**Performance Tunning for Tile-Based Architecture**

**Tile-Based架构下的性能调校**

**by Bruce Merry**

**GameKnife译**

译序

在大概1个月之前，花了两个小时的时间阅读了OpenGL Insights上的两篇关于移动平台GPU的优化文章。当时正巧在公司作移动渲染器的优化和整理，顿觉醍醐灌顶。同时搜索国内关于这方面的经验文正或者翻译文章，感觉少之又少。所以，萌发了翻译这两篇文章的想法。其实也是第一次做翻译工作，本觉读两篇文章就用两个小时，翻译大概一天就够了把。 结果第一天写了一下午，也就翻译出了一个开头，发现这不是一件容易事，因此干脆就慢慢来翻译。

终于，一个月之后，总算翻译到了最后一章。全文翻译下来，才发现第一次阅读其实读漏，读错了很多东西，同时，在开发实践的过程中，又对文中的很多章节有了更深刻的理解。因此在某些地方加上了自己的译注。

最后，把这篇译文分享出来，希望能给在移动GPU平台开发图形程序的朋友一些帮助。

引言

OPENGL 和 OPENGL ES标准描述了一种虚拟渲染管线，这个管线将三角形的处理规定为如下流程：

1. 变换三角形的顶点
2. 对变换后的顶点进行光栅化，生成像素
3. 对像素进行着色，写回帧缓存(framebuffer)

然后按照这个流程进行下一个三角形的处理，如此往复。然而，这对GPU的工作来说，并不是一个高效的流程，GPU会经常的重排和并行化这些过程以达到更高的效率。

在这篇文章中，我们会详细阐释tiled-based渲染方法，tiled-based架构在主流的移动平台GPU上是一个通常的流行图形管线组织方式。我们会关注：什么是tiled-based渲染，为什么要用它，以及我们需要做些什么（注: 相对于传统的立即渲染方式）来达到效率的优化。

我们假设阅读者是这样的人：

* 已经拥有优化OPEN GL程序的经验，熟悉标准优化技术，例如减少状态切换，减少DRAWCALL，降低shader复杂度以及纹理压缩。
* 想要得到关于tiled-based GPU上的优化建议。

有一点，需要牢记于心：每一个GPU, 每一个图形驱动， 每一个程序都是不同的，具有不同的性能特征。基本上，性能调校是一个不断分析和实验的过程。因此，这个本文几乎不会提供“立竿见影”的金科玉律，但是会尝试阐释出如何通过不同的手段来估算性能开销，最终帮助读者选择优化方法。

本文是主要是介绍如何最大化程序性能的，但由于tiled-based GPU是移动设备的主流，我们也会提到一些电量消耗的控制。大多数桌面程序的目标是简单的想尽一切办法在一秒内渲染更多的的帧，始终100%的消耗可以使用的电能（注：俗称跑满）。而在移动平台上，谨慎的将帧率限制在一个合适的水平，可以节省电能会有效的延长电池寿命，同时会相对的提高用户体验。当然，这并不意味着我们可以在达到了目标帧率后就可以放弃优化：越多的优化会给系统更多的休眠时间，这会更加节省电能。

最后，这篇文章的内容主要聚焦在OPENGL-ES上，因为这是tiled-based GPU的主要市场，我们会偶尔提到桌面OPENGL特性以及他们是如何运转的。

背景

桌面GPU的主要目标是最高性能。而移动GPU则不同，它必须要平衡性能和电能的消耗。设备中电量开销最大的消耗者之一就是内存带宽，相对来说，计算比缓存更加经济高效，但是计算也会产生一些临时数据，越多的数据被抛弃掉，就会消耗越多的电能。

