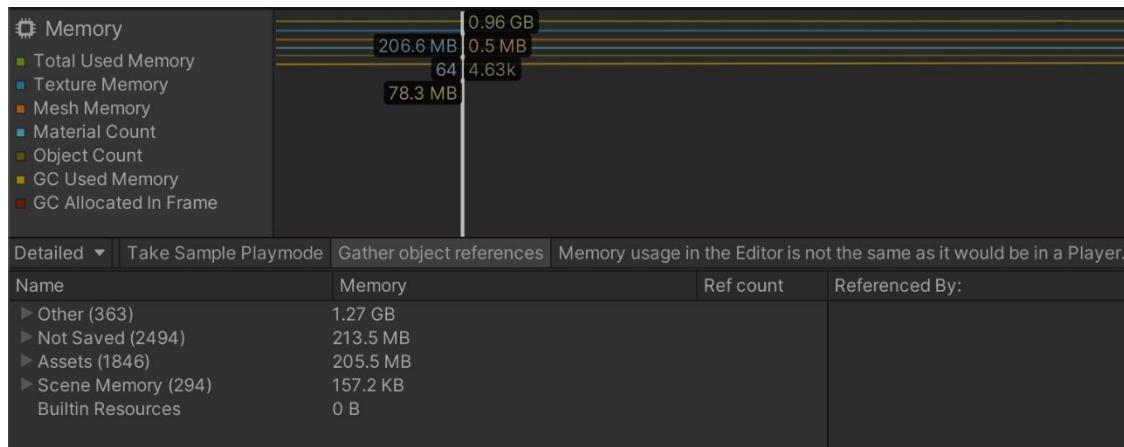
In Unity 2017.1, measurements for UI-related processing have been added, and UI-related processing is shown in purple in **CPU Usage**. Unfortunately, UI-related processes are not properly categorized, so there may be other UI-related processes that are not shown here. For example, `Canvas.SendWillRenderCanvases` (Canvas rebuild) is categorized as UI, while `Canvas.BuildBatch` (batch build) is categorized as Rendering (green) and Others (yellow-green). `BuildBatch` (batch build) is classified as Rendering (green) and Others (yellow-green). So, if you want to know more about UI-related load, check the UI area and UI Detail area described below rather than CPU Usage.

## Memory Area

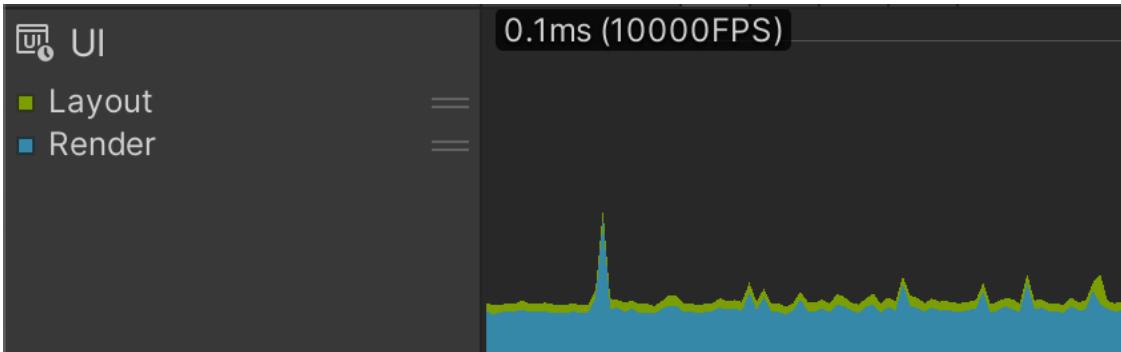


In the **Memory** area, **Texture Memory** and **GC Allocated In Frame** are what you want to pay attention to when profiling the UI. **Texture Memory** is literally the amount of memory used by textures. If you want to see specifically which textures are consuming how much memory, you can take a snapshot of memory usage using **Take Sample Playmode** in the breakdown view, and then look at *Assets/Texture2D* and *SceneMemory/Material*. If you want to see how much memory is being used, take a snapshot of the memory usage with Take Sample Playmode in the breakdown view and look at *Assets/Texture2D* and *SceneMemory/Material*.

