阴影锥原理与展望---真实的游戏效果的实现

DonewsBlog

http://blog.donews.com/yyh/archive/2005/05/19/387143.aspx

前言：真实的游戏效果

shadow volume 这个术语几乎是随着 DOOM3 的发布而成为FPS 玩家和图形学爱好者谈论的对象的。虽然这个游戏还没有上市，但是凭借 John Carmack 的传奇经历以及 DOOM3发布的一些让人惊讶的预览图片，我们仍然有理由认为它将会是 2004 年最热门的 FPS 游戏之一。 id software向来都不吝惜为了达到最好的图像效果而使用最先进的渲染技术 ,这曾经使得玩家为了玩它开发的游戏而不得不掏光口袋里面的钱来升级电脑，不知道这次我们可以幸免吗？

自 DX9 发布以来，大家的注意力似乎都被 shader 吸引住了， BBS里面谈论的话题也总是离不开 shader based rendering ，前一段时间关于 GPU内部精度的讨论大有遮天蔽日之感，但其实和闪闪发光的金属小球以及波光鳞鳞的水面比较，几个简简单单的影子常常能带给场景更多的真实感。也许这就是为什么DOOM3能够在多如牛毛的 FPS 游戏中脱颖而出的原因之一。

阴影的实现方法有很多种，现在比较流行的主要是 shadow mapping 和shadow volume. 前者实现起来相对简单，可以发挥现在 GPU 可编程流水线的能力，但是由于先天不足，shadow mapping在处理动态光源/物体的时候开销过大，经常作为一种静态场景中的廉价替代物。而 Shadow volume 的强项恰恰是 shadowmapping 的短处，像 DOOM3 这种大量运用动态光源，并且要对时刻都在运动中的物体投射阴影，shadow volume是现阶段唯一的选择。

Shadow mapping 的原理：

一个物体之所以会处在阴影当中，是由于在它和光源之间存在着遮蔽物，或者说遮蔽物离光源的距离比物体要近，这就是 shadow mapping 算法的基本原理。

Pass1: 以光源为视点，或者说在光源坐标系下面对整个场景进行渲染，目的是要得到一副所有物体相对于光源的 depth map （也就是我们所说的shadow map ） ，也就是这副图像中每个象素的值代表着场景里面离光源最近的 fragment 的深度值。由于这个 pass中我们感兴趣的只是象素的深度值，所以可以把所有的光照计算关掉，打开 z-test 和 z-write 的 render state 。

Pass2: 将视点恢复到原来的正常位置，渲染整个场景，对每个象素计算它和光源的距离，然后将这个值和 depth map中相应的值比较，以确定这个象素点是否处在阴影当中。然后根据比较的结果，对 shadowed fragment 和 lightedfragment 分别进行不同的光照计算，这样就可以得到阴影的效果了。

从上面的分析可以看出来，depth map的渲染只和光源的位置以及场景中物体的位置有关，无论视点怎么运动，只要光源和物体的相互位置关系不变，shadow map就可以被重复使用，因此对于没有动态光源的场景， shadow mapping 是很明智的一种选择。