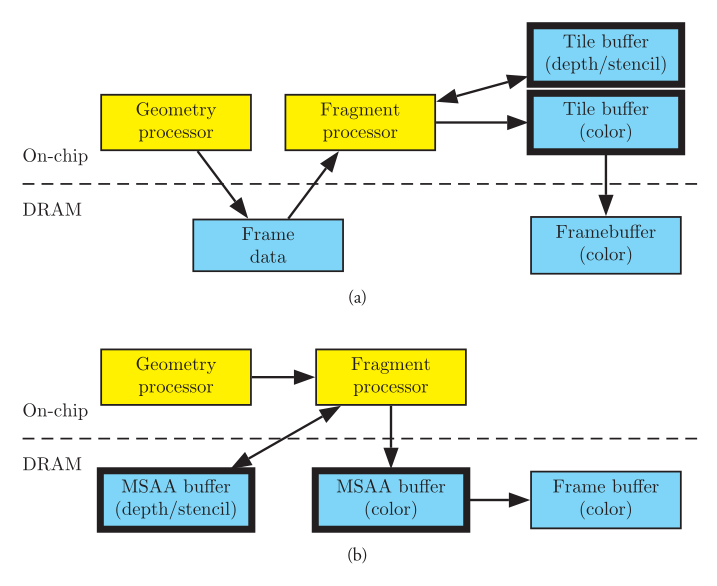
OPENGL的虚拟管线需要大量的显存带宽来支持。我们来举个例子，一个通常的用例：每个像素需要从 深度/模板缓存中读出深度值进行比较，然后将新值写回深度/模板缓存，同时将颜色值写上颜色缓存，按照常规的情况，这里算作传输了12byte(color = 4byte, d/s = 4byte, d/s读写，color只写，一共4+4+4=12byte)。不过，这是在假设没有重复绘制，没有颜色混合，没有多通道算法，没有多采样抗锯齿的情况下。如果加上这些花哨的操作，一个像素传输100bytes是一件轻轻松松的事情。因为最多有4bytes的数据需要用来显示一个像素，这是一个对带宽和电能的过度消耗。实际上，桌面GPU通常都会采用压缩技术来减少带宽消耗，但他仍旧是一个显著的消耗。

为了减少这个凶残的带宽需求，大多数移动GPU都使用了tiled-based渲染。在最基础的层面，这些GPU将帧缓存（framebuffer），包括深度缓存，多采样缓存等等，从主内存移到了一块超高速的on-chip存储器上（注：类似cache，十分昂贵高速的存储器，为不导致歧义，on-chip就不再翻译了）。这样，由于存储器on-chip了，他就和计算发生的芯片无限接近，这样，计算芯片就能以远低于常规消耗的电能来读写存储器了。如果我们可以放置一块超大的高速on-chip芯片，那么这篇文章就可以到此为止了... 但是不幸的是，那样会需要太多的硅片了。因此，最终on-chip存储器，或称tile缓存，在有些GPU中小到只能容纳16x16个像素。

这样就带来了一些新的挑战：如何在如此小的一块tile缓存中渲染出高分辨率的图像？解决方案就是将OPENGL的帧缓存切割成16x16的小块（这就是tile-based渲染的命名由来），然后一次就渲染一块。对于每一块tile: 将有用的几何体提交进去，当渲染完成时，将tile的数据拷贝回主内存，如同图表23-1所示。这样，带宽的消耗就只来自于写回主内存了，那是一个较小的消耗：没有d/s，没有重绘的像素，没有多采样缓存。同时，消耗极高的深度/模板测试和颜色混合完全的在计算芯片上就完成了。

现在，我们回到OPENGL API。这个没有根据tile-based架构来设计的渲染API。OPENGL API是典型的立即模式：他描述在当前状态下需要绘制的三角形，而不是提供一个装载了所有三角形和其状态的场景结构。因此，在tile-based架构上实现opengl，我们需要在一帧内收集所有提交过的三角形，并在之后再一并使用它们。相对的，在早期的固定管线gpu上，这一切工作都是通过软件方式完成的。现在大量的可编程移动平台gpu设计了专门的硬件单元来处理这件事(注：比如powervr的tiler等)。对于每个三角形，我们使用gl\_position的输出语义来决定哪个tile可能会被这个三角形影响，从而将这个三角形保存进这个tile的区域数据结构。同时，每一个三角形还需要将当时的渲染状态打包，例如：ps的shader，常量寄存器，深度判断方式，等等。当一个tile渲染结束后，区域数据结构就会被用于查找和这个tile相关的三角形以及像素渲染状态。