**GC Allocated** indicates the amount of memory allocated that will be subject to garbage collection in the future. ui operations cause GC Alloc, and the near future garbage collection will cause performance degradation. If you are experiencing frame rate degradation, it would be a good idea to pay attention to **GC Allocated In Frame**.

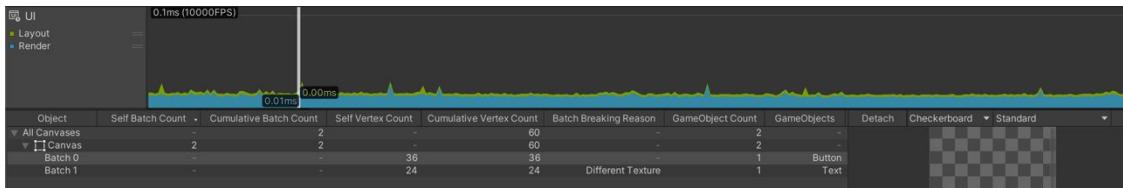


## UI Area



The UI area is a new area in Unity 2017.1, and appears when you scroll down in the Profiler window.

In the UI area, the CPU time consumed by Layout and Render is displayed. However, as explained above, there may be CPU processing related to the UI that is not shown here, so it should be viewed only as a reference.



Changing the preview type on the far right from **Standard** to **Overdraw** or **Composite overdraw** will allow you to see areas that are overloaded (being drawn on top of each other multiple times), which may help you see if you are experiencing fill rate problems.

In the case of overdraw, the bright red area indicates a high load, and in the case of mixed overdraw, the red area indicates a high load.

Each Canvas and Batch is displayed in the breakdown view, and clicking on each Canvas will display the drawing contents in the UI System View on the right. The preview background can be changed to **Checkerboard** (transparent), **white**, or **black**, and the preview type can be changed to **Standard**, **Overdraw**, or **Composite overdraw**. Pressing the Detach button will detach the window from the UI System View.

## Self Batch Count

---

Indicates the number of batches for this [Canvas](#). Basically, the fewer, the better.

## Cumulative Batch Count

---

The total number of batches for all [Canvas](#). Basically, the fewer, the better.

## Self Vertex Count

---

Number of vertices in this [Canvas](#).

## Cumulative Vertex Count

---

Total number of vertices in all [Canvas](#).

## Batch Breaking Reason

---

In Unity's **Profiler** and **Frame Debugger**, you can see the **Batch Breaking Reason** of a rendering batch. Reducing the total number of batches can improve rendering performance, so we want to avoid batch breaks as much as possible.

A list of reasons for batch interruptions and a sample project for generating batch interruptions are available on GitHub.

<https://github.com/Unity-Technologies/BatchBreakingCause>

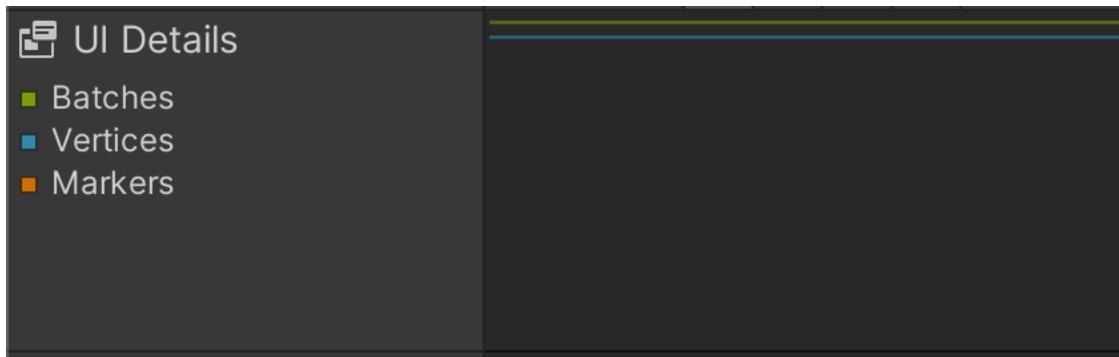
If you run the sample project and look at the **Frame Debugger**'s "Why this draw call can't be batched with the previous one" section of the Frame Debugger for details on why this mesh could not be included in the previous batch.

