移动平台的特点：

和 PC 平台相比，移动平台上的 GPU 架构有很大的不同。由于处理资源等条件的限制，移动设备上的 GPU 架构专注于尽可能使用更小的带宽和功能，也由此带来了许多和 PC 平台完全不同的现象。

例如，为了尽可能移除那些隐藏的表面，减少 overdraw（即一个像素被绘制多次），PowerVR 芯片（通常用于 IOS 设备和某些 Andriod 设备）使用了基于瓦片的延迟渲染（Tiled-based Deferred Rendering，TBDR）架构，把所有的渲染图像装入下一个瓦片中，再由硬件找到可见的片元，而只有这些可见片元才会执行片元着色器。另一些基于瓦片的 GPU 架构，如 Adreno（高通的芯片）和 Mali（ARM的芯片）则会使用 Early-Z 或相似的技术进行一个低精度的深度检测，来剔除那些不需要渲染的片元。还有一些 GPU，如 Tegra（英伟达的芯片），则使用了传统的架构设计，因此在这些设备上，overdraw 更可能造成性能的瓶颈。由于这些芯片架构造成的不同，一些游戏往往需要针对不同的芯片发布不同的版本，以便对每个芯片进行更有针对性的优化。尤其是在 Android 平台上，不同设备使用的硬件，如图形芯片、屏幕分辨率等，大相径庭，这对图形优化提出了更高的挑战。相比于 Android 平台，IOS 平台的硬件条件则相对统一。

影响性能的因素

CPU主要负责保证帧率，GPU主要负责分辨率相关的一些处理。

造成游戏性能瓶颈的主要原因：

1.CPU

过多的 draw call

复杂的脚本或者物理模拟

2.GPU

顶点处理

过多的顶点

过多的逐顶点计算

片元处理

过多的片元（既有可能是由于分辨率造成的，也可能是 overdraw 造成的）

过多的逐片元计算

3.带宽

使用了尺寸很大且未压缩的纹理

分辨率过高的帧缓存

对于 CPU ：

CPU 在每次通知 GPU 进行渲染之前，都需要：

1.提前准备好顶点数据（如位置、发现、颜色、纹理坐标等）

2.调用一系列 API 把它们都放到 GPU 可以访问到的指定位置

3.调用一个绘制命令，通知 GPU 渲染，

这样一次调用一个绘制命令，就会产生一个 draw call。过多的 draw call 会造成 CPU 的性能瓶颈，这是因为每次调用 draw call时，CPU 往往都需要改变很多渲染状态的设置，而这些操作是非常耗时的。如果一帧中需要的 draw call 数目过多的话，就会导致 CPU 把大多数时间都花费在提交 draw call 的工作上面了。还要物理、布料模拟、蒙皮、粒子模拟等都可能造成 CPU 瓶颈，这些都是计算量很大的操作。

对于GPU：

它负责整个渲染流水线。从处理 CPU 传递过来的模型数据开始，进行顶点着色器、片元着色器等一些列工作，最后输出屏幕上的每个像素。因此，GPU 性能瓶颈和需要处理的顶点数目、屏幕分辨率、显存等因素有关。而相关的优化策略可以从减少处理的数据规模（包括顶点数目和片元数目）、减少运算复杂度等方面入手。

由上面可以得知相关的优化技术：

1.CPU 优化

使用批处理技术减少 draw call 数目

2.GPU 优化

减少需要处理的顶点数目

优化几何体

使用模型的 LOD（Level of Detail） 技术

使用遮挡剔除技术（Occlusion Culling）

减少需要处理的片元数目

控制绘制顺序

警惕透明物体

减少实时光照

减少计算复杂度

使用 Shader 的 LOD（Level of Detail）技术

代码方面的优化

3.节省内存带宽

减少纹理大小

利用分辨率缩放

Unity中的渲染分析工具

渲染统计窗口

文本

描述已自动生成

CPU：主线程花费了31.7ms，渲染线程花费了29.2ms。

Batches：总共有30,003批次。

Saved by batching：分批处理节省的批次为0。

Shadow casters：图包含了10,000个点。

SetPass calls：使用6个Pass。渲染6次（所以每个点都被渲染了三次。一次用于深度测试，一次用于阴影投射(也分别列出)，还有一次用于渲染最终的立方体。其他三个批处理用于额外的工作，如天空框和阴影处理，它们独立于我们的图。还有6个set-pass calls，这可以被认为是GPU重新配置以不同的方式渲染，比如使用不同的材料。）

Tris：361.7k个面片

Verts：725k个顶点

Visible skinned meshes：渲染的蒙皮网格数目为0

Animation components playing：播放动画数目为0

性能分析器：

Window->Analysis->Profiler

日程表

描述已自动生成

显示了绝大部分在Statistics中提供的信息，还有draw call 数目、动态批处理／静态批处理的数目、渲染纹理的数目和内存占用。绿线是批数，深蓝线是Pass数、红色是面片数、浅蓝是顶点数

帧调试器：

Window->Analysis->FrameDebugger

文本

描述已自动生成

可以看到每一个 draw call 工作和结果，在左边的面板上，拖动整个面板上方的滑动条可以看每一帧里渲染的东西（把之前渲染好的表面和当前渲染的全过程，每一帧时调用draw call完成的画面）

优化部分：

1.减少Draw Call数目：

什么样的物体可以合批一起处理？使用同一材质的物体，因为使用同一材质的物体间的不同仅在于顶点数据的差别，因此可以把这些顶点数据合并在一起，再一起发给GPU完成一次批处理。

