红字具体优化方法 蓝字运行机制

1. UGUI shader特效

主要使用Image组件，与3D Plane的效果一样。

方式是给Image组件对应修改后的Shader（基于Sprite-Default），与普通shader不同的是，需要进行模板检测，所以需要对材质赋予指定ID作为组别。

优点：

1. 是基于Sprite-Default的着色器使用[PerRendererData]修饰MainTex，因此多组件可以选用不同Image但使用同一材质球，他们的贴图是经系统打包为一个图集的
2. 使用Stencil ID控制蒙版选择，即使是分离的两个组件，如果ID一致可以不受层级限制共用效果
3. UGUI粒子特效
4. UGUI渲染优化
   1. Draw call

如果GameObject下面有多个精灵，优先看父节点在Hierarchys视图中的排序，决定父节点的渲染先后。然后再依次看子节点中的Hierarchy视图的排序。如果还有孙节点一次类推。这样的话如果没有ABA叠层的情况那么图集永远是一个drawcall.（参考3.3.2子对象顺序）

* 1. Raycast Target

一般非Button不需要这个，可以关掉。Raycast会对命中目标进行排序（每帧），从而选择点击到的组件是哪个。关闭不需要的组件的Raycast Target可以减少Raycast目标列表，提高测试速度，降低CPU压力

Sub画布上的overrideSorting属性将导致Graphic Raycast测试停止攀爬Transform层次结构。如果可以在不导致排序或光线投射检测问题的情况下启用它，则应该使用它来降低光线投射层次结构遍历的成本。

* 1. 合批

1. 合并图集

静态Image组件的Source Image可以使用Sprite Atlas合并成一个图集，能够有效减少Draw Mesh调用次数降低Batches

2）设置子对象顺序

Unity通过从上到下遍历层次结构并收集使用相同材质，相同纹理且没有中间层的所有对象来构建批次。“中间层”是具有不同材质的图形对象，其边界框与两个其他可混合对象重叠，并且放置在两个可混合对象之间的层次结构中。中间层迫使批次被破坏。

当文本和精灵位于彼此附近时，最常出现此问题：文本的边界框可以无形地与附近的精灵重叠，因为文本字形的多边形大部分都是透明的。这可以通过两种方式解决：

①重新排序drawables，以避免不可处理的对象在可处理的对象中; 也就是说，将非batchable对象移动到可修改对象的上方或下方。

②调整对象的位置以消除不可见的重叠空间。

这两个操作都可以在Unity编辑器中打开并启用Unity Frame Debugger来执行。通过简单地观察Unity Frame Debugger中可见的绘制调用的数量，可以找到最小化由于UI元素重叠而浪费的绘制调用的数量的顺序和位置。

* 1. 精简着色器

默认UI着色器存在mask / clip等操作的支持，如果不需要可以创建去掉这些语句的更精简的Shader

* 1. 选择关闭组件而不是隐藏

Unity本身所有UI组件始终使用Tranparent queue，请避免通过将组件alpha设置为0的方式来隐藏UI元素，因为该元素仍将被发送到GPU并且可能需要宝贵的渲染时间

* 1. 查看Profiler
     1. 如果*Canvas.BuildBatch* 或*Canvas :: UpdateBatches* 似乎使用了过多的CPU时间，那么可能的问题是单个Canvas上的Canvas Renderer组件数量过多。
     2. 如果似乎在Text\_OnPopulateMesh中花费了过多的时间，那么罪魁祸首就是生成文本网格。请参阅最适合和禁用画布部分以获取可能的补救措施，如果正在重建的大部分文本实际上没有更改其基础字符串数据，请考虑拆分画布内的建议。
     3. 如果时间花在内部Shadow\_ModifyMesh 或Outline\_ModifyMesh （或任何其他实施ModifyMesh ），则问题是花费计算网格改性剂过多的时间。考虑删除这些组件并通过静态图像实现其视觉效果。
     4. 如果Canvas.SendWillRenderCanvases中没有特定的热点，或者它似乎在每一帧都运行，那么问题很可能是动态元素已经与静态元素组合在一起，并且迫使整个Canvas过于频繁地重建。请参阅拆分画布步骤。
     5. 如果*Canvas.SendWillRenderCanvases中*没有特定的热点，或者它似乎在每一帧都运行，那么问题很可能是动态元素已经与静态元素组合在一起，并且迫使整个Canvas过于频繁地重建。
  2. 使用静态背景或停止渲染背景