The following is a list and details of the reasons for the batch interruption.

- **Additional Vertex Streams** : Because we added vertex streams to the object with `MeshRenderer.additionalVertexStreams`.
- **Deferred Objects on Different Lighting Layers** : The `GameObject`'s Layer is different from the previous batch (using Deferred Rendering).
- **Deferred Objects Split by Shadow Distance**: Because the cascade of shadows (using Deferred Rendering) is different from the previous batch.

- **Different Combined Meshes** : Meshes combined (by StaticBatchingUtility.Combine or automatic static batching) are different from the previous batch.
- **Different Custom Properties**: Because there are properties in the Material Property Block that are different from those in the previous batch.
- **Different Lights** : Light is different from the previous batch.
- **Different Materials** : Material is different from the previous batch.
- **Different Reflection Probes** : Reflection Probes are different from the previous batch.
- **Different Shadow Caster Hash** : Shadow Casting shader is different from the previous batch. Or, a shader property/keyword that affects the Shadow Caster path has a different value than the previous batch.
- **Different Shadow Receiving Settings** : The Receive Shadows setting in MeshRenderer's Lighting is different from the previous batch.
- **Different Static Batching Flags** : GameObjects have different Static flags (Batching Static or not).
- **Dynamic Batching Disabled to Avoid Z-Fighting** : Dynamic batching has been disabled in Player Settings. Or dynamic batching was temporarily disabled to avoid Z-Fighting.
- **Instancing Different Geometries** : Trying to render a mesh/submesh with a different GPU instancing than the previous batch.
- **Lightmapped Objects** : The object has a different lightmap than the previous one. Lightmapped Objects : The object is using a different lightmap than the previous one, or the same lightmap but a different lightmap UV transformation is being used.
- **Lightprobe Affected Objects**: The objects are affected by a different Lightprobe than the previous batch.
- **Mixed Sided Mode Shadow Casters** : The Cast Shadows setting is different from the previous batch.
- **Multipass** : Because we are using multipass shaders.
- **Multiple Forward Lights** : Multiple lights are used due to the use of Forward Rendering.
- **Non-instantiable Property Set**: Property values that are not instantiated (due to GPU instancing) are different from the previous batch.
- **Odd Negative Scaling** : The meshes of the previous batch and the meshes of the previous batch have different negative scaling (the even-odd number of negative elements among the x, y, and z elements of the Transform Scale). For example, a mesh with a Scale of **(1, 1, 1)** and a mesh with a Scale of **(-1, 1, 1)** will be different batches.
- **Shader Disables Batching** : Shader Tags are set to "DisableBatching"="True".
- **Too Many Indices in Dynamic Batch** : Too many indices in dynamic batching (over 32768).
- **Too Many Indices in Static Batch** : Too many indices in static batching (over 32768 in MacOSX, over 49152 in OpenGL ES, over 65536 otherwise).
- **Too Many Vertex Attributes for Dynamic Batching** : Too many vertex attributes (over 900) for dynamic batching.
- **Too Many Vertices for Dynamic Batching** : There are too many vertices (over 300) in dynamic batching.

## UI Details area



The **UI Details** area is another area that was newly added in Unity 2017.1.

In the **UI Details** area, **Batches** shows the total number of batches in the [Canvas](#), **Vertices** shows the number of vertices, and **Markers** shows the number of event markers.

### Batches

You can see the number of batches for all [Canvas](#). The breakdown view shows all of the batches for each [Canvas](#). The most important thing to note here is the **Batch Breaking Reason**, which will be discussed later.

### Vertices

You can see the number of vertices in all [Canvas](#).