Unity支持的批处理方式：静态批处理、动态批处理

动态批处理和静态批处理的不同：

1、动态批处理：

优缺点：Unity 自动完成的，物体是可以移动的，但是限制很多，进行统一绘制。

限制：如网格的顶点属性规模要小于900，顶点数目不超过300；多Pass 的shader会中断批处理，即不会再使用动态批处理

2、静态批处理：（尽可能选择）

优缺点：自由度高，限制很少，但可能会占用更多的内存，且物体不可以再移动，即只运行在开始阶段，因此只进行一次合并操作。本质上是牺牲内存换取性能，VBO更大了：把静态物体变换到世界空间下，然后为它们构建一个更大的顶点和索引缓存，如果在静态批处理前一些物体共享了相同的网格，那么在内存中每一个物体都会对应一个该网格的复制，即一个网格会变成多个网格发送给CPU(一个模型使用了一千次，那么就会消耗一千倍的内存)。对于使用了同材质的物体统一绘制，不同材质的物体，静态批处理可以减少这些 draw call 之间的状态切换；处理平行光的BasePass部分会被静态批处理，但是处理其他光源的额外Pass不会。

共享材质（有助于进行批处理，因为批处理要求同一材质）

1、如果两个材质之间只有使用的纹理不同

  把这些纹理合并到一张更大的纹理中，这样的纹理被称为一张图集，使用同一个材质，再使用不同的采样坐标对纹理采样即可

2、不同的物体在材质上还有一些微小的参数变化

  一个例子是， 森林场景中所有的树使用了同一种材质， 我们希望它们可以通过批处理来减少draw call, 但不同树的颜色可能不同。 这时可以利用网格的顶点的颜色数据来调整。也就是单独调整一个模型中的Material而不是调整共享材质。

减少需要处理的顶点数目

1.优化几何体

建模时减少面数，移除不必要的顶点。移除不必要的硬边以及纹理衔接 避免边界平滑和纹理分离（Unity把一个点分为多个点是为了分离纹理坐标和产生平滑边界，因为同一个点在不同面上的纹理坐标可能不相同）。

2.使用模型的 LOD ( Level of Detail) 技术

LOD允许当对象逐渐远离摄像机时，减少模型上的面片数量，从而提高性能(与MipMap相似，不过LOD用于模型资源，而MipMap用于纹理)。Unity中使用LOD Group组件，对同一个物体用不同细节程度的模型赋给组件中的不同等级， Unity 就会自动判断当前位置上需要使用哪个等级的模型。

3.使用遮挡剔除 (Occlusion Culling) 技术

对比视锥体剔除：应用在程序阶段CPU端，剔除掉那些不在摄像机的视野范围内的对象，遮挡剔除可以直接去掉多余的顶点数目，因此CPU提交更少的draw call给GPU

减少需要处理的片元数目

1.控制绘制顺序

让物体从前往后绘制，可以很大程度上减少overdraw，由于深度测试的存在后面的物体都无法通过深度测试。可以尽可能地把物体的队列设置为不透明物体的渲染队列（Pass小于2500），对于不透明的物体是从前往后绘制的，而尽量避免使用半透明队列，因为为了得到正确的渲染结果，半透明物体必须从后往前渲染。

2.警惕透明物体

对于半透明的对象，造成overdraw是必然的。若场景中包含了大面积的半透明对象，或者有很多层相互覆盖的半透明对象（即便它们每个的面积可能都不大），或者是透明的粒子效果，在移动设备上也会造成大量的 overdraw。这是应该尽量避免的。透明度测试也会影响游戏性能，因为可能会使得一些硬件优化策略失效。

3.减少实时光照和阴影

对逐像素的光源来说，被光源照的物体需要被再渲染一次，而且这种额外的处理逐像素光源的 Pass 都无法进行批处理，会中断批处理。

模拟光源的方法：烘焙技术(把光照提前烘焙到一张光照纹理 lightmap)、 God Ray。

使用烘焙把静态物体的阴影信息存储到光照纹理中，而只对场景中的动态物体使用适当的实时阴影

节省内存带宽

1.减少纹理大小

尽可能使用多级渐远纹理技术 (mipmapping) 和纹理压缩，把纹理类型设置为 Advanced, 勾选Generate MipMaps（GUI纹理除外）就会构建一个纹理金字塔，这样就会在更远的位置使用分辨率低的纹理。

2.利用分辨率缩放

对特定机型降低分辨率。

减少计算复杂度：

1.使用 Shader LOD (Level of Detail) 技术

与模型的 LOO 技术类似，控制使用的 Shader 等级；Shader的 LOD值小于某个设定的值，这个Shader才会被使用（在SubShader中加入类似“LOD 200”之类的代码），比如LOD 200仅有漫反射，LOD 400才有漫反射和镜面反射。

2.代码方面的优化

通常游戏需要计算的对象、顶点、像素数目排序是对象数＜顶点数＜像素数, 尽可能地把计算放在每个对象或逐顶点上(如实现高斯模糊和边缘检测时,把采样坐标的计算放在了顶点着色器中)

使用低精度的浮点值进行运算，高精度的float/high用于存储顶点坐标等变量, half/mediump用于存储一些标量、纹理坐标等变量，（计算速度大约是float的两倍），fixed/lowp适用于绝大多数颜色变量和归一化后的方向矢量（计算速度大约是 float的4倍)。用插值寄存器把数据从顶点着色器传递给下一个阶段时，使用尽可能少的插值变量，例如对两个纹理坐标进行插值，使用float4类型的变量，对应了xy和zw分量。尽可能不要使用全屏的屏幕后处理效果，如果必须要使用也尽量使用fixed/lowp进行低精度运算，而纹理坐标使用half/medium计算。高精度运算可以使用查找表的方式或者转移到顶点着色器中进行处理。并且尽量把多个特效合并到一个Shader中。