5.5　针对较快镜头眩光的纹理屏蔽

Chris Maughan, Nvidia Corporation
cmaughan@nvidia.com

这篇文章介绍一种全新的途径从已经渲染到帧缓冲的像素中生成纹理信息。这种技术可以按几种不同的方式来使用，但这里介绍的方案针对阻塞镜头眩光的常见问题。许多游戏为了准确决定最后场景中的可见物，试图回读帧缓冲中的像素。下面将讲解一种无需 CPU 的协助，也不需要从图形卡读取数据的技术；还将归纳为什么从图形卡回读信息是一种高开销运算，因而应该尽可能避免的原因。

5.5.1　镜头眩光遮挡

现代游戏为场景添加镜头眩光增加真实感。镜头眩光通常是场景中应用的最后一项，它使用一个渲染为帧的布告版（Billboard）的 2D 纹理贴图。进一步完善此技术的办法是，场景中的物体可以阻塞太阳图像。在这种情况下，正确的视觉效果是镜头眩光强度减弱。可以形象地想像成在一条大树成排、阳光灿烂的大道上驱车。如果太阳在树平线之下，随着通过树的视点的改变、到达眼睛的太阳光数量的变化，您就会感觉到有闪烁出现。

检测太阳阻塞的通常途径是首先渲染场景中的物体，包括太阳本身。然后，回读帧缓冲数据，此处正是太阳像素，而且可以推导出场景中的可见太阳光量。完成的方式有两种：可以回读颜色缓冲，并查找匹配太阳颜色的源；或者回读 Z 缓冲，查找同太阳远距离的 Z 值。可见与阻塞太阳像素的比率是镜头眩光强度的近似。现在使用一个 alpha 值设置它的强度，将它混合到最后场景中，就可以绘制出镜头眩光了。

5.5.2　硬件问题

当在游戏中渲染镜头眩光时通常采用前述方式。虽然它能很好地工作，但在图形管线中导致了不必要的停顿，而这可能会严重降低性能。

现代图形管线非常深。多边形数据不仅在图形芯片管线中排队，而且在大的“分段式”缓冲中排队。一般情况下，游戏中有数千个多边形在分段式缓冲中排队，而且图形芯片在绘制多边形到帧缓冲中时消耗该缓冲。在优秀的并行系统中，游戏可以在 CPU 上进行有用的工作，如物理计算、AI 等，同时图形芯片（graphic chip，GPU）排泄分段式缓冲。实际上，如

