NGUI 优化技巧

一、背景介绍

随着商业引擎的发展，游戏开发的门槛越来越低，在只考虑功能的情况下，开发出来一款游戏的难度并不高。但是，现在的游戏对画面和游戏性能的要求越来越高，而某种意义上这两个名词又是相互背驰的。所以，想要达到高标准的要求，一个是要使用更精致的算法去实现更绚丽的效果，另外一个就是要对游戏进行无止境的优化。在这篇文章中，我们先谈优化。

想要做好优化，首先需要了解游戏在设备上运行的本质。当游戏打开的时候，我们会根据 mesh 等顶点信息之类，在 CPU 做一定的计算，然后把所需要的纹理、各种计算后的顶点信息，通过总线传给 GPU，然后在 GPU 中再经过一系列的计算，保存在一张名叫 framebuffer 的内存块中，将这块内存块传给显示器显示出来，最终，我们就看到了画面。所以，在游戏的运行过程中，用到了 CPU、GPU、内存、总线等组件。

我们都知道木桶定律，一只水桶能装多少水取决于它最短的那块木板。任何一个组件，都有可能成为瓶颈，限制了游戏的整体性能。所以，我们需要从各个方面去优化，比如：通过尽可能避免合并 mesh 操作的方式，减少 CPU 的计算量；通过减少 Draw Call 的方式，减少 GPU 的计算量；合理分配计算的位置，是放在 Shader 中还是放在 CPU 中，平衡 CPU 和 GPU 的消耗；及时的进行内存释放，以防内存泄漏；不要经常的通过 glTexImage2D 等操作往 GPU 上传 texture 等，以防占用太多的总线带宽，而过多的使用总线，就是手机发烫的元凶。

NGUI 是为 Unity 设计的，Unity 主要是用于开发手游，定位做手游，就决定了这个游戏一定要平衡好画面、性能、功耗和发烫等问题，而不能像做端游那样，无视耗电、发烫等问题。而且经过研究表明，UI 的消耗已经占据了手游各个模块性能消耗的第二位，UI 效率比较低是各个项目组经常会遇到的问题。所以，对 UI 的优化非常重要。

本节我们来聊一下如何对 NGUI 进行性能优化，以及使用 NGUI 进行 UI 开发的一些小 tips。

二、NGUI vs UGUI

目前市面上唱主调的是 NGUI 和 UGUI 这两种，NGUI 是第三方工程师独立开发的，而 UGUI 是 Unity 官方提供的。NGUI 已经存在了很久，而 UGUI 还很年轻，从 5.x 之后，Unity 才大力推广自己的这一套 UI。下面从使用率、易用性和性能来对这两种 UI 系统进行分析。

使用率：目前还是使用 NGUI 的开发者比较多，毕竟 NGUI 久经沙场，成熟度比较高，教程和相关插件也会比较多。所以，项目组如果选择继续使用 NGUI 并没有什么问题，毕竟使用者还是很多的。

易用性：UGUI 是 Unity 本身的一套原生框架，虽然目前在易用性上来说 NGUI 要占一定的优势，但是 UGUI 现在也在改善这一块，所以 NGUI 在这一块并没有特别有优势。

性能：不管是战斗时候大量的 HUD，还是主界面上有一大堆 UI 元素，这两个框架只要使用合理，都可以做出性能不错的 UI 界面。但是如果使用的时候不注意一些细节，这两个 UI 框架都是会出现一些比较严重的性能问题。相对来说，UGUI 毕竟是属于原生的，所以一部分的开销是可以在 native 的代码里面来做，比如在 5.2、5.3 之后 UGUI 是有一部分网格合并的操作是放在多线程里面，所以相对来说，UGUI 在性能上还是有一定优势的。但是 NGUI 也有一定比 UGUI 占优势的地方，就是 NGUI 的优化相对更简单一些。NGUI 可以非常容易的查看 draw call 信息，得知 draw call 是由什么产生的；网格的更新也可以通过一定的方式进行准确的定位，得知网格的刷新和重建是什么元素引起，以及每次的重建开销是多少。但是对于 UGUI 来说，draw call 比较难以估计，而且由于它的网格重建引入了多线程，所以其开销也隐藏了起来。所以说，如果使用 UGUI 的时候性能不好，优化起来，暂时还是比优化 NGUI 难度要高一些。