这样看来，我们貌似将一个带宽的问题挪到了另一个地方：不同于顶点数据立即的被光栅化然后被像素着色，在tile-based架构中，三角形会被保存下来以备之后使用。这样的话，就需要有足够的内存来保存原始顶点数据，顶点着色器的输出结果，三角形索引，像素状态，以及其他的一些在区域数据结构中的内容。我们可以把这些收集的数据称为frame data（ARM文档把这些数据叫做多边形列表polygon lists，而Imagination Technologies文档把他们称为参数缓存parameter buffer）。tiled-based GPU是成功的，因为这些额外的数据读写对带宽需求一般会比我们通过on-chip操作节省下来的带宽开销要小得多。只要提交的三角形保持在一个合适的水平，这个说法就一直是成立的。而过度的细分模型的几何表面，会使得frame data过度膨胀，从而导致tile-based GPU的优势不再，反而由于frame data的高带宽消耗，反而比立即模式更慢。



上面的图表表示了tile-based GPU中数据的流向。最大的带宽消耗在像素处理器和tile buffer之间，他们都位于on-chip存储器上。相对的，下面的图表是立即模式GPU的数据流向，多采样，颜色，深度，模板缓存都位于主存储器上。数据的流向需要经过存储总线。

清空和抛弃FRAMEBUFFER

当我们在做性能调校时，关于tile-based GPU最需要铭记的一件事就是：我们正在渲染的这一帧并不是framebuffer而是frame data。他是生成framebuffer的一系列数据：转换过的顶点，多边形，状态切换。不同于framebuffer，这些数据是随着DRAW CALL的增长而增长的。所以保证每一帧正确的终止是非常重要的，否则frame data会变得无穷的大。

当交换双缓冲窗口时，窗口系统会负责交换BACK BUFFER。在EGL和GLX这两种窗口系统中都有允许使BACK BUFFER失效的实现。因此，驱动程序可以在每次交换的时候，将frame data丢弃掉。然后从一张空白的画布开始。（EGL实现中，程序可以设置让后备缓冲保留，在下一节中会详细介绍）。

在使用framebuffer objects的时候，情况就变得更加复杂了，framebuffer objects不存在一个“交换”的操作。特别的，可以考虑使用glClear操作。典型的桌面GPU是立即模式架构的（immediate-mode），意味着他们在三角形准备好的时候就开始绘制像素。在一个立即模式的GPU上，一个glClear的调用就直接将clear值写入framebuffer了，因此这个代价是较高的。程序员们会使用各种技巧来避免这个操作，比如：如果他们知道接下来将会完全覆盖的绘制，他们就不clear颜色buffer。交替的利用半深度空间来避免clear深度缓存。然而，这些技巧只是在以前有用，他们已经被硬件级别的优化给超过，甚至，可能由于和硬件优化冲突而降低效率！(注：这儿的说法实在不能完全苟同...很多避免clear的操作还是很有用的)

在tile-based架构中，防止clear可能对于性能来说是个灾难（注：没错，绝对是一个大灾难，导致你的帧率下降为1/4都有可能），因为每一帧都是构建在frame data中的，清空buffer的时候会简单的释放掉frame data中的已有的数据。换句话说，glClear的性能代价不仅非常的低，而且他还能通过抛弃冗余的frame data而提高效率。

为了得到这个特性带来的全部好处，清空所有应该清空的framebuffer是很有必要的。使用scissor，color write mask或者只clear一部分颜色/深度/模板缓存会阻碍frame data的清空（这里要特别注意，译者吃过这亏，在tilebased gpu上，scissor, stencil这些常见的性能优化手法在这里都是性能灾难…）。最安全和最易移植的方法如代码列表所示。