全屏不透明Canvas下其他的Canvas以及3D场景摄像机可以关闭（不是隐藏）。因此，如果打开一个完全全屏的UI，禁用被遮挡的世界空间相机将有助于通过简单地消除渲染3D世界的无用工作来减少GPU压力。

如果UI未覆盖整个3D场景，您可能希望将场景渲染到纹理一次并使用它而不是连续渲染它。您将无法在3D场景中看到动画内容，但这在大多数情况下都是可以接受的。

许多“全屏”用户界面实际上并没有掩盖整个3D世界，而是让世界的一小部分可见。在这些情况下，捕获渲染纹理中可见的世界部分可能更为理想。如果世界的可见部分在渲染纹理中“缓存”，则可以禁用实际的世界空间相机，并且可以在UI屏幕后面绘制缓存的渲染纹理以提供3D世界的假冒版本。

* 1. Canvas重建

当给定Canvas上的任何可绘制UI元素发生更改时，Canvas必须重新运行批处理构建过程。此过程重新分析Canvas上的每个可绘制UI元素，无论它是否已更改。

通常选择将Canvas拆分，使用subCanvas以减少UI刷新时需要重新绘制的组件数量（组件修改后会直接重新绘制同Canvas下所有组件）。

例如拆分成静态和动态元素的两个Canvas。

在一个Canvas上，放置所有静态且不变的元素，例如背景和标签。这些将在首次显示Canvas时批量处理一次，然后不再需要重新加载。

在第二个画布上，放置所有“动态”元素 - 经常变化的元素。这将确保此Canvas主要重新调整脏元素。如果动态元素的数量变得非常大，则可能需要进一步将动态元素细分为一组不断变化的元素（例如，进度条，计时器读数，任何动画）以及仅偶尔改变的一组元素。

注意，重建画布也会导致Canvas丢弃其VBO数据。重新启用Canvas将需要Canvas（和任何Sub-canvases）来运行重建和重新分配过程。如果经常发生这种情况，CPU使用率的增加会导致应用程序的帧速率出现问题。

一种可能但又很糟糕的解决方法是将UI显示/隐藏到其自己的Canvas或Sub-canvas上，然后仅启用/禁用此对象上的Canvas组件。

这将导致UI的网格不被绘制，但它们将保留在内存中，并且将保留其原始批处理。此外，不会在UI的层次结构中调用OnEnable或OnDisable回调。

但请注意，这不会禁用隐藏UI中的任何MonoBehaviours，因此这些MonoBehaviours仍将接收Unity生命周期回调，例如Update。

* 1. 文本网格重建

每当更改UI文本组件时，文本组件都必须重新计算用于显示实际文本的多边形。

对于显示大量文本标签的任何UI，此行为都存在问题，最常见的是排行榜或统计屏幕。(可以选择将元素移出摄像机而不是关闭/隐藏)

* 1. 字体图集

原理是将需要的文字渲染到一张图集上进行获取渲染。图集包含所有可能显示文字，他们存储在内存中，需要的时候调用

•如果两个Text组件共享相同的大小，则字体图集中将包含一个字形。

•如果两个文本组件不共享相同的大小（例如，一个是16点，另一个是24点），则字体图集将包含两个不同大小的字母“A”的副本。

•如果一个Text组件是粗体而另一个不是粗体，那么字体图集将包含粗体'A'和常规'A'。

使用当前由活动UI文本组件显示的字形，以相同的大小重建图集。如果当前使用的一组字形不能适合与当前地图集大小相同的地图集，则通过将地图集的较短维度加倍来创建更大的地图集。