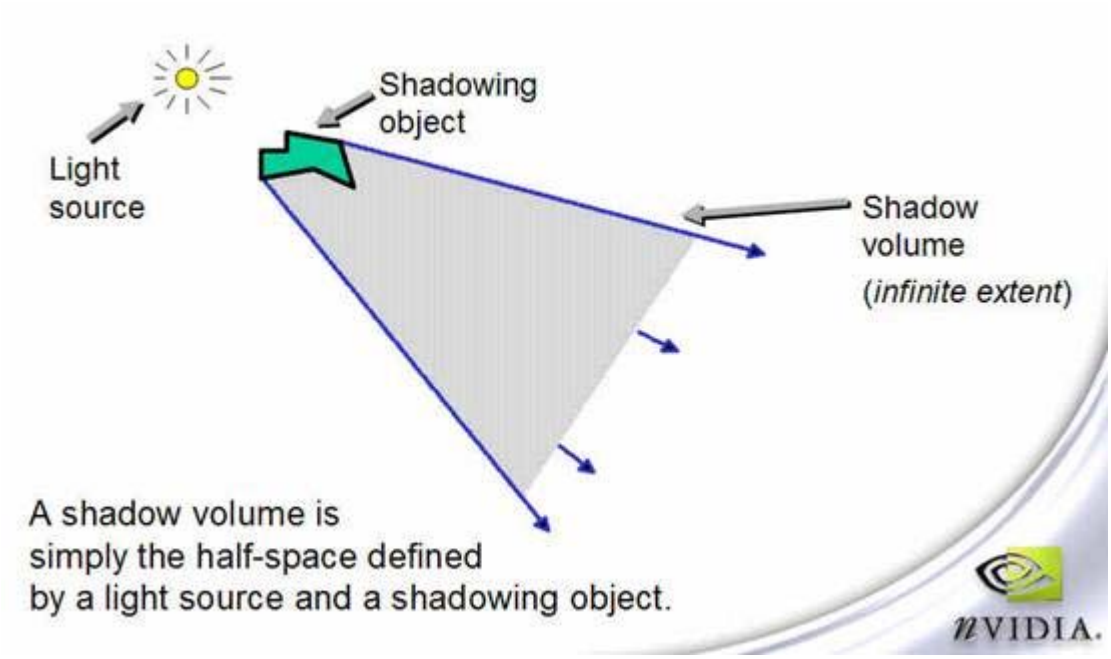
除了上面提到的不能很好应付动态光源场景的限制之外，shadow mapping 还存在着所有使用 texture的场景面临的共同问题－锯齿。根据采样定理，只有纹理分辨率小于或者等于物体的实际分辨率时才不会失真，而当一副很大的纹理被贴到尺寸比它小的物体上时，会出现一个 fragment 覆盖多个 texel 的情况，这时要准确的再现这个 fragment 的颜色信息，就要综合考虑所有被它覆盖的texel 产生的影响，这就是各种纹理滤波方法最基本的原理。但是由于 depth map 是在不断的变化当中，所以不能像一般的纹理那样把各个 mip -map 事先计算好放到显存里面。有一种利用 pixel shader 的方法对 depth map 做 bilinearfiltering, 但是开销很大，在现阶段不具备实用意义。同样的问题在纹理分辨率小于屏幕分辨率的时候仍然存在，这时多个 fragment会被投射到同一个 texel 上面，虽然从再现纹理的角度来说并不存在失真，但是由于多个 fragment共用同一个纹理值，锯齿问题还是存在。更糟糕的是，没有一种滤波技术可以从根本上解决这样的锯齿，因为从数学上讲，人们不可能通过运算来创造出比原始量更多 的信息。近年来，为了解决 shadow mapping 的锯齿问题，人们做了很多努力，比较有前景的是 adaptive shadowmap(ASM) 和 perspective shadow map(PSM) 。两者的基本原理都是在可能产生锯齿的地方人为增加采样率，使得一个fragment 至少对应一个 texel , 区别是 ASM 增加采样率的地方是在 shadow 边缘，而 PSM是在靠近视点的地方。修补一个本身存在缺陷的方法从数学上来说是缺乏美感的，正像 John Carmack 在 2002年8月的一封 email中所说：

“ Shadow buffers makegood looking demos with controlled circumstances, but when you startusing them for a “real” application, you find that you need absolutelymassive resolution to get acceptable results for omni － directionallights, and a lot of the artifacts need to be tweaked on a per-lightbasis. While it is possible to do shadow buffers on GF1/radeon classhardware, without percentage closer filtering they look wretched. If wewere targeting only the newest hardware, shadow buffers would have abetter shot, but even then, they have more drawbacks than are commonlyappreciated. ”

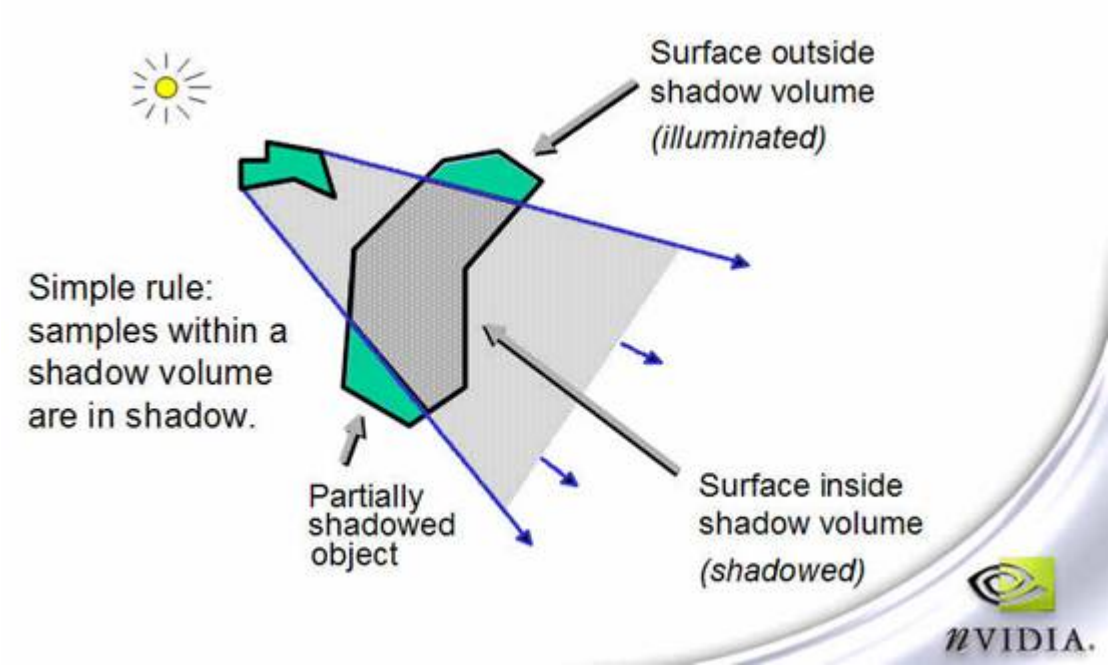
看起来似乎 John Carmack 找到了实现阴影更好的方法？让我们来看看它究竟是什么。