果要获得系统的最佳性能，推荐大家这样做。图 5.5.1 显示了具有优先级并行性的系统。

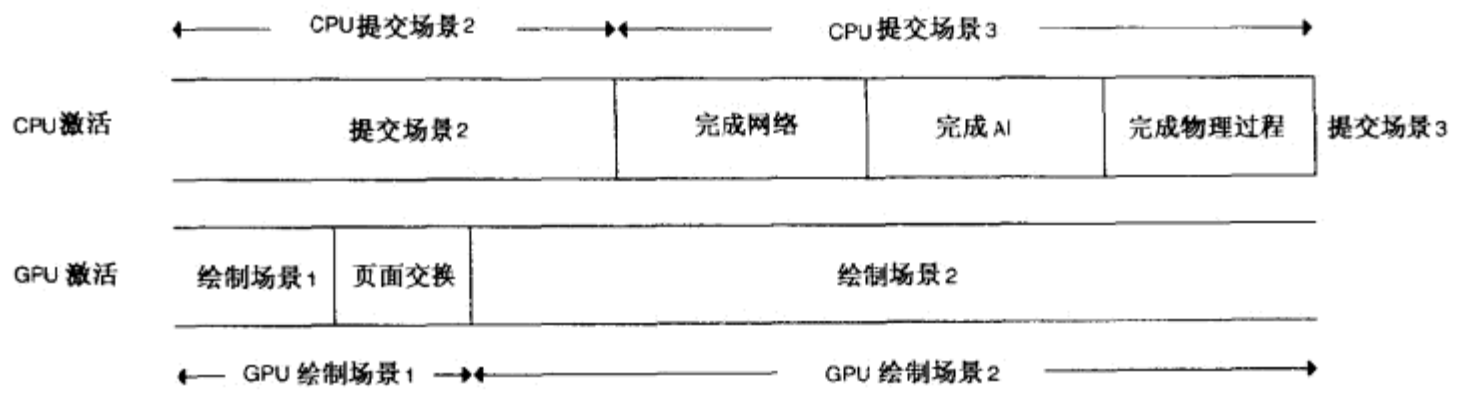

图 5.5.1 理想游戏引擎中 GPU/CPU 的并行性

现在考虑游戏将当前场景中的所有多边形提交给图形芯片后的情形。许多场景仍然在分段式缓冲中排队等候，在一段时间内不会绘制。游戏接下来要做的是回读场景中的内容，决定镜头眩光计算中的太阳可见度。这里就出现第一个问题。为了从帧缓冲读取完整场景，必须等待渲染的完成，而且为了发生此操作，必须刷新整个管线。等待操作发生的同时，CPU 未有效使用，因为它只是在图形卡驱动程序内空闲，等待渲染的完成。理想情况下，我们希望 GPU 和 CPU 时刻同时保持激活。解决这个问题的一种可能方案是，在场景渲染之后，但在计算镜头眩光这前，插入物理/AI 代码。按这种方式，GPU 渲染多边形的同时 CPU 也将保持繁忙。

在这种情况下，如果 CPU 的工作负荷比 GPU 大，就会产生一个问题。因为 GPU 要等待 CPU 完成工作，可以认为系统是非平衡的，它此时可以进行更多的渲染，如图 5.5.2 所示。

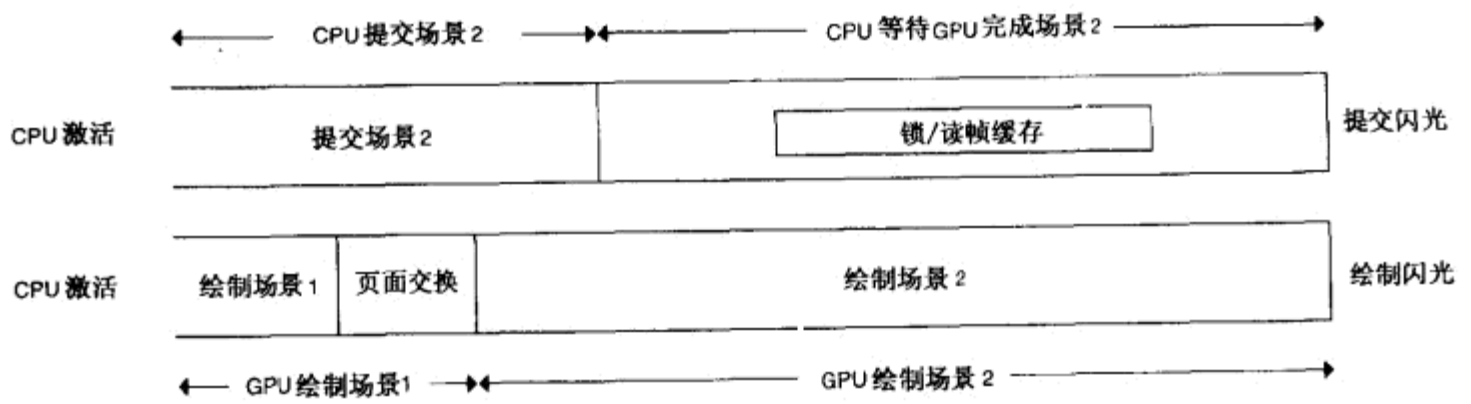

图 5.5.2 渲染结束时的刷新导致的管线延迟

如果 CPU 的工作负荷小于 GPU 的工作负荷，当回读镜头眩光数据时同样会延迟。这不是我们所期望的；我们期望的理想情况是，CPU 在准备游戏下一帧的同时 GPU 完成最后一项工作（如图 5.5.3 所示）。并行是这里的关键所在，插入任何延迟都会造成相当大的损害。