### Markers

Indicates interaction events such as button clicks and slider operations. It may get in the way of performance investigation, in which case you can click the orange box on the left to disable it for display.

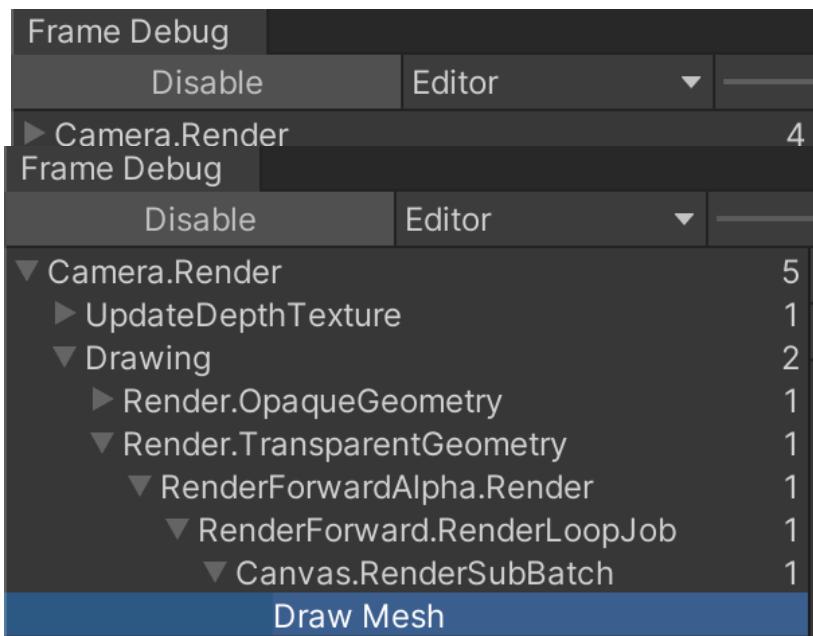
## Frame Debugger

Unity's Frame Debugger is useful for reducing the number of draw calls generated by uGUI and for investigating the reason for batch interruptions. Pressing Enable on the Frame Debugger allows you to see how the frame is being rendered at that point, one step at a time.

The Frame Debugger can be run even if you are not in Play Mode, which is useful for doing various trial and error operations.

Note that where the draw call is displayed depends on the `renderMode` of the `Canvas` component.

*RenderingOverlays -> Canvas.RenderOverlays.*

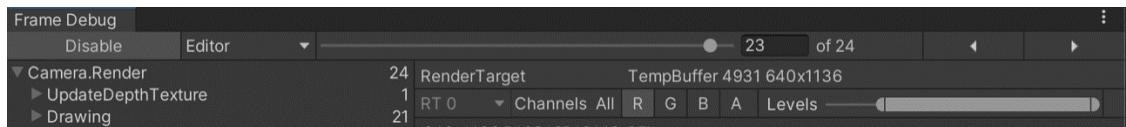


*In the case of ScreenSpaceCamera or WorldSpace, this is the Draw Mesh under Camera.Render -> Drawing -> Render.*

Move the slider at the top of the window to the left or right, or press the left or right triangle button to see how each step of the rendering process is executed; moving the slider to the left or right with **Game View** open will help you understand.



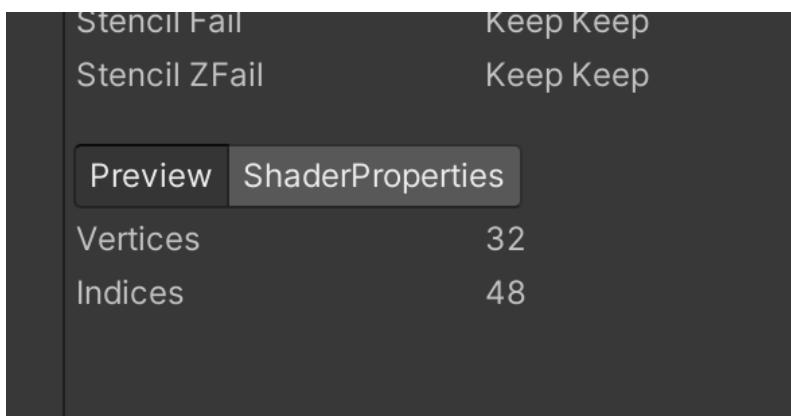
Only when rendering to [RenderTarget](#), it can be displayed separately in the Red/Green/Blue/Alpha channels at the top of the information panel. You can also use the **Levels** slider to divide the display by brightness level.