总的来说，长久来看，还是比较推荐 UGUI，但是就目前来看，这两款 UI 并没有特别明显的差距。

下面，将从浅到深讲一下 NGUI 中比较容易产生困惑的几个问题。

三、DrawCall 的优化

减少 DC 是个永恒的话题。因为一般来说，特别是静态页面，DC 越少越好。当然万事皆有反例，一会我们再说什么时候不能节省 DC。

NGUI 提供两个工具，Panel Tool 和 Draw Call Tool，可以看到当前界面每个 UI panel 产生多少 DC。在 Panel Tool 中可以看到有多少 panel 以及每个 panel 使用多少 DC，然后我们可以选择 DC 比较高的 panel 进行优化。选择之后，在 Draw Call Tool 中选择 Selected Panel，然后就可以看到当前 panel 使用的所有 DC 信息。比如每个 DC 来自哪个 Atlas，以及渲染顺序等。

优化的时候可以从以下几个方面进行排查：

1. 查看相邻的 DC，是否使用相同的 Material，如果是的话，说明使用的是 UITexture，而非 UISprite。因为 UISprite 在这种情况下会自动合并，而 UITexture，一般一个 UITexture 对应一个 Draw Call（除非是两个完全一样的 UITexture 也可以对应一个 DC），所以为了减少 DC，我们要尽量把 UITexture 转换成 Atlas。首先，用 UITexture 的原因大部分是因为需要用到自定义材质，为了图方便就直接使用了一个 UITexture，然而其实把它打成一个 Atlas 也很方便，新的 NGUI 的版本甚至可以自动做替换。可以通过 NGUI 的 Atlas Maker，选中所需要的 texture，就可以创建一个新的 Atlas，创建好之后，可以看到使用的 UITexture 都会被自动转为 UISprite，但是效果是不对的，因为 Atlas 的 prefab 使用的材质是默认创建的一个材质，而没有使用原来用的材质，所以需要将 Atlas 使用的材质的 Shader 替换成原来的自定义的 Shader 即可。这个时候再看下 DC Tool 就会发现 DC 变少了。
2. 需要关注的就是 Atlas 的在渲染顺序中的交替出现，如果出现这种情况的话，就说明深度设置并不合理。这个时候可以通过 DC Tool 定位到对应的 UI 元素，然后通过调整这些 UI 元素深度的方法，把可以合并的尽量合并起来（只要不影响真正的遮挡关系即可）。深度调节是在 UI 元素的 inspect 中调整 Widget 的 Depth 值。调节完之后就可以看到 DC tool 中前一个 DC 中就会增加一些 UI 元素，后一个 DC 中就会减少一些 UI 元素。

其实在搭 UI 界面的时候，就可以尽量避免 DC 浪费。比如背景层放在哪个

深度下，icon 层放在哪个深度下，如果制作过程就有这个规范，那么实际运行的时候就会发现 DC 比较少。这一点在 UGUI 就简单很多，因为 UGUI 会自己调整渲染顺序，即使结构不对，也会做 UI DC 合并。但是 NGUI 就要非常合理的设置深度值。

总的来说，如果在渲染报告中，如果看到半透明部分的开销比较大，那么就可以考虑优化 UI 的 DC。如果本身就不高，或者基本就是粒子系统造成的，那么就不太需要优化。

四、更新的开销

我们知道，游戏是根据玩家的输入，或者游戏的设定，使得画面中的内容不停进行调整，然后一张张的按照 FPS 不停的播放显示出来的。所以，假如说游戏画面没有变化，那么也就基本没有消耗，只需要显示一张静态图片即可，而消耗，主要就是发生在画面更新的时候产生的。我们来看一下，UI 界面更新的时候会发生什么。

我们在研究 Draw Call 的时候，可以修改一下代码 UIDrawCall.cs 文件，将宏 SHOW_HIDDEN_OBJECTS 打开，然后我们就可以在 hierarchy 中看到几个新的 GameObject，这些 GameObject，每个都对应了一个 UIDrawCall，而每个 UIDrawCall 都会对应一个 mesh。所以在绘制 UI 的时候，其实就是将若干 UI 元素先进行合并 mesh，然后再统一进行绘制。那么我们可以想象到，如果 UI 元素发生变化，比如消失、显现，比如放大缩小，都会影响到 mesh，那么进而也会触发网格更新的操作。