1 glDisable(GL\_SCISSOR\_TEST);

2 glColorMask(GL\_TRUE, GL\_TRUE, GL\_TRUE, GL\_TRUE);

3 glDepthMask(GL\_TRUE);

4 glStencilMask(0xFFFFFFFF);

5 glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT | GL\_STENCIL\_BUFFER\_BIT);

这些操作需要在每一帧开始前完成，除非窗口系统已经代为操作。当然，mask和scissor如果已经设置正确后，就不必每一次都显示的设置了。（一般不会广泛的使用这两个功能）（注：在PVR硬件上，如果不将COLORMASK归位，CLEAR直接就不会成功...）

上面的讨论都只局限于这个API上：glClear。glClear是一个底层命令，而不是一个高层的hint。如果你的程序需要同时运行在tile-based架构和immediate-mode架构上的话，就会比较尴尬。所以，OpenGL ES开发者需要可移植的性能的话，可以考虑EXT\_discard\_framebuffer这个扩展，他提供了这个hint。这个扩展的调用是glDiscardFramebufferEXT，他告诉驱动程序：当前bind的这个framebuffer已经没用了，你可以随时把他用来做任何事。tiled-based架构可以通过这个hint来更加显式的释放frame data，同时，immediate-mode架构可以忽略掉这个hint。代码列表23.2展示了应该如何使用这个Hint。

const GLenum attachments[3] = { COLOR\_EXT , DEPTH\_EXT , STENCIL\_EXT };

glDiscardFramebufferEXT(GL\_FRAMEBUFFER , 3, attachments);

Discards操作在framebuffer object，或者说render-to-texture上有另外的作用。当渲染一个3D几何体到一张纹理时，例如生成环境贴图，我们需要使用深度缓存。但是，一旦渲染完后，我们就不需要使用了。这时，我们就可以通过调用glDiscardFramebufferEXT来告诉驱动程序可以释放frame data了。但是这个时候，我们还不用unbind这个framebuffer object，同时tile-based GPU还可能根据这个hint，来决定不把depth从tile buffer拷贝回主内存。尽管还在桌面OPEN GL上还不可用，EXT\_discard\_framebuffer还对多采样抗锯齿的framebuffer起作用，多采样buffer可以在他解算回单采样buffer时就被抛弃掉，节约带宽。在写作这篇文章的这时，EXT\_discard\_framebuffer相对来说还比较新，还需要一些实验来确定这些Hints在各种特别的实现下到底有多高效。

增量式的帧更新

对于一个移动相机的3D场景，比如第一人称射击游戏，我们有理由相信每一个像素在每一帧都会改变，所以，每一帧清空framebuffer不会摧毁掉任何有用的信息。但对于更多GUI类的程序来说，有很多类似控件或者消息窗口等不会改变的东西，他们没有必要每帧都重新生成。开发者在tile-based GPU上使用EGL时会经常惊奇的发现，backbuffer不会被保留到下一帧。EGL 1.4允许显式的通过EGL\_SWAP\_BEHAVIORL来申请保留，但是这在tile-basedGPU上不是默认设置，因为他会降低效率。

要理解为什么back-buffer保留机制会降低效率，让我们重新考虑一个tile-based GPU如何在一个tile内渲染像素。如果framebuffer在一帧开始时被clear掉，tile buffer只需要初始化所有像素为clear color即可。但是如果framebuffer需要保留，tile buffer就需要从原来的buffer中取出颜色，并安放到tile上正确的位置，这是需要大量带宽的。带宽的消耗可以看做是将上一帧图像作为纹理绘制到这一帧。是否使用帧保留，要根据场景的复杂度来决定，如果重新绘制一次这个帧都会比保留帧来得快，那么就选择每一帧都重绘，否则，选择帧保留。