不过，事情并未到此结束。失去并发后，还会出现另外一个问题。从当前生成图形卡回读数据是一个缓慢的运算，在某些情况下根本不能执行。虽然大部分图形卡写入速度为 AGP 2x/4x，但回读速度仍然限制在 PCI 水平上，这是现有 AGP 总线设计的固有本质。结果是，对于 32 bit 帧缓冲，在没有缓冲和突发的情况下，最高只能以 66MHz 的速度读取像素。因此，从图形卡读取 256×256 的区域要花费 1 / (66 000 000 / (256×256)) = ~1 ms。

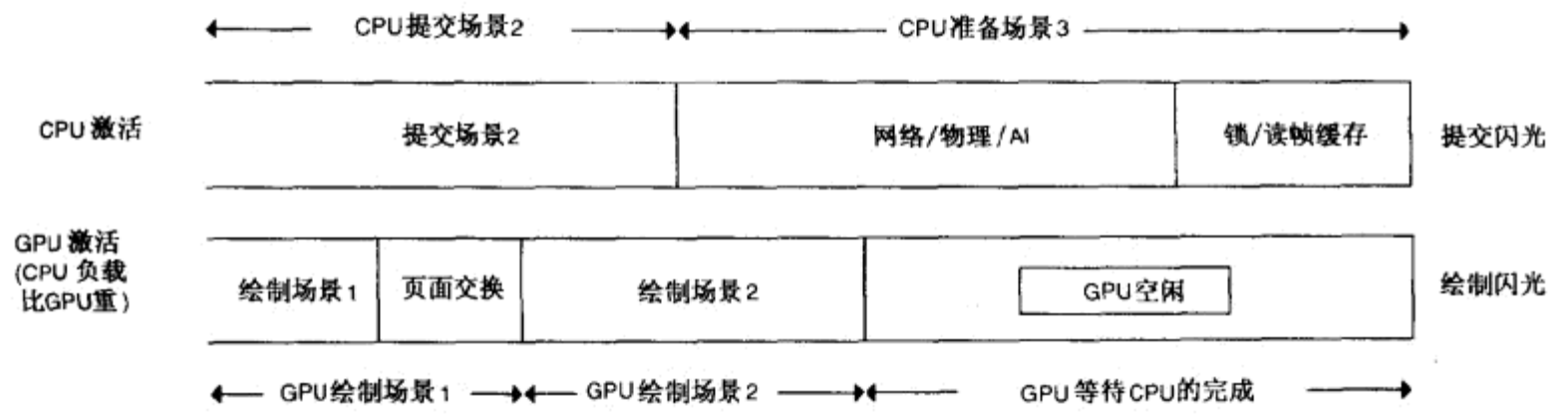

图 5.5.3　CPU 结束工作后，帧结束前的刷新导致的管线延迟

假定 256×256 像素太阳区域投影到屏幕上。如果我们离光源（例如街灯）较近，则投影面积可能变化相当大，直到回读变得非常突出为止。即使我们愿意接受这种性能上的降低，也不能保证图形卡让我们回读数据。事实上，很多图形卡并不允许这样做。大部分图形卡不允许读取 Z 缓冲，有一些图形卡不允许读取帧缓冲，没有图形卡允许读取反锯齿缓冲（antialiased buffer）。

5.5.3　纹理屏蔽

如何才能减轻读取帧缓冲带来的延迟呢？我们通过横向思维来寻找答案。基于前述原因，我们知道，不能从 CPU 向下传递值来调整镜头眩光亮度。现在只剩下一种可能性：通过卡中图形内存里已经生成的数据来调整镜头眩光值。这个值必须以纹理贴图中质素的形式出现，而且这个贴图像素需要包含一个照明颜色值，或者包含一个 alpha 值。我们可以在 Direct3D 或 OpenGL 中渲染到纹理。剩下的问题是如何基于一套源像素，实际生成贴图像素。这要通过 GPU 的 alpha 混合功能来实现。alpha 的混合单元基本上是一个加法器，它把 GPU 中的一个值添加到帧缓冲中的某个值上，而且可用于累积颜色信息。再加上场景中的物体用我们选择的颜色进行渲染，就得到合适的解决方案了。下面是达到所需结果的步骤：