网格更新基本上会影响到所有的组件：

1. 会在 CPU 中进行计算。实际上 UI 在 CPU 中的消耗，主要都是由于这一块。
2. 将计算出来的数据通过总线传给 GPU，然后在 GPU 中形成新的 VBO。我们知道基本所有渲染都包含在 Camera.Render 函数中，比如不透明渲染、半透明渲染等。不透明和半透明渲染又包含了很多情况，比如场景的静态物件、蒙皮人物等。在 NGUI 中渲染是通过 MeshRenderer.Render 函数，是在 transparent 下。包含 DrawVBO 和 CreateVBO 两个函数。其中

CreateVBO 中基本都是被 UI 占用，一旦调用了 CreateVBO，那么就说明这个网格在当前帧发生了变化，进行了更新。对于常见的透明物体、模型，基本很少在每帧中出现 createVBO 的开销，因为这些网格基本都是静态的。而对于 UI，一旦出现了 NGUI panel 的刷新，或者某些 UI 元素的变动，导致底层网格的变化，都会触发 createVBO 函数的执行和开销，所以如果 createVBO 开销比较大，可能就说明了 UI 网格的刷新过于频繁。

3. 占用一部分内存。NGUI 对内存分配也主要是 UIPanel.LateUpdate 这个函数造成，并且这个函数基本也只是在完全重建时才会对内存出现比较高的开销，**一般如果堆内存中总分布在 20M 以内，还算比较合理**。基本上一旦出现内存分配，就说明 UIPanel 本身进行了刷新。所以如果这个内存越高，说明 UI 刷新的频率越高，以及所涉及的 UI 元素的数量越大。

网格更新的场景非常多，比如在战斗的时候，UI 元素非常多，各种连击 UI 的显示和消失等操作，都会导致网格刷新。在进入战斗之前，UIPanel.LateUpdate 可能只有 3、4ms，而战斗提示之后，经常会出现 10+ms 的小峰值，影响到游戏的流畅性。实际上，当 **UIPanel.LateUpdate 耗时超过 3-4ms，基本可以认为这一块的开销是比较高的**。可能其中存在一些比较浪费的重建。理论上 UI 更新的数量和频率是可以通过合理布局来将开销进行限制。一般来说对于合理的 NGUI 的 UI 模块，在关卡切换，或者点击一些 UI 界面时，是可以出现一些比较高的峰值，这个很正常，但是在战斗部分等，是要尽可能保证耗时在一个很小范围能波动。

以上，就是总结了一下，如果我们看到哪些现象，或者从数据看出哪些数值的时候，可以宏观的判断项目的 UI 是否合理。那么如果发现不合理，那么需要从 NGUI 的更新开销下手进行优化。