Shadow volume 的原理：

Shadow volume 这种算法第一次被提出是在Franklin C. Crow 在 1977 年写的一篇论文 “SHADOW ALGORITHMS FOR COMPUTERGRAPHICS ”里。其基本原理是根据光源和遮蔽物的位置关系计算出场景中会产生阴影的区域（ shadow volume），然后对所有物体进行检测，以确定其会不会受阴影的影响。



图中的绿色物体就是所谓的遮蔽物，而灰色的区域就是 shadow volume。



只有处于 shadow volume 里面的物体才会受阴影的影响。

shadow volume的算法

现在清楚了 shadow volume 的基本原理，那么如何确定一个物体或者一个物体的某一部分处于 shadow volume 中呢？这就要用到 stencil buffer 的帮助了。

z-pass 算法：

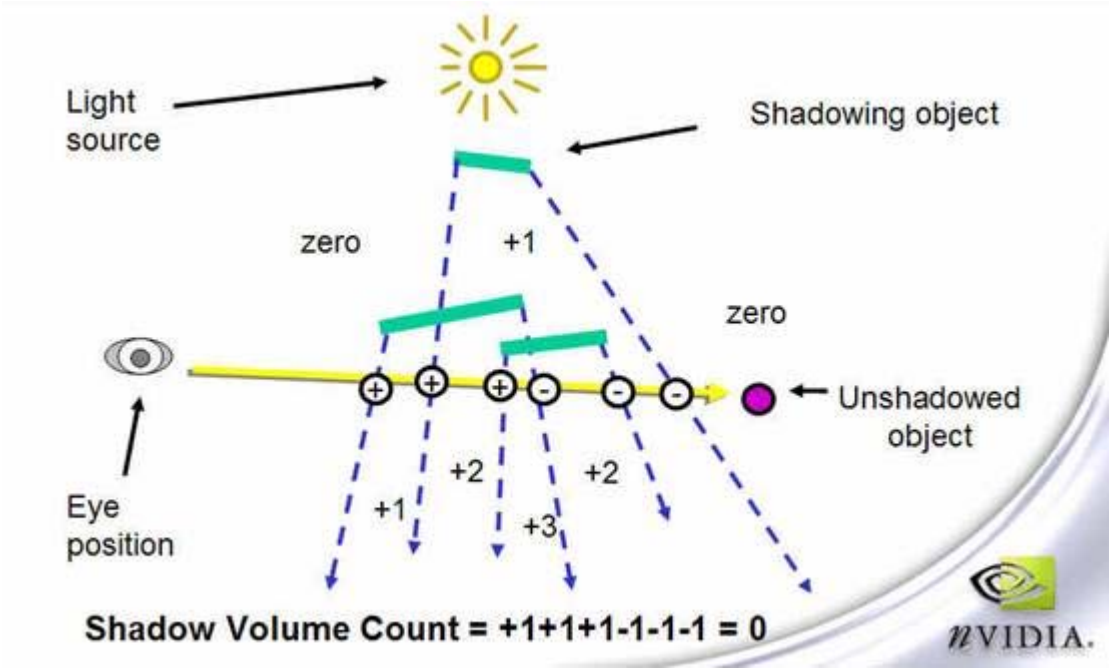
z-pass 是 shadow volume 一开始的标准算法，用来确定某一个象素是否处于阴影当中。其原理是：

Pass1：eNABlez-buffer write，渲染整个场景，得到关于所有物体的 depth map。注意这里的 depth map 和 shadowmapping 里面的区别是 shadow volume 里面的 depth map 是以真实视点作为视点得到的，而 shadowmapping 里面的 depth map 是以光源为视点得到的。

