

Optimizing Unity UI

性能问题的关键

- [CPU](#)
- [GPU](#)
- **Memory**

影响CPU效率：

drawcall
physics
gc
粒子
动画
logic
...

影响GPU效率：

填充率
像素的复杂度，比如动态阴影，光照，复杂的shader等等
几何体的复杂度（顶点数量）
显存带宽
...

提升效率

减少顶点数量、简化复杂度

遮挡剔除（Occlusion culling）
LOD
纹理图集 更快加载 更少的状态切换
简单的shader
使用纹理光照，尽量不用实时光照
材质数量尽量少
...

优化显存带宽

图片压缩
mipmap

通过大量的性能测评数据，我们发现**渲染模块**、**UI模块**和**加载模块**往往占

据了游戏CPU性能开销的Top3

<http://youxichaguan.com/news/11204.html>

渲染模块可以说是任何项目中最为消耗CPU性能的引擎模块。

降低Draw Call

Draw Call是渲染模块优化方面的重中之重。一般来说，Draw Call越高，则渲染模块的CPU开销越大

<http://stackoverflow.com/questions/4853856/why-are-draw-calls-expensive>

降低Draw Call的方法主要是减少渲染物体的材质种类，并通过Draw Call Batching来减少其数量。

<http://docs.unity3d.com/Manual/DrawCallBatching.html>

但是，需要注意的是，游戏性能并非**Draw Call**越小越好。这是因为决定渲染模块性能的除了Draw Call之外，还有用于传输渲染数据的总线带宽。当我们使用Draw Call Batching将同种材质的网格模型拼合在一起时，可能会造成同一时间需要传输的数据（Texture、VB/IB等）大大增加，以至于造成带宽“堵塞”，在资源无法及时传输过去的情况下，GPU只能等待，从而反倒降低了游戏的运行帧率。

Draw Call和总线带宽是天平的两端，我们需要做的是尽可能维持两者的平衡，任何一边过高或过低，对性能来说都是无益的。

关于UGui可以参考

官方参考文章: <https://unity3d.com/cn/learn/tutorials/topics/best-practices>

ui source code: <https://bitbucket.org/Unity-Technologies/>

Unity UI 的组织与渲染

Canvases

负责组织几何数据合批，产生渲染指令提交给 Unity's Graphics system.

Geometry is provided to Canvases by *Canvas Renderer components*.

A *Sub-canvas*

隔离rebuild

The Batch building process (Canvases)

canvas 负责所有子元素（Canvas Renderer）组织几何数据，根据它们的排序、材质等等合批，产生渲染指令提交给Unity's graphics pipeline. 这个处理的结果是缓存的和重用的，直到这个canvas为dirty。（它的任何一个子元素发生改变<rebuild>）

其中据它们的排序、材质等等合批，这步操作是多线程的，所以在pc和mobile上性能是不同的。

The rebuild process (Graphics)

canvas 每帧调用 [WillRenderCanvases](#)

```

public class CanvasUpdateRegistry
{
    private static CanvasUpdateRegistry s_Instance;

    private bool m_PerformingLayoutUpdate;
    private bool m_PerformingGraphicUpdate;

    private readonly IndexedSet<ICanvasElement> m_LayoutRebuildQueue = new IndexedSet<ICanvasElement>();
    private readonly IndexedSet<ICanvasElement> m_GraphicRebuildQueue = new IndexedSet<ICanvasElement>();

    protected CanvasUpdateRegistry()
    {
        Canvas.willRenderCanvases += PerformUpdate;
    }
}

```

PerformUpdate runs a three-step process:

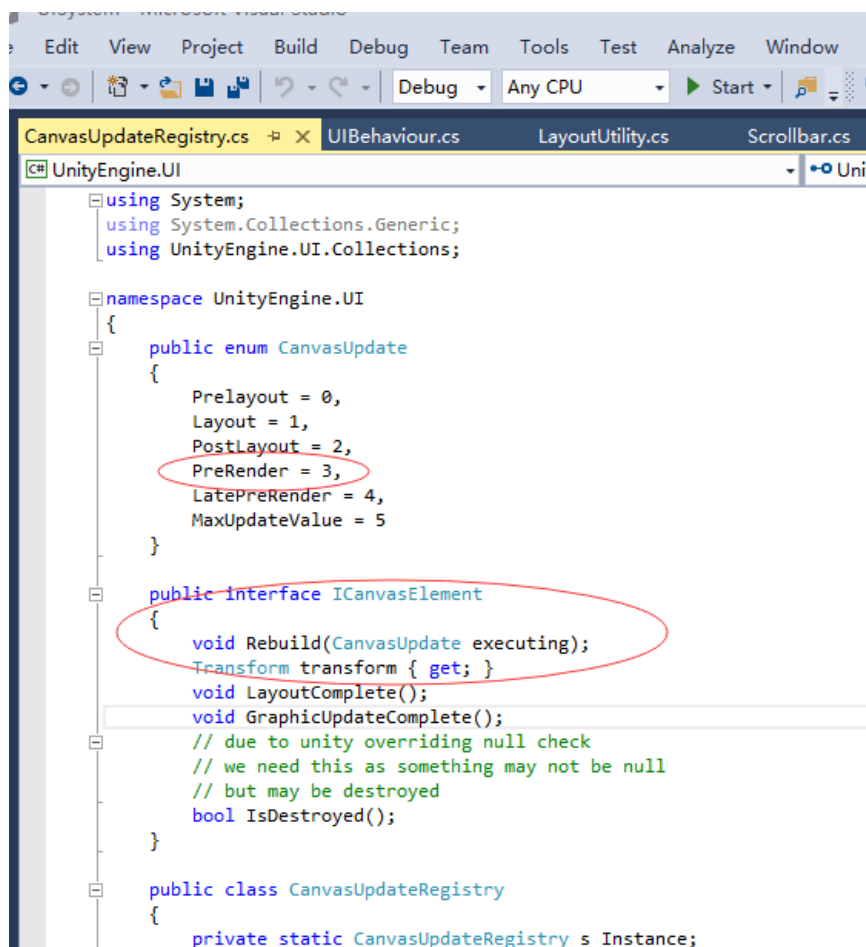
- Dirty Layout components are requested to rebuild their layouts, via the [ICanvasElement.Rebuild](#) method.
- Any registered Clipping components (such as Masks) are requested to cull any clipped components. This is done via ClippingRegistry.Cull.
- Dirty Graphic components are requested to rebuild their graphical elements.

For Layout and Graphic rebuilds, the process is split into multiple parts. Layout rebuilds run in three parts (PreLayout, Layout and PostLayout) while Graphic rebuilds run in two (PreRender and LatePreRender).