第 1 步

创建一个 16×16 贴图像素的纹理贴图，称为“太阳帖图”。我们将在其上渲染太阳，并遮挡太阳的几何体。表面格式应该至少为每颜色分量 8 bit。在一般图形硬件上，这等于 32 bit 的 ARGB 表面。选择太阳贴图为 16×16，因为它包含 256 个贴图像素。我们以后会看到，并不一定要限制到 16×16 大小，如果必要，可以通过滤波技巧来减小大小。

第 2 步

创建 1×1 纹理贴图，称为“亮度贴图”。这是强度数据的最终目标。这只需要单个贴图像素的信息，但当然，这个贴图像素可以按需要变大，以满足特殊的硬件需要。同样，表面的格式应该至少为每颜色分量 8 bit。

第 3 步

将包含太阳的场景渲染到太阳贴图。方法有多种，但我们选择一种简单的途径。我们导引观察者直视太阳的中心，同时设置投影矩阵，让太阳填充视口。前端平面正好在眼睛前面，后端平面在太阳之后，用黑色清除太阳贴图。我们在太阳贴图上渲染两种颜色。太阳图像本身用白色渲染。遮挡几何体用黑色渲染，于是覆盖了所有已经渲染的太阳值。如果太阳首先渲染，则不需要 Z 缓冲，因为我们只对阻塞感兴趣，不对深度级感兴趣。太阳贴图现在包含 256 个值，一些值为白色，一些为黑色。在 32 bit 颜色中，这意味着帧缓冲包含 0xFFFFFFFF 或 0x00000000。请注意，为了保持一致，我用设置颜色通道的相同方式来设置 alpha 通道，打开 alpha 混合或颜色混合选项。

第 4 步

渲染 256 像素到 1×1 强度贴图。如果 API/硬件支持，使用点列表或 point-sprite 来绘制它们，因为它们需要的转换工作比完整的三角形或四边形少。设置纹理坐标，让每个太阳贴图像素被引用一次。设置 alpha 混合单元，在目标亮度贴图上添加每个源太阳贴图的颜色信息。我们还希望从太阳贴图一次只采样一个贴图像素，所以设置纹理采样模式为点采样。注意，我们将使用不同的源贴图像素，多次写入单个目标像素。下一步是混合每个太阳贴图质素与常量值 0x01010101。它实际上是每个颜色通道的 1/255。结果是，调制后的太阳贴图值要么是 0x01010101（如果源太阳贴图像素为白色），要么是 0（如果源太阳贴图像素为黑色）。我选择白色和黑色作为太阳贴图中的值，是因为在渲染中它们一般容易获得，而且可以在演示中显示，便于调试。

这里所使用的技巧是，从太阳贴图向亮度贴图添加 256 个值，在亮度贴图中形成 256 个值的可能范围。亮度贴图中的像素越亮，太阳贴图中就有越多的可见太阳像素；场景本身也是如此。我们遮掩了前面渲染的纹理结果，并将它们加合在一起。我们以后会看到，对不同的应用，纹理屏蔽方式非常有用。

整个过程使用 GPU 从帧缓冲中的信息生成镜头眩光。因为 GPU 管线是串化的，在生成镜头眩光之前，写入了亮度信息，而且管线不需要在前面讨论的运算过程中刷新。另外，因为我们分离了镜头眩光强度与场景渲染，所以可以在任何时候执行这些步骤，而且可以在渲染实际场景期间而不是渲染结束时使用结果值。也可以使用这个值来修改场景的环境强度，来模拟观察者的瞳孔对光的反应。除非希望渲染两次场景，否则不能采用这种需要锁定的方法。

5.5.4 性能考虑

以上介绍的算法看起来有些冗长，但事实上，与读取帧缓冲和刷新管线相比，运算的开销相当少。考虑到在帧尾我们添加了少量的 GPU 工作，它可以继续与引擎的余下部分平行运行。要当心的是，太阳贴图的渲染必须快速。若考虑太阳通常在地面之上，而且对太阳的视场实际上非常小，所以确保这一点。它大量减少了必须考虑的物体，能够快速排除那些通过剖面挑选进行的渲染。如果我们判定某些物体完全阻塞了太阳，则可以进行早期排除，因此