想要对网格更新进行优化，需要做两件事情，第一件事情就是了解这些网格的由来，第二件事情就是了解何种情况会触发网格更新，能否尽可能的避免或者减少网格更新的消耗。

先说这些网格的由来，这些网格其实就是 UI 元素的网格，我们可以举例说两个 UI 元素的网格，UILabel 和 UISprite。

从 UI 元素 **UILabel** 说起：

1. 最主要的还是 label 本身的字数。字数越少，网格也就越少。如果 label

是移动的话，那么字数越少，带来的更新开销越小。

2. 文本特效。比如阴影、outline。NGUI 有 outline 和 outline8，UGUI 只有 outline。如果选择 outline，那么会把字体左上移左下移右上移右下移，拼成一个外框，如果选择 outline8，就会往 8 个方向移动，拼成一个外轮廓线。当然，如果选择 outline8，外轮廓线会更加光滑，效果更好一些，如果 outline，可能会有些地方不太光滑。但是，Outline8 的使用代价更高一些，使用 outline，网格数量会增加 4 倍，而 outline8，会增加 8 倍。如果这些文字又是在每帧不停的移动，进行更新，那么由于网格更新，对耗时会有一个比较明显的影响。经过测试显示，如果文本都在移动，对 NGUI 来说每帧网格都在更新，开启 shadow 的话，耗时会提高 20-30%，然而如果使用 outline8，耗时会翻 3-4 倍甚至更多。所以虽然 outline8 会使得文本外观好看很多，但是对于频繁移动的文本来说，性能的开销是非常大的。所以对频繁移动或者变颜色的动态文本尽可能少用 outline。而对于静态文本由于不会每帧去更新网格，所以没有关系，可以放心使用。
3. 额外说一点 Label 中由于更新产生的其他消耗，动态纹理的重建。如果在 Unity 中，使用 Unity 的动态字体。那么在改变文字，或者频繁的让一些新的文本出现的时候，在 profile 会看到 Font.CacheFontForText。因为动态字体的生成是由 Unity 动态生成一张纹理，相当于随着文字首次出现，将用到的文字合在纹理中，从而显示到屏幕上。这个纹理可以从 label 的字体的材质中看到。可以看到这个纹理从最初的 256*256 alpha8，随着文字慢慢变多，只要文本字体或者内容不一样，在动态纹理中就会增加一些内容，进而改变纹理的尺寸，变成 256*512，甚至 512*512。如果纹理尺寸不够了，那么就会刷新一下，根据当前能够创建的纹理大小的限制，而在 Editor 中纹理大小可以开的比较大，如果在手机上，纹理大小会有所限制，如果空间不够了，只能创建一个小的纹理，那么就会把激活状态的文字就会被填入纹理，而那些出现过的，但是却被禁用的文字就会被剔除出去，不停的变化文字，纹理内容就不停更新，所以经常会出现纹理的内容不停变动，导致纹理不停刷新，也就是

Font.CacheFontForText 这个函数不停的被触发。且完全更新了纹理中的内容，这个操作很耗时。这个函数可能会消耗 20ms 左右，所以对于出现文本比较大，如果它是首次出现，且使用的是动态字体，通常会产生一个比较高的耗时。所以建议：如果反复使用的文本，尽可能使用静态字体，使用定义好的纹理，对于静态字体而言，就不会去刷新动态字体对应的动态文本了。这样就可以避免 Font.CacheFontForText 函数的调用。如果对于本身只出现一次，且文字非常多的情况，就很难避免了。如果不希望在文字出现的时候出现耗时上的峰值，那么就只有让它预先出现，比如创建一个 label，其中包含这个文字，处于激活状态，只是相机看不到它，这样就把峰值移动到了接受卡顿的地方（比如场景切换读条的时候），然后在显示文字的时候，再把这个 label 逐渐显示出来，或者移动或者通过修改 layer 的方式让它更快更流畅的显示出来。

再说一下 UI 元素 **UISprite**：

跟 UILabel 比起来，开销不会特别大，但是有一个选项会比较影响性能，也就是 UISprite 的显示模式 type，默认为 simple，选了 simple 的话，场景中可以看到每个 sprite 都是一个四边形，跟文本比起来，sprite 包含的网格数量非常少，开销也会非常小。但是如果把 type 模式改成 tiled，为了做平铺，会把一个 sprite 分割成很多四边形，网格数量会随着平铺面积大小的增长，翻倍增长，导致网格更新的开销变大。所以需要谨慎使用 tile 模式。

需要注意的是：以上所说的所有设置都是在 UI 元素本身在移动或者变颜色的时候才会真正产生开销，如果文本或者 sprite 只是静态的放在 UI 面板上，那么实际上不会涉及到更新的开销，那么这些功能也就都可以放心使用了。

了解了网格的构成，下面来了解一下网格在什么时候会被更新，那么首先要了解的就是 **UIPanel 的更新规则**：

在 NGUI 里面 UIPanel 的更新分为两种类型。一种是只更新一个 UIDrawCall，也就是在更新的过程中，某个 UI 元素进行了移动，这个时候只会更新它所在 DC 中的网格，而不会影响其他 DC。第二种方式，就是一旦满足了某种条件，就会直接重建 UIPanel 中所有的 DC，这个开销就会非常大。目前看到实际项目中耗时超过 5ms 以上，通常都是因为这种类型的更新被触发。一般来说，如果一个