Layout rebuilds

Graphic rebuilds

[ICanvasElement](#)



```

using System;
using System.Collections.Generic;
using UnityEngine.UI.Collections;

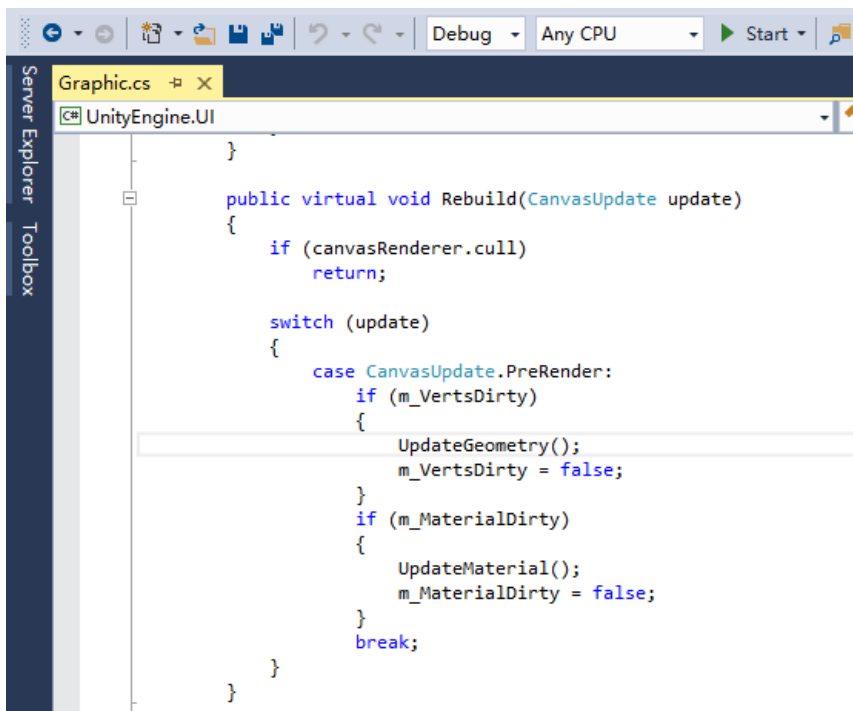
namespace UnityEngine.UI
{
    public enum CanvasUpdate
    {
        PreLayout = 0,
        Layout = 1,
        PostLayout = 2,
        PreRender = 3,
        LatePreRender = 4,
        MaxUpdateValue = 5
    }

    public interface ICanvasElement
    {
        void Rebuild(CanvasUpdate executing);
        Transform transform { get; }
        void LayoutComplete();
        void GraphicUpdateComplete();
        // due to unity overriding null check
        // we need this as something may not be null
        // but may be destroyed
        bool IsDestroyed();
    }

    public class CanvasUpdateRegistry
    {
        private static CanvasUpdateRegistry s_Instance;
    }
}

```

- If the vertex data has been marked as dirty (e.g. when the component's RectTransform has changed size), then the mesh is rebuilt.
- If the material data has been marked dirty (e.g. when the component's material or texture has been changed), then the attached Canvas Renderer's material will be updated.



Unity提供给我们一些默认的渲染队列，每一个对应一个唯一的值，来指导Unity绘制对象到屏幕上。这些内置的渲染队列被称为**Background, Geometry, AlphaTest, Transparent, Qverlay**。

ui渲染使用的是透明队列。

默认情况下，unity会根据物体与摄像机的距离来决定渲染顺序。

渲染队列	渲染队列描述	渲染队列值
Background	这个队列被最先渲染。它被用于skyboxes等。	1000
Geometry	这是默认的渲染队列。它被用于绝大多数对象。不透明几何体使用该队列。	2000
AlphaTest	通道检查的几何体使用该队列。它和Geometry队列不同，对于在所有立体物体绘制后渲染的通道检查的对象，它更有效。	2450
Transparent	该渲染队列在Geometry和AlphaTest队列后被渲染。任何通道混合的（也就是说，那些不写入深度缓存的Shaders）对象使用该队列，例如玻璃和粒子效果。	3000
Overlay	该渲染队列是为覆盖物效果服务的。任何最后被渲染的对象使用该队列，例如镜头光晕。	4000

CPU VS GPU

一般原则上认为ui性能问题是dc（cpu），然而实际过程中可能是fill-rate overutilization.（gpu）。

首先 精确定位性能瓶颈、不同项目问题可能不同。

通常有以下4种情况

- Excessive GPU fragment shader utilization (i.e. fill-rate overutilization)

- Excessive CPU time spent rebuilding a Canvas batch
- Excessive numbers of rebuilds of Canvas batches (over-dirtying)
- Excessive CPU time spent generating vertices (usually from text)

Unity UI Profiling Tools

There are several profiling tools useful for analyzing a Unity UI's performance. The key tools are:

- Unity Profiler
- Unity Frame Debugger
- XCode's Instruments or Intel VTune
- XCode's Frame Debugger or Intel GPA

Analyzing profiler results

cpu

单canvas元素过多

If `Canvas.BuildBatch` or `Canvas::UpdateBatches` seems to be using an excessive amount of CPU time, then the likely problem is an excessive number of Canvas Renderer components on a single Canvas. ([Splitting Canvases](#))