高通提供了一个设备扩展（QCOM\_tiled\_rendering）来应对这种用例。程序可以显示的指明哪个区域是准备更新的，然后所有不在这个区域的渲染都会被屏蔽掉。GPU只需要处理与这个区域有交集的tile，剩下的区域不会被触碰到，因此framebuffer会被保留下来。这个扩展同样包含了类似EXT\_discard\_framebuffer的特性，来允许用户显示的指出当前的内容是否需要保留。举个例子，我们考虑一个程序，一个3D渲染的区域，在x,y处，长宽为w x h，将要被完全的替换，剩下的区域都不需要改变。那么我们就可以使用如下的代码，来加速这个过程：

glStartTilingQCOM(x, y, width , height, GL\_NONE);

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT | GL\_STENCIL\_BUFFER\_BIT);

glViewport(x, y, width , height);

// Draw the scene

glEndTilingQCOM(GL\_COLOR\_BUFFER\_BIT0\_QCOM);

eglSwapBuffers(dpy, surface);

注意，我们必须要为EGL的EGL\_SWAP\_BEHAVIOR设置为EGL\_BUFFER\_PRESERVED。

FLUSHING

Tiled-based GPU有时又被称作“延迟”的，这是因为显卡驱动程序会尽可能的屏蔽掉不必要的像素渲染。下面的这些操作，会导致framebuffer的内容被强制更新

* eglSwapBuffers 以及和它类似的窗口系统操作
* glFlush和glFinish
* glReadPixels, glCopyTexImage, 和glBlitFramebuffer
* 在当前帧使用遮挡查询
* 使用render to texture的结果进行渲染
* 切换framebuffer的绑定，例如glFramebufferRenderbuffer或者glRenderbufferStorage,又或者切换的RTT，都会导致framebuffer的flush。因为framedata只对原来的attachment有效。

在tiled-based gpu上，下面这个渲染方式的效率会十分低下：

1. 绘制一些三角形
2. 使用framebuffer的结果
3. 绘制另外一个三角形
4. 使用framebuffer的结果
5. 再绘制一个三角形...

当每次需要使用framebuffer结果的时候，都会导致另一次全新的像素着色流程：最坏的情况是每一个framebuffer中的像素都执行一次读取、写入的操作，即使你只需要绘制仅仅一个三角形。因为每一次像素着色流程的消耗都非常高，因此我们的目标是尽量每帧只存在一个像素着色流程。

即使是从GPU中取得framebuffer的内容，这个消耗也是存在的，比如说读取render to texture的纹理颜色或者调用glReadPixels去读取pixel pack buffer，因为每一次的“绘制-读取”操作都会需要像素着色流程再次运行。和桌面平台的立即模式GPU对比，glReadPixels在执行时的消耗基本上可以不用关心。

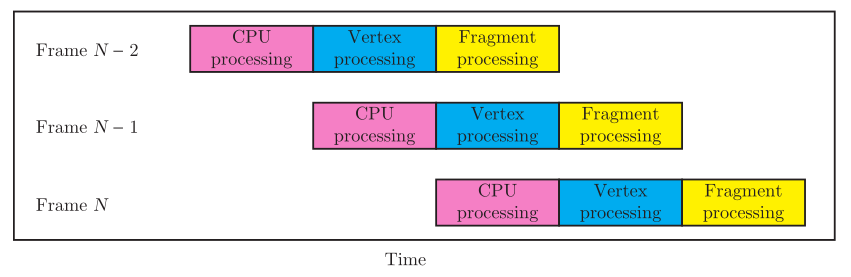
在某些driver下，glBindFramebuffer也会导致为绑定前的那个framebuffer开启一次像素着色流程。因此，一帧内，每个framebuffer最好只绑定一次。举个例子，考虑一个场景，里面有一些光滑的物体，使用了实时生成的环境图。常规的做法是：在每个物体渲染前，将这个物体需要的场景图渲染出来，然后使用。但是，这里更好的做法是，在绑定最终绘制的framebuffer前，就将所有物体需要的环境图全部生成好，这样能够减少framebuffer的重复绑定，从而提高效率。