UIPanel 中 UI 元素数量越多，那么重建的耗时也就会非常高。

如果所有的 UI 元素，都在一个 DC 中，那么当其中某个顶点发生任何变化，由于顶点与其他所有元素的顶点都在一个 mesh 上，那么整个 mesh 就会进行更新，更新一个大的 mesh 的开销就会很高。

所以应该把经常动的 UI 元素从 DC 中拆分出来。

会导致完全重建 UIPanel 的方式有如下几种：

在 UIPanel 中添加 UI 元素导致把原有 DC 进行了拆分，删除 UI 元素导致原有的 DC 合并，或者只是增加了一个新的 DC 并没有对原有 DC 产生任何影响。

总的来说就是因为某个操作（通常是动态添加或者删除元素）使得 UIPanel 中的 UI DC 发生变化。

比如添加的元素的深度会打断之前的一个合并 atlas 的 DC，那么就会把之前的一个 DC 变成三个，而删除的时候又会重新变回一个。如果添加删除的元素深度并没有刚好影响到其他的 DC，也没有增加新的 DC，就没关系。

重建 DC 的时候，其实就是会把所有的 GameObject deactive 禁用（原始多少个 DC 就有多少个 GameObject），然后再一个个激活（之后会有几个 DC 就要激活几个 GameObject）。同时，这个操作还会触发 createVBO 函数，这个函数很耗时，比 DrawVBO 还要耗时。所以在添加或者删除的瞬间会很卡。

如果添加的 mesh 无法合并到别的 DC 中（因为使用了不一样的字体之类），会生成一个新的 DC，虽然不影响其他的 DC，但是 NGUI 处理的时候，依然会全部进行重建（这不合理，但是 NGUI 目前还是这种设计方式）

优化方式：

动静分离：

第一步，调整深度，把深度调整到不会影响其他合并 DC 的值，并且不会生成新的 DC。这样 DC 不会变。

第二步，尽可能的把 UI 元素所在的 DC 的 mesh 降到很低（类似更新单个 UI DC 的优化）。直接把不相关的 UI 元素分出去，或者把动态的 UI 元素独立成一个 panel。拆分 panel，不同 panel 不能合并 DC，所以以增加 DC 数量的方式降低网格更新的开销，因为网格更新的开销远大于 DC 的开销。这样即使出现了整个 panel 的重建，也只会影响很少的 mesh 数量。

总结一下：

1. 对于频繁变动的 UI 元素，其所在的 UI DC 面片数越少越好
2. 动态添加 UI 元素的时候，首先注意深度设置，不能正好穿插打破原有的 UI DC，特别是跟原有 UI DC 用的并非同一材质，这种情况会产生完全重建。最好是插入一个使用同一材质的 UI DC（只要原有面片不多）
3. 如果一个 Panel 需要经常添加和删除 UI 元素（比如战斗中的连击提示），尽可能减少复杂度，不要把静态的 UI 元素添加在这个 UIPanel 中。否则很难控制其深度和确保原有 Panel 中存在相同材质。这样虽然这个 panel 要经常重建，但是因为这个 panel 很小，所以不会有很大影响。

优化其实不难，难度是定位需要优化的点，比如定位哪个 panel 开销大，或者 panel 的开销当前是否合理，以及在出现峰值的时候是出现在哪里。

比如从 profile 看到在切换场景，点击 UI 的时候出现峰值，这里是优先级比较低的，因为这里的耗时不可避免，但是如果在战斗过程中出现峰值，那么需要重点看，看哪些 UI 元素触发了刚才那些问题。

一般来说，多个动的 UI 元素分成多个 panel 会比较好，这样会少量提高合并网格的计算量，以及 createVBO 会更省时一些，因为 createVBO 的时候，网格数量越大，越耗时，且并非线性增长。

可以想办法动态的把一个 UIPanel 中静态的 UI 元素根据一定的规则（比如几秒钟不动了）归在一个静态的 panel，对于移动的专门放在另外一个 panel 中，起到一个动静分离的作用。也就是动态的去分组。

需要注意的一点：当一个粒子系统穿插在两个 UI 元素之间，而假如这两个 UI 元素的 DC 是可以合并的，那么在开始的时候，DC 是会被打断，之后，哪怕粒子不停的变化，而对 UI 是不会有影响。