--布局

In the case that a large portion of *WillRenderCanvas* is spent inside *IndexedSet_Sort* or *CanvasUpdateRegistry_SortLayoutList*, then time is being spent sorting the list of dirty Layout components. Consider reducing the number of Layout components on the Canvas. ([Replacing layouts with RectTransforms](#))

-- 文本更新

If excessive time seems to be spent in *Text_OnPopulateMesh*, then the culprit is simply the generation of text meshes. See the [Best Fit](#) and [Disabling Canvas Renderers](#) sections for possible remediations, and consider the advice inside [Splitting Canvases](#) if much of the text being rebuilt is not actually having its underlying string data changed.

-- 效果器

If time is spent inside *Shadow_ModifyMesh* or *Outline_ModifyMesh* (or any other implementation of *ModifyMesh*), then the problem is excessive time spent calculating mesh modifiers. Consider removing these components and achieving their visual effect via static images.

--动态元素

If there is no particular hotspot within *Canvas.SendWillRenderCanvases*, or it appears to be running every frame, then the problem is likely that dynamic elements have been grouped together with static elements and are forcing the entire Canvas to rebuild too frequently.

gpu

If an excessive amount of time is spent drawing the UI on the GPU, and the frame debugger indicates that the fragment shader pipeline is the bottleneck, then the UI is likely exceeding the pixel fill rate which the GPU is capable of. The most likely cause is excessive UI overdraw. ([Remediating fill-rate issues](#))

UI Canvas rebuilds

- If the number of drawable UI elements on a Canvas is large, then calculating the batch itself becomes very

expensive. This is because the cost of sorting and analyzing the elements grows more-than-linearly to the number of drawable UI elements on the Canvas. (元素过多)

- If the Canvas is dirtied frequently, then excessive time may be spent refreshing a Canvas that has relatively few changes. (频繁更新)

Child order

ui的构建是从后往前依照元素在hierarchy的顺序.批处理也是依照这个次序合并相同材质和纹理的元素。如果元素没有被打断。(

1. An “intermediate layer” is a graphical object with a different material, whose bounding box overlaps two otherwise-batchable objects and is placed in the hierarchy between the two batchable objects.)

- Reorder the drawables so that the batchable objects are not interposed by the non-batchable object; that is, move the non-batchable object above or below the batchable objects.
- Tweak the positions of the objects to eliminate invisible overlapping space.

Splitting Canvases

Performant UI design requires a balance between minimizing the cost of rebuilds and minimizing wasted draw calls.

canvas太多 太少都不好，具体怎么平衡

General guidelines

拆分成2个canvas。同时变化的放在一个canvas上比如progress bar and a countdown timer。

On one Canvas, place all elements that are static and unchanging, such as backgrounds and labels. These will batch once, when the Canvas is first displayed, and then will no longer need to rebatch afterwards.

On the second Canvas, place all of the “dynamic” elements – the ones that change frequently.

(如果变化的元素过多，进一步细分，经常变化的、偶尔变化的。)

Remediating fill-rate issues

- Reducing the complexity of fragment shaders.
 - See the “UI shaders and low-spec devices” section for more details.
- Reducing the number of pixels that must be sampled.

Eliminating invisible UI

隐藏不可见的ui。在不更改设计的情况下最简单的方法。主要适用全屏ui，其后面的ui都可以disable，

disable the root GameObject or GameObjects containing the invisible UI elements, For an alternate solution,

[Disabling Canvas Renderers](#)。

Disabling invisible camera output

全屏ui显示时，关闭不可见的camera。

Majority-obscured cameras

近全屏ui，使用render texture 渲染场景 然后绘制在ui后面。关闭不可见的camera。

Composition-based UIs

--实际应用意义不大

UI shaders and low-spec devices

Input and raycasting in Unity UI

Erroneous mouse input detection on mobile (5.3)

5.4以前

Graphic Raycaster 即使在安卓和ios上，每帧都会检测鼠标位置。

This is a waste of CPU time, and has been witnessed consuming 5% or more of a Unity application's CPU frame time.