除了上述的这些情景，这里还有一个情况会导致framebuffer的强制刷新。由于内存是有限的，所以用于framedata的内存大小会随着这一帧提交的几何体大小而变化，当你一直渲染越来越多的几何体而不进行framebuffer交换或者清空的话，程序最终会导致内存溢出。为了防止这个情况发生，driver最终会进行强制更新，来确保内存不会溢出。这个操作代价是十分大的，不同于交换操作，所有的缓存，包括MSAA缓存，会被写到其他地方，之后再重新加载回来以继续渲染，这会导致一次16倍于常规强制刷新的贷款消耗。

这个情况，就意味着，场景的性能表现和渲染的三角面数量并不是线性相关的。因此，我们不能简单的用小场景来作性能测试，在估计当前程序能够承载的目标三角面数量时，有必要针对这些应用情景作一些检查。

渲染迟滞

由于一帧内的顶点和像素的处理是发生在相对独立的阶段的，应用程序会将CPU处理, 顶点处理，像素处理安排在相邻的三帧中。如下图所示。当一个渲染命令提交后，要在当帧之后的第三帧，渲染结果才会显示出来。



延迟除了会影响用户的操作感受外，还会影响从GPU中往CPU回读信息的操作。同步的查询操作，例如glReadPixels，将会导致CPU挂起强制等待后两帧的返回。即使是异步查询，例如遮挡查询，查询结果最终会被读回，过于频繁的查询调用也会导致CPU挂起等待。

如果可以等待结果返回时再使用，那么 GL\_QUERY\_RESULT\_AVAILABLE这个check就有用了。为立即模式编写的代码一般会断定，查询的结果在一个固定的时间段内一定会得到返回，或者可以等待带更长时间，甚至poll到它返回为止。同样的，如果必须使用glReadPixels的话，从那些已经完成的framebuffer上取得像素颜色而不是当前绘制的framebuffer会极大的提高效率，降低渲染迟滞，同时得到一个相对可接受的查询反馈。

迟滞还体现在另一个重要的地方：当资源在使用时修改资源。一个普遍的例子就是动画mesh，这是一个每帧都需要更新顶点数据的用力。之前的顶点数据可能还在被上一帧的顶点计算单元使用，而这是如果应用程序要更新顶点缓存，那么这块内存必须等待上一帧的使用操作完成后才能被写入。在大多数情况下，drivers通过创建一块额外的内存拷贝来防止等待，（将新值写入拷贝内存，待之前的使用完成后再更新）。但是，在一个内存，贷款都十分吃紧的移动设备上，这个copy-on-write的发生是值得我们关注的。特别的，当频繁更新的资源在一帧多次使用时，这个问题会更加严重，会导致多次的copy-on-write。所以，如果可能，所有的资源更新尽量在资源使用前完成，我们显式的进行copy-on-write的管理，而不是将它扔给driver。

在使用诸如EGL\_KHR-image\_pixmap或GLX\_EXT\_texture\_from\_pixmap这类扩展时需要非常小心，他们会修改操作系统的pixmaps。驱动程序通常不会有更多的自由在内存中移动这些资源。有可能会使得整个系统挂起，或是强制提交所有结果到framebuffer，然后重新加载。

tiled-based GPU通常比立即模式GPU有更高的迟滞，因此，为立即模式GPU优化的代码可能在tiled-based GPU上需要重新优化。

隐藏表面剔除

立即模式GPU处理重叠物体是用新像素覆盖已绘制像素，这里会有两个多余的消耗：一个是被覆盖像素的着色消耗，一个是被覆盖像素的冗余带宽消耗。在tiled-based gpu上，后一个消耗是不存在的，因为屏幕像素将在完全处理结束之后再写会主内存，但是着色的消耗依然存在。所以，进行高层裁剪，从前往后的组织不透明物体的渲染仍然十分必要，这可以通过硬件的“预深度检测”(early depth test)来提高效率。因为上述这这些问题，在tiled-based gpu上，在CPU排序和GPU着色消耗之间的平衡方式的选择上可能和立即模式GPU有所不同。