因此尽可能使用非动态字体并预配置对所需字形集的支持。这通常适用于使用良好约束字符集的UI，例如仅拉丁字符/ ASCII字符，并且具有小范围的大小。

如果必须支持极大范围的字符，例如整个Unicode集，则必须将字体设置为Dynamic。为了避免可预测的性能问题，请通过[Font.RequestCharactersInTexture](http://docs.unity3d.com/ScriptReference/Font.RequestCharactersInTexture.html)使用一组适当的字符在启动时[填充](http://docs.unity3d.com/ScriptReference/Font.RequestCharactersInTexture.html)字体的字形图集。

对于分数，可显示的字符是从众所周知的字形集（数字0-9）中提取的，不会跨地方变化，并且彼此之间的距离固定。将整数分解为数字并显示适当的数字精灵是相对微不足道的。这种专门的数字显示系统可以以无分配的方式构建，并且比Canvas驱动的UI Text组件更快地计算，动画和显示。

如果字体的字符集非常大，则回退字体占用的内存量可能会过多。这在包括象形字体（例如日文汉字或汉字）时最常见。

* 1. 混合使用文字系统

对于游戏中显示的文字可以使用TextMesh Pro的SDF字体，提高显示效果和特效处理。

对于游戏中的输入框建议使用UGUI自带输入框，使用动态字体。

* 1. 使用RectTransform进行布局

Layout组件相对昂贵，因为每次标记为脏时，它们必须重新计算子元素的大小和位置。（有关详细信息，请参阅“基础”步骤的“图形重建”部分。）如果给定布局中的元素数量相对较少且数量固定，并且布局具有相对简单的结构，则可以使用RectTransform替换Layout组件进行布局。

1. 总结

* 限制Text组件中“Best Fit”和“Rich Text”的使用
* 不要使用dyanmic，如果可能的话使用位图/静态字体
* 限制Text游戏对象上Outline / Shadow组件的使用。相反，在位图字体中烘焙效果
* 如果可能的话，坚持使用'Simple'的精灵。避免使用大型“Clamp”精灵
* 展平变换层次结构。嵌套越多，正确定位子项所需的变换计算就越多
* 取消选中“画布”设置中的“Pixel Perfect”.
* Screen Space - Overlay比Screen Space - Camera略快。在Overlay中，UI立即绘制在屏幕顶部，避免了Camera空间必须经过的所有剔除和排序。（性能降低应该不是特别严重，而且屏幕空间相机很方便，所以可以选择Screen Space - Camera，只要不引起问题）
* 尽量避免将UI图像/文本叠加在一起。这会导致更多的Overdraw，并会降低速度
* 根据静态与动态移动的内容，将UI细分为多个画布。更改画布中UI元素的位置/旋转/缩放/子画面/文本会导致为其所有subCanvas重建画布的网格。因此，为游戏的整个UI设置单个Canvas并不是一个好主意。将其划分为一起更新/更改的逻辑组。也许每个菜单/屏幕/子屏幕/弹出窗口都有自己的Canvas，也许hud top有subCanvas和hud底部也有单独subCanvas等等。
* 访问时避免使用Camera.FindObjectWithTag和Camera.main调用，直接参考您正在使用的相机。
* 首选启用/禁用画布本身而不是整个游戏对象。避免整个层次结构上的OnEnable / OnDisable回调，这可能很昂贵。
* 避免使用LayoutGroups。推出自己的。（<https://www.youtube.com/watch?list=PLX2vGYjWbI0Rzo8D-vUCFVb_hHGxXWd9j&v=_wxitgdx-UI>）
* 在从池中借用对象之前启用它时，首先更新池对象属性。这样可以避免对对象层次结构造成不必要的污染
* 使用包含许多项目的ScrollRects，将画布（没有Pixel Perfect）附加到滚动矩形，以便滚动项目不会弄脏外部画布中的元素
* 汇集视图中的项目。进入/退出视图时启用/禁用元素
* 压缩滚动视图的速度，这样您就不会得到非常小的增量，只需对元素进行非常小的更改，而不会在屏幕上显着改变
* 关闭静态或非交互元素的Raycast Target。这减少了碰撞/点击/输入光线投射检查器必须经历的UI对象的数量，以确定哪些UI元素应该接收事件
* 从Canvases中取出不需要输入事件的GraphicsRaycaster组件
* 将精灵包装在一起以减少拉扯。使用Texture Packer或Unity的atlas / packer系统