下面几个章节，说一些 NGUI 开发中的一些小的 tips，某些也可以略微的提升 NGUI 的性能。

五、屏幕自适应

屏幕自适应上，NGUI 和 UGUI 的方式还是比较接近的。

不管是苹果还是 Android，不同的设备会有不同的宽高比和不同的分辨率，

特别是在 Android 机器上。所以做 UI 的时候，遇到的第一个问题，一般都是要把屏幕的自适应做一个规划，做一个设计。

NGUI 的自适应可以通过几个组件进行简单的规划。

首先，是在 **UIRoot** 上面，会有一个 **Scaling style** 属性，可以整体的去控制整个屏幕的缩放规则。共有两种模式，第一个是 **flexible**，这个模式的意思就是可以让用户来指定一个区间，比如当真实屏幕像素的高度是在 320-1536 之间的话，所有的 UI 元素的分辨率保持不变，也就是说 UI 元素的像素和设备的像素是对应的。也就是说在分辨率很高的设备上，UI 元素会显的比较小，而分辨率很低的设备上，UI 元素又会比较大。这种模式相对来说复杂一些，所以使用第二种模式的会比较多。第二个模式是 **Constrained**，指定一个固定的分辨率，比如指定 1280*720 的分辨率，那么不管设备的分辨率多少，长宽比是多少，UI 元素使用都会以这个比例去显示。所以这样的话，不同的设备上，做的操作就是将 UI 元素的分辨率进行等比缩放，不会出现高分辨率的机器上图标更小的情况。这种模式的布局就会容易一些，这也是目前使用最多的一个整体布局的方式。

如果设备长宽比与 Constrained 模式指定分辨率的长宽比一致的话，那么只要做简单的分辨率缩放即可，然后显示在不同设备上看上去就会完全一样（但是如果设备的尺寸不同，那么看上去还是会有大小之分的，比如 iPad 上的 UI 元素尺寸比 iPhone 上的大）。然而如果长宽比不同的话，在老的 NGUI 版本中，会使用到另外一个组件了，UIAnchor，它可以控制一些元素始终对应左上角或者右上角，但是在新的 NGUI 版本上，这个组件使用的就很少了，因为如果用这个组件做自适应布局，那么 UI 的层级就会非常多，因为通常情况下 UIAnchor 上本身并不会去挂其他的 UI 元素，所以看上去会觉得结构复杂。在新的 NGUI 版本里面，已经将 UIAnchor 中的一些信息，放在 UIRect 这个类里面，这样的话就可以直接对一些 UI 元素进行对准。在 UI 元素中，使用的是 UIRect 的子类 UIWidget，在每个 **UIWidget** 下面都可以看到 **Anchors** 的这样一个设置。可以将其设置为 None，但是也可以通过一些设置，将其上中下对应到一些锚点上，然后它就可以根据屏幕上的比例，对应到各自的位置上。然而 Anchor 的设置也是有一定讲究的，**合适的设置可以在性能提升上有一定帮助**，在 Anchors 的第二个属性 Execute 设置的是 Anchor 的执行方式，执行模式分为三种，On Enable，On Update，On Start。

其中 On Enable 和 On Start 对应的执行频率很低，执行次数会很少。也就是这个 UI 元素的锚点对准操作，只会发生在 Start 的时候，或者 setActive 的时候。然而如果设置的是 On Update，那么每一帧都会根据设置好的几个锚点信息做对准操作。一般来说，对于静态的 UI，这个属性应该设置为 On Enable 或者 On Start，这样节省一些性能消耗。如果 UI 元素中 On Update 设置的比较多的话，那么在 profile 中就会发现有一项叫做 **UIRect.Update** 会特别高。所以如果看到这一项，那么就会需要去排查一下是否是 On Update 设置的比较多。

以上就是一种比较简单的屏幕自适应的方法，基本上可以满足大多数屏幕自适应的需求。

六、事件处理

在事件处理上，NGUI 和 UGUI 相差很大。因为在 UGUI 中的事件处理没有使用物理上的 Recast，而是通过 RectTransformUtility 里面的一些接口，它的检测做的会相对简单一些，是通过射线和四边形来进行检测。而 NGUI 会借用物理系统，会使用包括 PhysX 和 Phyx2D。也很难说到底哪种更高效，但是一般来说检测事件的消耗都不会很高(需要注意的是：在 4.X 上 UGUI 的事件开销有时候会比较大，因为所有的 graphic UI 元素都会作为事件检测的目标，甚至比如 image 或者 text，除非直接把 Canvas 禁用掉，而在 5.2 之后，就给每个 graphic UI 元素增加了 Recast Target 的属性，去掉之后才会不参与检测)。在 NGUI 中，只有比如 Button 等这些需要检测的元素上，才会挂上碰撞体，所以在检测的时候，也只有挂了碰撞体的元素才会参与检测。