PowerVR的GPU家族，拥有像素着色阶段的逐像素表面剔除特性[Ima 11]。在运行任何像素shader之前，多边形会被预处理来决定哪些像素可能对最终的结果有贡献，之后，只执行这些有贡献的像素，剔除掉其他像素。这个剔除方式需要对几何体进行排序，要完成这个优化，像素shader的结果必须要确保能够完全的覆盖他们遮挡的像素。而如下注入带有discard指令的shader，或者使用了Mask，alpha-test, alpha-to-coverage, 颜色混合特性将会屏蔽掉优化，因为他们“遮挡”的像素有可能对最终的图像产生贡献。因此，逐像素表面剔除特性只会对需要他们的物体开启。

当PowerVR系硬件的“隐藏面剔除”功能失效时，还有另外一个选择，使用一个（深度流程）depth-only pass：关闭颜色写入，赋予空的像素shader来生成深度缓存，接下来再使用正确的像素shader来正式的绘制场景。depth-only pass会决定每个可见像素的深度，接下来在真正绘制的时候，只有这些可见像素会被处理（通过预深度检测）。

depth-only-pass技术在立即模式GPU和tiled-basedGPU上对于复杂的着色计算场景来说都是十分有效的优化手段，但trade-off却是不同的。在两种平台上，depth-only-pass都会增加顶点处理和光栅化的消耗。在立即模式GPU上，depth-only-pass会增加额外的带宽消耗，因为深度缓存的访问次数会翻倍。在tiled-based GPU上，depth-buffer的访问很快，不会增加主内存的带宽消耗，但是由于所有提交到frame data的几何体信息都复制了一次，因此这里会有一个较小一些的带宽消耗。因此，对于带宽瓶颈的程序来说，depth-only-pass在立即模式GPU上没有优化效果时，换到tiled-based GPU上也许会有优化效果。

颜色混合

在立即模式GPU上，颜色混合通常是一个代价很高的操作，因为完成混合需要一个在framebuffer上的读取-写入循环，这个操作发生在相对较慢的主内存中。而在tiled-based GPU上，读取-写入循环完全发生在芯片的快速存储器中，所以这个操作的代价非常小。一些GPU还专门实现了处理颜色混合的硬件，来使颜色混合变得几乎免费，其他的GPU一般都使用shader指令来实现颜色混合。因此，颜色混合会降低着色运算的最大指令数。

需要注意的是，我们只是指出了颜色混合的直接消耗，让一个物体部分透明还带有间接消耗，因为这个物体不能被当做遮挡体了。被物体遮住的像素需要被处理，而如果不用透明，他们本该可以被隐藏面剔除或者预深度检测时被剔除。

多采样

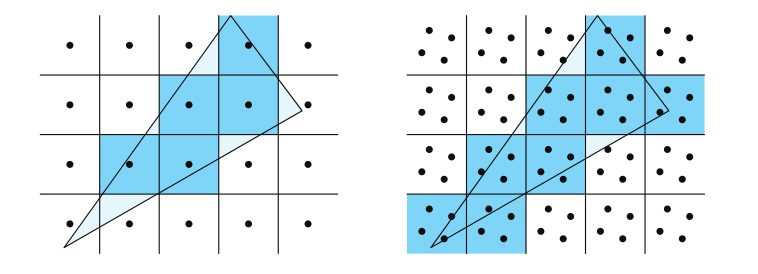
多采样是一个相对高效的提升画面质量的技术，而又不用牺牲像超采样那样多的代价。每一个在framebuffer上的像素都存储多个采样结果，这些采样结果将在最后生成抗锯齿图像。然而，被光栅化后的图源，每个像素只用被着色一次。但这已经让像素着色消耗变得很大了，在immediate立即模式GPU上，多采样会带来很大的带宽消耗：4次多采样（通常的采样数选择），使得所有对framebuffer的读写操作的带宽消耗提高四倍。多种硬件在选择多采样位置的方法上进行优化，但多采样仍然是代价很高的操作。