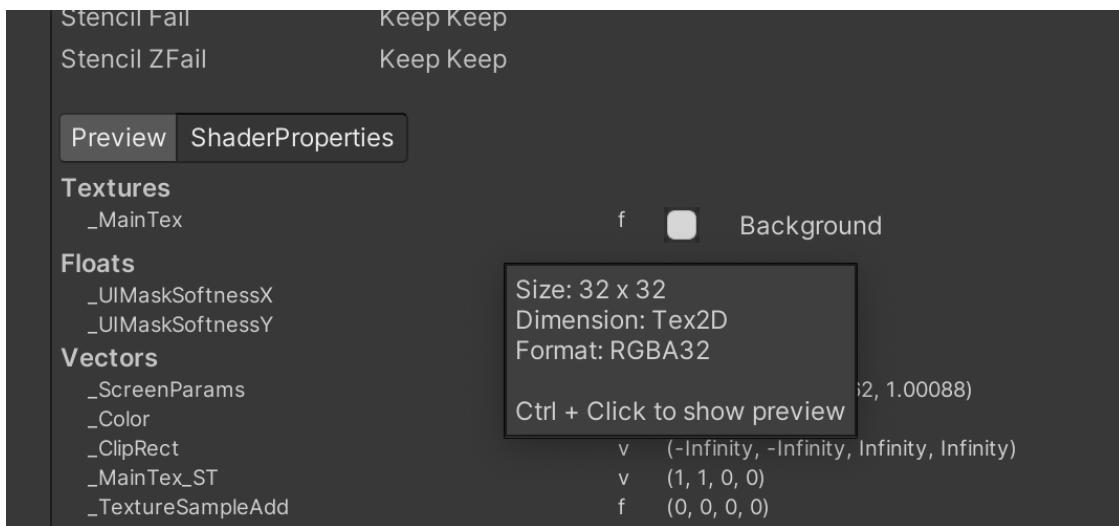
For each draw call, you can see the shader name, shader path, and each property.

If you are not using your own shaders for the UI, then your shaders will probably show built-in shaders such as *UI/Default*.

When you select the **Preview** tab, the number of vertices and indices for that draw call will be displayed.



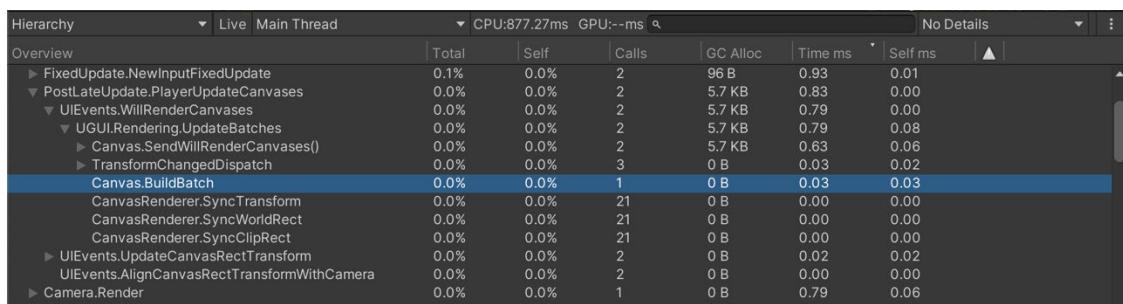
Select the **ShaderProperties** tab to display the properties of the shader.