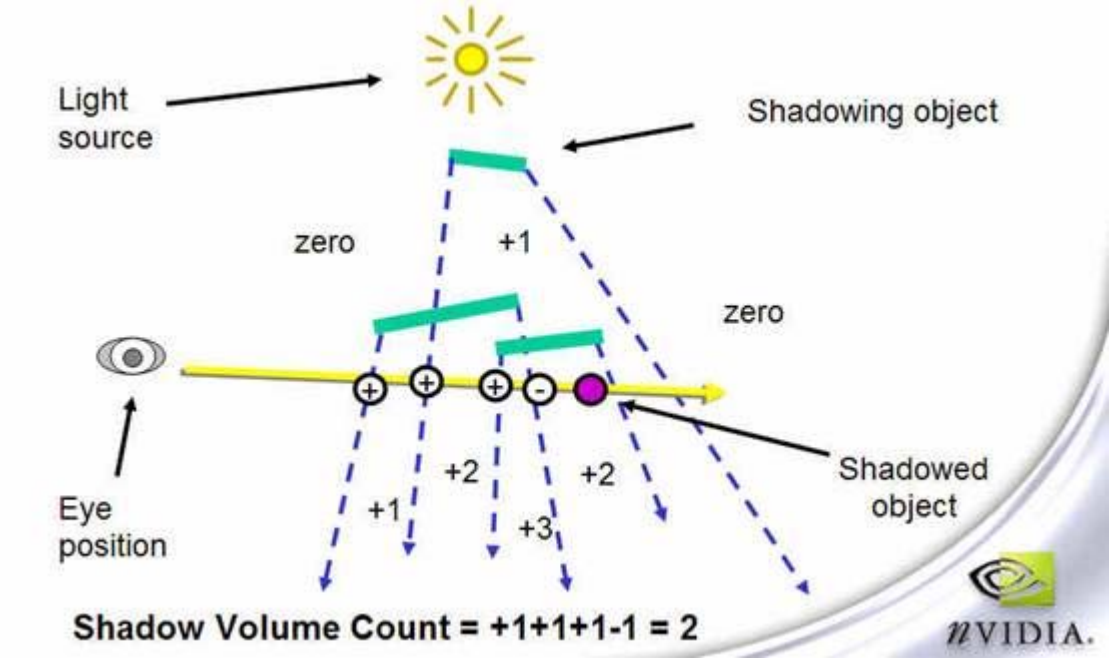
Pass2：disable z-buffer write， eNABlestencil buffer write, 然后渲染所有的 shadow volume。对于 shadow volume 的 frontface(既面对视点的这一面)，如果 depth test 的结果是 pass, 那么和这个象素对应的 stencil 值加一。如果depth test 的结果是 fail, stencil 值不变。而对于 shadow volume 的 back face (远离视点的一侧)，如果 depth test 的结果是 fail, stencil 值减一，否则保持不变。

用一句简单的话来概括 z-pass的算法就是从视点向物体引一条视线，当这条射线进入 shadow volume 的时候， stencil 值加一，而当这条射线离开 shadowvolume 的时候，stencil 值减一。如果 stencil 值为零，则表示实现进入和离开 shadow volume的次数相等，自然就表示物体不在 shadow volume 内了。

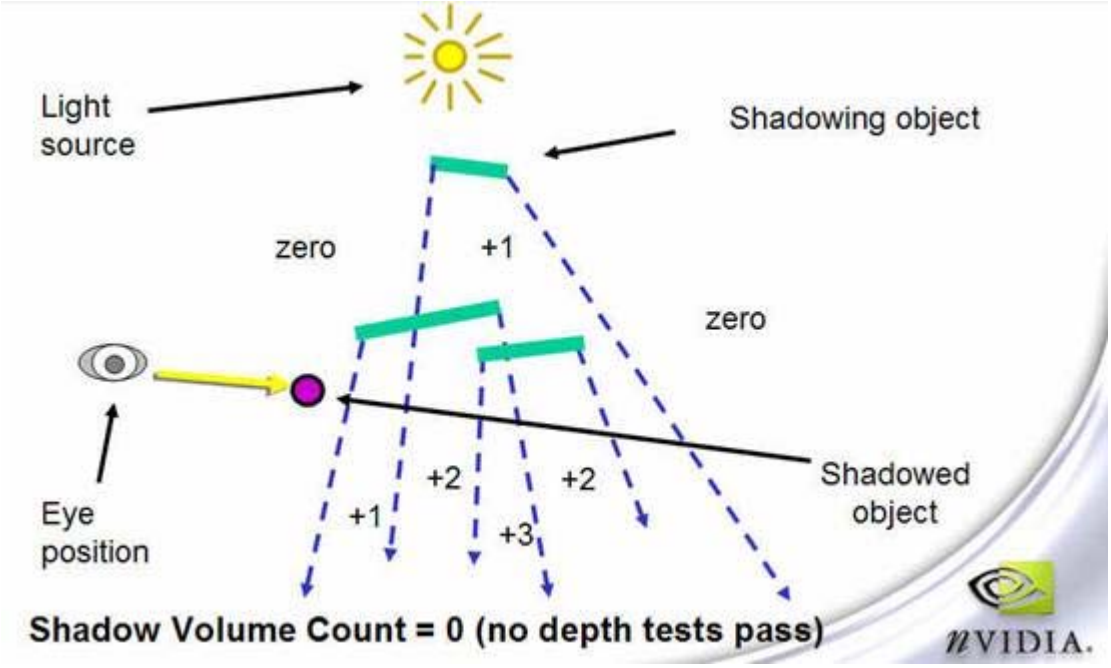
Pass3：第二步完成以后，根据每个象素的 stencil 值判断其是否处于阴影当中（如果 stencil 的值大于零，则这个象素在 shadow volume 内，否则在 shadow volume 的外面），然后据此绘制阴影效果。