This issue is resolved in Unity 5.4.

5.4之前unity建议自己处理。

If using a version of Unity older than 5.4, it is strongly recommended that mobile developers create their own Input Manager class. This can be as simple as copying Unity's Standard Input Manager from the Unity UI source and commenting out the ProcessMouseEvent method as well as all calls to that method.

Raycast optimization

去掉不必要的Raycast Target

UI优化总结

0 最好的优化就是设计简单，同时简化资源。

简化资源是非常行之有效的优化手段。在大量的项目中，其渲染资源其实是“过量”的，如过量的网格资源、不合规的纹理资源等等。

设计从简太复杂的ui得不偿失。复杂的ui开发维护耗时、玩家体验也不会好。

1.制作要求。 排列顺序：UGUI的渲染顺序为从上到下，在使用同一图集的情况下，连续多次对图片（相同或者不同）的渲染，只占用一个dc，对文字的渲染同理，只要不交叉渲染图片和文字，理论上讲，这样可以把整个UI的Draw降低到2。

如果顺序为：

- 1.渲染A，使用材质1
- 2.渲染B，使用材质1
- 3.渲染C，使用材质2

那么drawcall是2个，AB进行了动态批次。

如果顺序为：

- 1.渲染A，使用材质1
- 2.渲染C，使用材质2
- 3.渲染B，使用材质1

那么drawcall就是3个，AB的批次被打断

2 界面上字体尽量不要多种。 确实需要，同一种字体尽量一起排列。

3 “易变”的单独canvas

(Important reminder: Whenever any drawable UI element on a given Canvas changes, the Canvas must re-run the batch building process. This process re-analyzes every drawable UI element on the Canvas, regardless of whether it has changed or not. Note that a “change” is any change which affects a UI object's appearance, including the sprite assigned to a sprite renderer, transform position & scale, the text contained in a text mesh, etc.

)

(

Performant UI design requires a balance between minimizing the cost of rebuilds and minimizing wasted draw calls.

)

4 去掉不必要的Raycast Target

5 Layout components是相当费的，少用。 如果元素少 可以使用RectTransform-based Layouts

For example, a simple two-column layout can be achieved with two RectTransforms:

- The left column's anchors should be X: (0, 0.5) and Y: (0, 1)
- The right column's anchors should be X: (0.5, 1) and Y: (0, 1)

6. Disabling Canvas Renderers。

(不enable or disable, 这样缓存batching, 不会rebuild。注意这时候ui上的逻辑MonoBehaviours也应该disable, 不然依然响应事件, 虽然不可见了。

)

7 避免复杂的shader。

8 ui尽量改大小 不用缩放。

9 特定控件的优化(scroll view) 与注意

(比如 text控件 使用相同字体的不同style, 导致不同的纹理)

10 固定字符数量的文本用图片代替。

11 小心不可见的大控件。

ps:

根据unity官方文档描述:

1). 动态批次是逐顶点处理的, 因此仅对少于900个顶点的mesh有效。如果shader使用了顶点位置, 法线和UV那么仅支持低于300顶点的mesh, 而如果shader使用了顶点位置, 法线、UV0、UV1和切向量, 则之多仅支持180顶点。

2). 缩放问题

缩放对于批次是有影响的, 这里涉及到一个统一缩放和非统一缩放的概念。统一缩放即为三轴同比例缩放, 比如(1,1,1), (2,2,2) (5,5,5) ... 非统一缩放即为三轴不同比例缩放, 如(1,2,1) (2,1,1) (1,2,3) 等等。

Unity对统一缩放的对象是不进行动态批次的, 而对非同一缩放的对象是可以进行动态批次的。

统一缩放是一份mesh单独一次提交渲染。非统一缩放的是复制mesh, 所以可以和批。