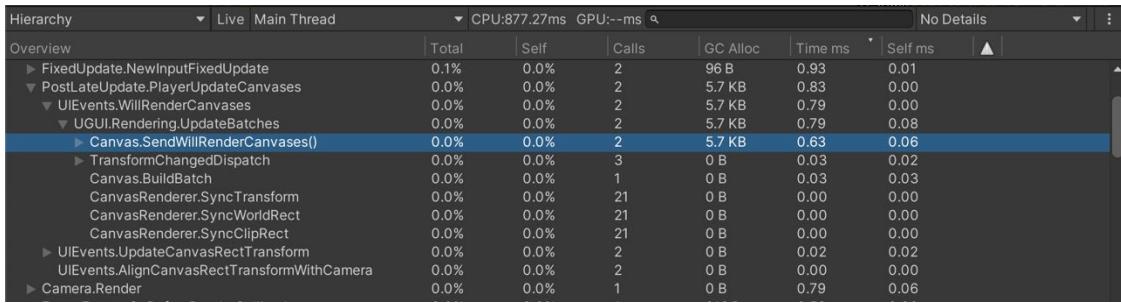
This will be useful when troubleshooting and debugging shaders. Also, if you hover the mouse cursor over the property where the texture is set, the size, dimension, and format will be displayed. You can also Ctrl-click on the texture to make it appear larger.

## Analysis of CPU Usage area and patterns of causes

Let's take a look at the CPU Usage area of the Unity Profiler.



`Canvas.BuildBatch` is the native code computation that performs the batch build process for Canvas.



`Canvas.SendWillRenderCanvases` contains a call to a C# script that subscribes to the `willRenderCanvases` event of the `Canvas` component. As mentioned earlier, the uGUI's `CanvasUpdateRegistry` class receives this event and uses it to perform the rebuild process. The dirty UI component will also update its own `CanvasRenderer` at this time.

The causes of performance degradation obtained from the analysis results of the CPU Usage area include the following patterns.

- [BuildBatch](#) or [Canvas::UpdateBatches](#) were consuming too much CPU time, it is possible that there are too many CanvasRenderer components for one [Canvas](#). Consider splitting the Canvas into sub-Canvas.
- If a lot of time is being spent on GPU UI rendering, it may indicate that the fragment shader pipeline is the bottleneck and is exceeding the pixel rate that the GPU is capable of handling. The most likely culprit is too many UI overdraws, which can be improved by reducing the number of pixels sampled by referring to the *Rendering Performance Improvement section* in *Chapter 3, Rendering*.
- [SendWillRenderCanvases](#) or [Canvas::SendWillRenderCanvases](#), then the Layout rebuild or Graphic rebuild is using too much CPU. Further analysis is needed.
- If a large portion of the [WillRenderCanvas](#) is consumed by [IndexedSet\\_Sort](#) or [CanvasUpdateRegistry\\_SortLayoutList](#), then it is consumed by sorting a list of dirty [ICanvasElement](#), which is done at the beginning of each stage of the [Canvas](#) rebuild. This sorting is done at the beginning of each stage of the Canvas rebuild; consider reducing the number of [ICanvasElement](#) components in the Canvas ( $\approx$  reducing the number of UI elements); splitting the Canvas into sub-Canvas can also help.
- If [Text\\_OnPopulateMesh](#) is consuming a lot of time, the cause is the creation of the mesh for the text. [resizeTextForBestFit](#) in [Text](#) should be set to [true](#). Also, make sure that the text is not being updated too often. If you don't want to just display the text, you can set [localScale](#) to [\(0, 0, 0\)](#) for better performance.
- If [Shadow\\_ModifyMesh](#) or [Outline\\_ModifyMesh](#) (or any other [ModifyMesh](#) implementation) is consuming your time, then the problem is the computation time of [MeshModifier](#). Consider removing these components, using static images to achieve these image effects, or using TextMesh Pro.

Please refer to the above to help improve your performance.

## **Unity uGUI Advanced Reference (English ver.)**

Published on November 17, 2021

Author: Heppoko

Published by: ReCirculation Project

Contact: [heppoko@heppoko-net.jp](mailto:heppoko@heppoko-net.jp)