在这副图里面，视线三进三出 shadow volume, 最后的 stencil 值为零，表示物体在 shadow volume 外，不受阴影的影响。



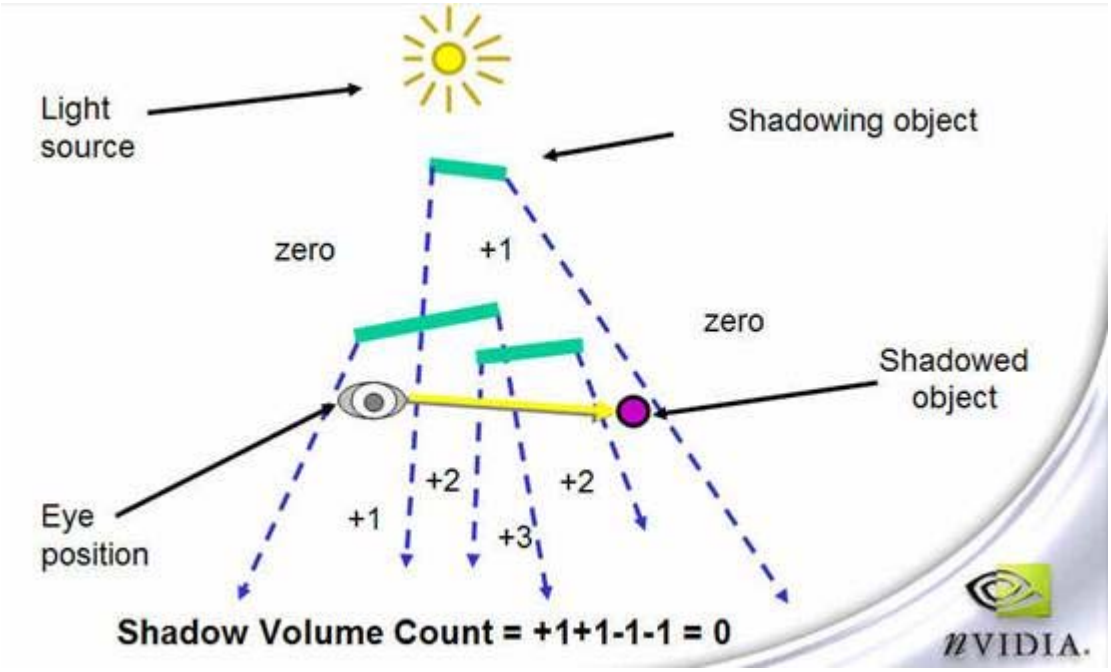
这副图里面视线三进一出， stencil 值为 2，表示物体在 shadow volume 内，有阴影产生。



这副图里面从视点到物体的视线中止于 shadow volume 前，也就是说所有的 z-test 都是 fail, 相应的 stencil 值为零，表示物体在阴影外面。

z-pass 算法缺点及补救办法

以上的讨论都是基于视点在 shadow volume 外面的情况。在这个条件可以得到满足的情况下，z-pass 算法工作的很好，不过一旦视点进入到了 shadow volume 里面，z-pass 算法就会立即失效。



这副图里面的视线二进二出，按照 z-pass的算法，最后的 stencil 值为 0，表示物体在阴影外，可实际上物体是处于阴影内的。错误的原因就在于视点进入到阴影内，使得视线失去了一次进入 shadow volume的机会，让原本应该是 1 的 stencil 值变成了 0 。