举个例子，如果对 UIButton 做事件处理。那么事件处理的代码经常会被直接放在 UIButton 这个组件下面，这样会导致事件处理的代码分的比较散。比如每个 Button 对应一个逻辑，这样管理起来会比较复杂。一般来说，事件检测方式需要借助 UIEventListener，这样一个组件的作用是，可以在 Button 或者检测事件的 UI 元素上挂载这个组件，然后在一个统一的地方去处理这些事件。（比如创建一个 EventManager，将其挂在 UIRoot 上，然后抓取 UIRoot 下面所有 Button，调用 EventListener 的 Get 函数把所有 Button 挂在一个 EventListener，然后设置 EventListener 的 OnClick 委托，不过需要确认挂上 Button Script 的 UI 元素同时也

挂在了 Collider）

七、自定义材质

自定义材质在 NGUI 还是很常见的。比如用户要做一个会变灰的按钮，那么写一个变灰的 Shader，放到 UI 元素上，是没有问题的。但是如果把这个 Shader 挂载到 ScrollRect 或者 ScrollView 里面，也就是滚动区域里面，就会发现好像材质丢了变黑了等问题。这个问题的原因是 NGUI 在处理滚动框里面的内容的时候，为了实现类似遮罩的效果，是会动态替换里面元素的 Shader，所以需要了解这个替换规则，才能正确是使用自定义材质。

用一个例子来解释：

我们知道针对带 alpha 通道的纹理，由于 ETC1 不支持带 alpha 通道的图片，所以一般选择 ETC2。然而低端 Android 手机不支持 ETC2，所以 ETC2 的纹理直接会翻四倍。如果想用透明纹理，则需要把纹理拆成 RGB 和 Alpha 这两 ETC1 图片，然后通过两次纹理采样，得到一个透明的图片。

那么如果想使用在 ScrollRect 上使用透明贴图，那么需要采两次，所以要替换 NGUI 的材质。那么我们可以先创建一个普通的 Shader，然后使用在 UI 元素上。但是如果被一个 panel 做了 clipping，那么就会使用 Shader 名字加 1 的 Shader，如果是两个，则名字加 2，如果是三个，则名字加 3。一般不会有四个 panel 做 clipping，所以 NGUI 的作者提供了三个可替换机制。

通过这个例子可以看到，当我们要提供自定义 NGUI 材质的时候，需要附带至少三个配套版本（为了保证 UI 元素放到滚动区域，也可以正常显示）。

八、NGUI 的 UI 元素与 Particle System 进行交互

在交互过程中也会遇到一些问题。

比如：如何把一个 Particle System 放在两个 UI 元素之间，这个还比较容易，因为 NGUI 元素本身就是通过 Render Queue 控制渲染顺序，只需要通过脚本把 Particle System 的 renderer 所对应的 Render Queue 控制在两个元素之间即可。

再比如：当 Particle System 被放到了滚动区域（比如背包中的一些图标需要高亮或者一些特殊效果，需要在周围画一圈粒子），这样在滚动背包的时候，最

理想的是背包的区域会把粒子裁减掉，但是默认情况下并不会发生这种现象，因为毕竟粒子系统的 Shader 是完全独立的渲染方式，跟 NGUI 的遮挡和裁剪是完全不一样的。因为 NGUI 不仅自身有 UI 的 Shader，同时还要替换适合做 Clip 的 Shader 版本（如第五章所述）。所以这种需求会难一点，解决方式有如下几种：