根本不必绘制它们的镜头眩光。另一种选择是，用较少的几何体细节来渲染一些阻塞体。

另一个性能上的问题是，我们使用 256 个混合多边形的渲染来查找强度结果，但这不会有问题，因为现代 GPU 每秒约可以渲染 4 亿个点，消耗的时间只是用于从帧缓冲回读大量数据的时间的一小部分。另外，从帧缓冲回读数据用到 CPU，我们要尽量减少 CPU 在非图形任务中的使用。

5.5.5　改进

前述方案能很好地发挥作用，而且得到理想的近似效果。但该技术有一个明显的缺陷：太阳通常是圆形，而我们所使用的太阳贴图是一个正方形纹理。事实上，这不算一个问题，因为我们可以从源太阳贴图按需要采样任何质素，包括圆形采样模式。那当然，为了让它覆盖所要求的采样数，必须让太阳贴图更大。

如果愿意，可以只采样太阳贴图的选择性贴图像素。为此，只要采样太阳贴图中需要的贴图像素，并在写入强度贴图时更改调整运算来放大结果。

注意，我们只是作为一个例子来说明如何解决镜头眩光问题，不过其中的许多技术在其他场合是可重用的。也许强度值可以用于变暗场景来模拟感光过渡效果，或者为人物添加轮廓或晕轮。事实上，只要我们意识到 GPU 具有作为数学解决工具的巨大灵活性，把 GPU 用作一般的平行数学引擎，就可以用多种方式来修改算法。例如，可以使用纹理滤光，一次采样 4 个贴图像素，得到 4 个样品的平均值。在前面的例子中，如果我们依靠双线性滤光得到 4 个样品的平均强度，只需要绘制 64 个点就可得到强度贴图结果。也可以改变混合运算，进行若干数学运算，如使用调制，来缩放值。如果我们想在强度贴图中累积更多的样品，它就可以发挥作用，因为我们能够以动态范围为代价，使用调制给强度贴图结果添加缩放。

5.5.6　示例代码

附带光盘上的演示例子说明了锁定技术与纹理屏蔽技术之间的性能差异，也说明了如何实现本文所描述的算法。在菜单中提供了一些选项，可以显示太阳贴图和强度贴图。

当在我的目标系统上运行这个演示时，与纹理屏蔽方法相比，使用锁定方法花费了约 50% 的帧频。帧缓冲变大，性能也得到增加。开放场景的典型数字如下：

纹理屏蔽 (600×500×32bit) = 53.5 fps

帧缓冲锁定 (600×500×32bit) = 27.8fps

源代码中有详细的注释，很容易理解和实战。我使用 Direct3D 8 来编写这个演示，但其概念与 OpenGL 相同。请参考光盘中包含的 README 文件，了解如何运行演示和分析结果的更多信息。README 文件内容还包括如何使用演示中的“Increase CPU Work”选项来研究使用锁定调用与纹理屏蔽技术对系统并行的影响。

5.5.7 替代途径

为了叙述的完整性，还应提及实现镜头眩光的两种替换途径：

- 有时，可采用一种基于几何体的途径来判断眩光的可见性。这种途径扫描几何体，进行光线交集测试，来决定太阳像素网格的可见性。虽然它能正常工作，但 CPU 的计算时间可能占用较多。而且在使用告示牌时，它不再有用，因为必须扫描源纹理查找“黑洞”。
- 一些图形卡可以异步回读帧缓冲数据。在这种情况下，光源区与其余场景的渲染并行上载。假定在启动回读的阻塞物体被绘制之后、在完成其余场景或光照工作前，场景中存在适当的点，则这一功能可以发挥作用。这个方法当然依赖于 API 的支持，来异步回读数据，而且依赖于硬件的支持来上载数据。在撰写这篇文章时，在 Direct3D 或 OpenGL 中还不提供这种支持，不过，两种 API 的扩展已经得到了提议。

5.5.8 参考文献

[King00] Yossarian King, “2D Lens Flare,” *Game Programming Gems*, Charles River Media Inc. 2000: pp. 515~518.