Z-Pass 这种错误的行为可以从下图中看出 ：

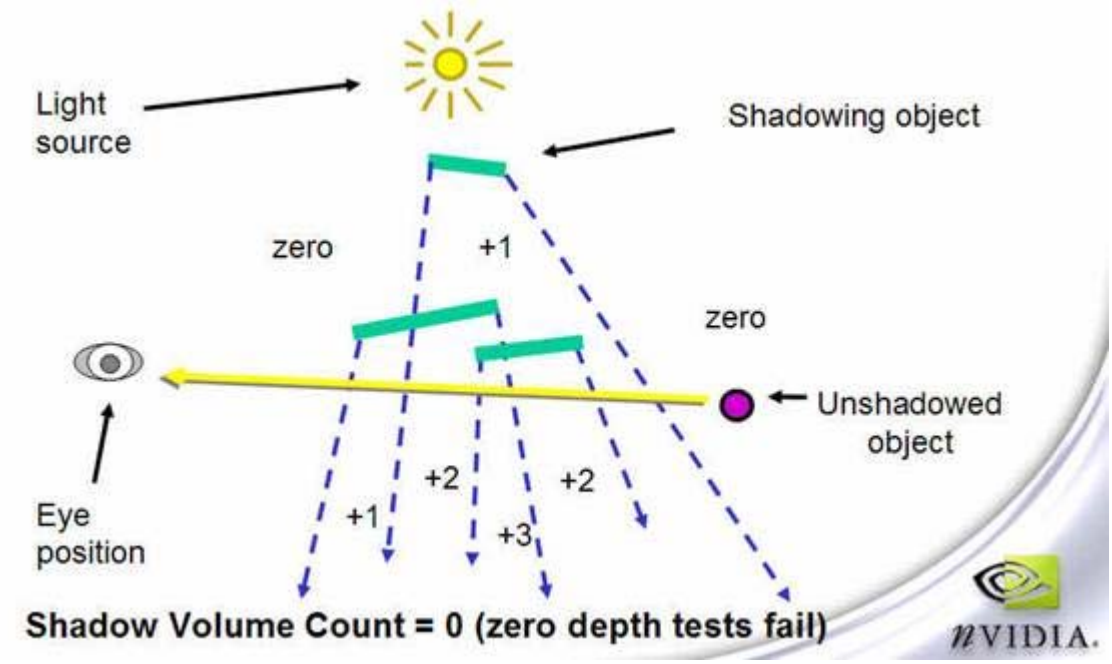
注意地下的影子

Z-Fail 算法：

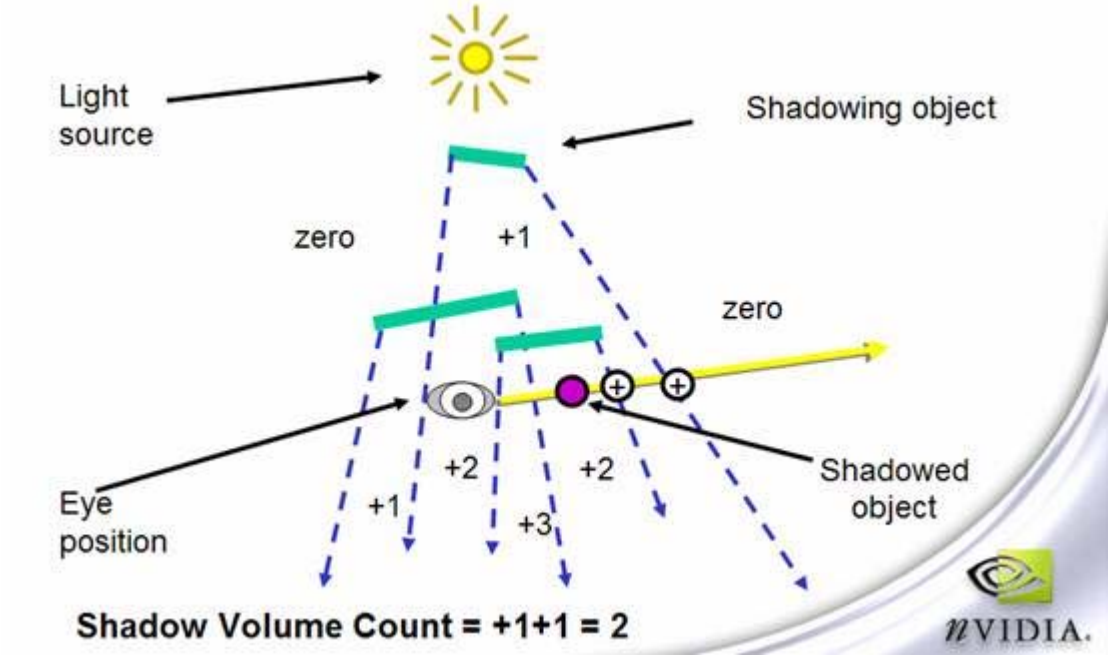
Z-Fail 算法是 John Carmack，Bill Bilodeau 和 Mike Songy 各自独立发明的，其目的就是解决视点进入 shadow volume 后 z-pass 算法失效的问题。