1. 转成序列帧，还是做成 UI 的方式来做，这样全部都是 NGUI 的元素，但是问题也很大，表现力会大打折扣，而且会增加图片量。
2. 设计自定义的 Particle System 的 Shader，并将 NGUI 的裁剪机制放到自定义 Shader 中，这样会复杂一些（Particle System 的坐标系和 UI 产生的 mesh 的坐标系的对应有一些复杂）。
3. 通过一个额外的 Camera 来做，也就是把 Particle System 的裁剪通过 Camera 的显示区域来做。这个限制也比较大，因为多加了一个 Camera，那么 UI 的层级管理就会变得比较复杂。

以解决方案 2 为例来说：让 Particle System 直接使用 NGUI 自带的 Shader，因为 NGUI 实际上使用的就是 Alpha Blend，所以我们真正要解决的如何通过一定的计算把 Particle System 的裁剪区域换算出来。

首先，先挂载一个 Particle System，可以看到默认 Particle System 的粒子是会显示在 UI 元素之后的，创建一个 UIParticle 的空脚本，先将该脚本挂载到 Particle System 上。

然后在 UIParticle 脚本的 Start 函数中创建一个新的材质，然后使用 NGUI 的裁剪材质（因为 NGUI 的材质实际上只有一个 Alpha Blend，所以显示 Particle 没问题），以及原材质的纹理和一个新的 RenderQueue。

这个材质在脚本 OnDestroy 函数中一定要删除掉。在 Editor 里面会导致由于不停的执行 Start，所以不停的实例化，虽然 Editor 也会去释放，但是会在释放的时候报一个错误。而在运行的时候，则会出现内存泄漏，一直等到切换关卡，或者调用 OnLoad 或者 OnUseAsset 的时候这个材质才会被丢掉。

还有一个问题，如果在这里不新创建一个新的材质，而只是修改 renderer 材质的 RenderQueue，这个时候 Unity 会动态创建一个新的材质，这样的话这个材质就不太容易被拿到，也无法判断当前 renderer 的 material 是新创建的材质还是原始材质，然后在删除的时候，就不知道是否要删除 renderer 下的材质，因为

原始材质就不应该被删除，如果是新的材质，就要被删除。

在这里，我们知道，我们肯定会去修改这个材质，那么就手动新创建一个就好，便于管理。NGUI 中基本都是这么操作的，每个 UI DrawCall 都会保留一个动态材质变量，从而在滚动区域的时候，就可以把动态材质改成标记为 1 的 Shader，然而在销毁的时候再把动态材质删掉，shared material 是公用的，不用管。

UIDrawCall 的代码是用来管理 UI 元素的显示的，包含在渲染前任何材质的替换以及裁剪区域的设定等。其中一个重要的回调函数就是 OnWillRenderObject，是物件或者 mesh 真正在被渲染前要走的逻辑，在这个逻辑中可以看到这个 UIDrawcall 对应的 panel 是否是纹理所对应的 clip。然后判断 panel 是否开了裁剪，如果开了，就拿到 panel 的裁剪区域，这个裁剪区域在 editor 里面是有设置的（offset、center、size、softness 渐变）。然后根据这些参数算出裁剪区域根据裁剪区域的数量进行多次 setclip。

在 Shader 中也就可以看到，拿到刚才算出来的裁剪区域算出 UI 元素对应的 mesh 的世界坐标（添加了偏移计算），可以认为这里就是把 UI 元素的世界坐标对应到 panel 的裁剪区域里面，偏移完毕之后就可以拿世界坐标和（1，1）做对比，以及渐变参数来计算最终的像素点的 alpha。区域外的 alpha 全部归零。

将 UIDrawCall 代码中的 show hidden object 打开，就会看到几个对应 UI DrawCall 的 GameObject，其中的 mesh 就是 UI 元素，也就是说真正渲染的还是 mesh。可以看到这些 mesh 的顶点都是以 panel 作为坐标系，所以如果要裁剪粒子系统，那么就要把粒子系统的坐标系转为 panel 的坐标系。

回到 UIParticle 的代码，加上 OnWillRenderObject 函数，粒子系统产生的 mesh 不会被缩放，所以转换到 panel 坐标系的时候，需要先将缩放记录下来，然后再进行位移转换。

不过这样的话，性能会略微有点浪费的，因为在每次 render 之前都会计算裁剪区域，因为如果 panel 不动，这块计算可以放到初始化的时候做。

本节教程就到此结束，希望大家继续阅读我之后的教程。

谢谢大家，再见！