1. 实例
   1. 特效层级
      1. Hierarchy中从上到下依次渲染
      2. 在两个UI中夹一个粒子系统

次序从后往前：UI A 🡪粒子系统🡪UI B

需求是B会遮住粒子系统，那么需要给B一个新的父级Canvas B。

设置粒子系统的Sorting Layer，对Canvas B勾选Override Sorting，选用相同的SortingLayer并设置Order大于粒子系统的Order

* + 1. 层级渲染优先级（从最优先开始）
       1. Camera.depth
       2. Sorting Layer
       3. Order in Layer
       4. Specify Render Queue
       5. Distance to Camera
          1. Perspective/Orthographic
          2. Custom Axis sort mode
          3. Sprite Sort Point
       6. Sorting Group
       7. Material/Shader
       8. A Tiebreaker occurs when multiple Renderers have the same sorting priorities.

## 特效遮罩解决方案

* + 1. RectMask2D

RectMask2D不需要依赖一个Image组件，其裁剪区域就是它的RectTransform的rect大小。

优点：

1. 没有额外Drawcall消耗
2. 不会Overdraw
3. 性能快

缺点

1. RectMask2D节点下的所有孩子都不能与外界UI节点合批且多个RectMask2D之间不能合批。
2. 计算depth的时候，所有的RectMask2D都按一般UI节点看待，只是它没有CanvasRenderer组件，不能看做任何UI控件的bottomUI。
3. 只支持矩形
   * 1. Unity Mask

优点：

1. 支持透明贴图以控制蒙版形状

缺陷：

1. Mask需要单独设置一个Draw Call
2. 可能打断其他UI合并Draw Call的过程
3. 开销大，会有不必要的Overdraw
4. 可能会有边缘锯齿
5. 对粒子系统无效
   * 1. Sprite Mask

优点：

1. 支持透明贴图以控制蒙版形状，且可以在设置alpha cutoff后完全抠除透明部分
2. 支持粒子系统

缺点

1. 大量的Overdraw（尤其粒子）
2. 其他未知（开销可能有点大）
   * 1. 第三方解决方案 基于网格修改的方案

圆形/环形方案 ：http://www.cnblogs.com/leoin2012/p/6425089.html

根据图片生成方案 ： <http://www.cnblogs.com/leoin2012/p/6822859.html>

优点

1. Drawcall消耗随组件数量增加大量减少（Unity Mask因为打包图集所以不同组件各需消耗一个Drawcall）
2. Overdraw也大量减少
3. 依据贴图和Collider程序化生成Mask对美术操作较为友好

缺点

1. 本质上是手动绘制顶点和面作为蒙版进行裁切的过程，增加摄像机内的定点数和面数，这个数值受蒙版图片边缘复杂度影响，边越少开销越少
2. 为Unity Mask的替代方案，有可能继承Unity Mask的特性产生问题，在矩形区域开销仍然会比RectMask2D大
3. 可能不适用于粒子系统
   * 1. 第三方整体解决方案 UI Particle

优点：

1. 解决Overlay上Particle无法显示的问题
2. 解决Mask不能准确抠除透明不部分的问题

缺点

1. Overdraw较高
2. 脚本略复杂
3. 有一定额外开销
   * 1. Stencil检测遮罩（Unity Mask实际使用此方法）

使用支持Stencil的shader的材质的Image作为遮罩，对使用支持Stencil的shader的材质的组件使用。要求两者Shader都要进行Stencil检测，并匹配Stencil ID

优点

1. 可以跨组件使用。同一Canvas下不论层级关系和空间位置关系，同一Stencil ID共用遮罩
2. 对粒子系统有效
3. 对有透明部分的Image开启Use Sprite Mesh后可以根据图像进行较复杂遮罩

缺点

1. Sprite Mesh经过简化，不能完全表示原贴图轮廓
2. 多Canvas吃CPU资源