Pass1：eNABle z-write/z-test, 渲染整个场景，得到 depth map 。（这一步和 z-pass 的完全一样）

Pass2：disable z-write, eNABlez-test/stencil-write 。渲染 shadow volume, 对于它的 back face，如果 z-test 的结果是fail, stencil 值加一，如果 z-test 的结果是 pass, stencil 值不变。对于 front face, 如果z-test 的结果是 fail, stencil 值减 一 ，如果结果是 pass, stencil 值不变。

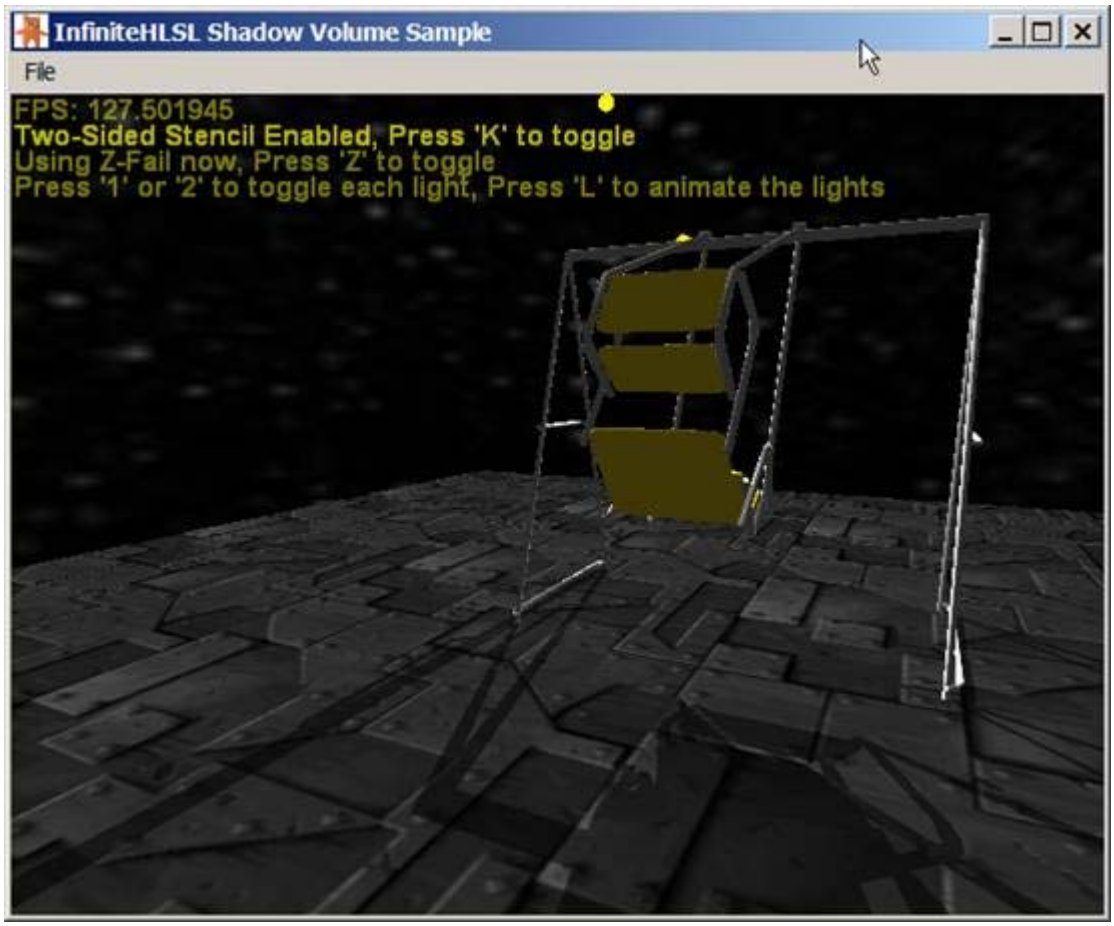


图中所有的 shadow volume 都处在 z-pass 的位置，因此 stencil 值不会改变。



视点在 shadow volume 内也没有问题，最后 stencil 的值是 2, 表示物体在阴影内。

上面那个 Z-Pass 无法处理的场景，用 Z-Fail 计算则可以得到正确的结果：

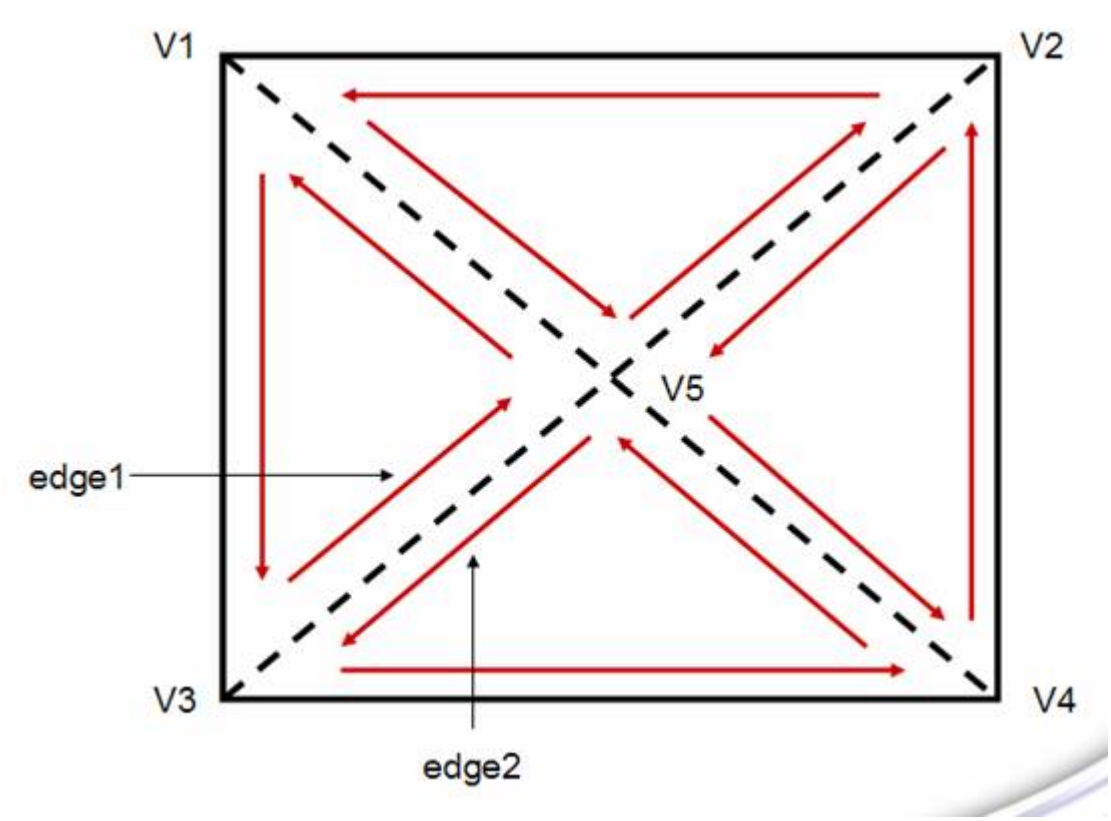


如何建立 shadow volume?

shadow volume的建立是整个算法里面最重要的部分，在 GPU 出现以前， shadow volume 的建立都是基于 CPU 的。随着 GPU应用的逐渐开展，人们又将 shadow volume 运算移植到了 GPU上，不过后面一种方法需要对物体的几何数据进行预处理，下面就对两种方法分别进行解释：