相反的，在tile-based GPU上，多采样的代价是很小的，因为多采样的采样点只需要保留在on-chip缓存中，在所有处理完成后才写回framebuffer的主内存。因此，多采样不会带来多余的带宽消耗。

当然，这里依然会有两个消耗代价：

首先，4次多采样需要4倍的tile缓存。由于tile缓存容量相当的宝贵，一些GPU会在开启多采样时，缩小tile的尺寸，以容纳采样点需要的缓存。缩小的tile会带来一些额外的性能开销（每个tile都需要存储更多的图元），但是减半tile的尺寸并不一定会减半性能，所以当程序瓶颈在像素着色时，只会看到一个很小的性能下降。

其次，多采样的另一个开销（在immediate立即模式GPU上也会存在）则是在物体边缘会生成更多的像素。如下图所示，每个多边形会检测到更多的像素。这还不止，这些多采样的区域，前景和背景几何体会同时向同一个像素贡献颜色以供混合，这些图元都需要被着色，因此硬件隐藏面剔除机制不能剔除掉这些图元。这些额外的图元消耗会根据场景的边缘多少而不同，但10%是一个比较好的初始猜测值。



性能分析方法

在立即模式GPU上，ARB\_timer\_query扩展可以用来获取某一段场景渲染的消耗。以个范围内需要分析的渲染命令被glBeginQuery和glEndQuery包裹起来，然后这些渲染命令消耗的时间会被测量，并返回。

尽管这个扩展有可能被实现在tiled\_based GPU上，但这个结果可能在帧粒度一下都不会有任何作用。这是因为，在tiled-based GPU上，命令不一定会按照他们提交的顺序来执行：所有的顶点处理会在第一遍完成，然后像素处理会按照tile的顺序依次执行。因此，性能分析需要依靠更多的干预技术，例如切换场景的启用/禁用开关，来观察性能的下降。硬件提供商提供的读取内部性能计数器的特供工具也能为确定渲染流水线的性能瓶颈提供很大的帮助。

不同于后期优化，在开始开发时就引入一个性能衡量基准，来帮系统决定三角形，纹理，着色器复杂度的预算是一个通常的好办法。在这样做的时候，一定要记住，当提交了过多的三角形而没有交换，会导致一个性能消耗的陡然提升，这在第五节中已经提过。同时，确保哪些渲染命令真正被执行了也是十分重要的，在渲染调用后放置一个glClear在他们到达GPU前取消这些渲染调用（因为他们根本不起作用）。

总结

每一个GPU，每一个设备驱动都是不同的。而在优化和尝试中的不同选择意味着唯一能够真正决定性能瓶颈的选择就是去实际的测试它。然而，下面的这些没有科学根据的规则，可能是在tile-based GPU上发现高效率渲染的一个起点。

* 在一帧的开始清空或抛弃整个渲染内容：包括颜色，深度，模板缓存。
* 对于每一个framebuffer object，一帧中只绑定一次，并且，在解绑或使用framebuffer object的结果之前，要确保所有影响它的渲染指令提交完毕。
* 在使用遮挡查询或其他取得渲染结果的机制时，记得渲染迟滞效应。并且，如果该程序在之前为立即模式GPU优化过，那么，在tile-based GPU上需要重新优化。
* 将三角面数量控制在一个合适的水平，尽量避免大量的精细模型。
* 在拥有内建的隐藏面提出的硬件上（POWER VR），就没有必要消耗CPU计算能力去从前往后的排序不透明物体了，在其他硬件上，这个步骤很有必要。同时，可以考虑快速深度渲染（depth only pass）。
* 利用代价较低的多采样。
* 记住，在移动平台上，性能必须和电量消耗做一个平衡。