CPU based method（基于CPU建立方法）：

想必熟悉 shadow volume 的朋友对silhouette edge 这个词会很熟悉。它表示从光源的角度看物体所得到的轮廓线。 Shadow volume 就是由silhouette edge 扩展到一定距离以外或者无穷远处得到的。 silhouette edge的确定方法有很多种，基本思想就是找出那些被朝向相反（一个面向光源，另一个背向光源）的两个三角形（相对于光源来说）所共享的边，因为只有这样的边会最终成为 silhouette edge，其他的边在光源看来都在物体投影的内部而不是边缘。



这副图是一个由 4 个三角形组成的多边形，假设光源处在读者头部的位置，那么外围的一圈实线就是所谓的 silhouette edge 。我们所要做的就是从原始数据里面将内部多余的 4 条边 (虚线) 去掉。具体实现是这样：

- 遍历模型的所有三角形

- 计算 dot3(light_direction , triangle_normal) 。用这个结果判断三角形是面向光源 (dot3>0) 还是背向光源 (dot<0) 。

- 对于面向光源的三角形，将所有的三条边压入一个 栈 ，和里面的边进行比较，如果发现重复的 (edge1 和 edge2) ，将这些边删除

- 检测过所有三角形的 所有边 以后， 栈 里面剩下的 边就是 当前光源 / 物体位置下面的 silhouette edge.

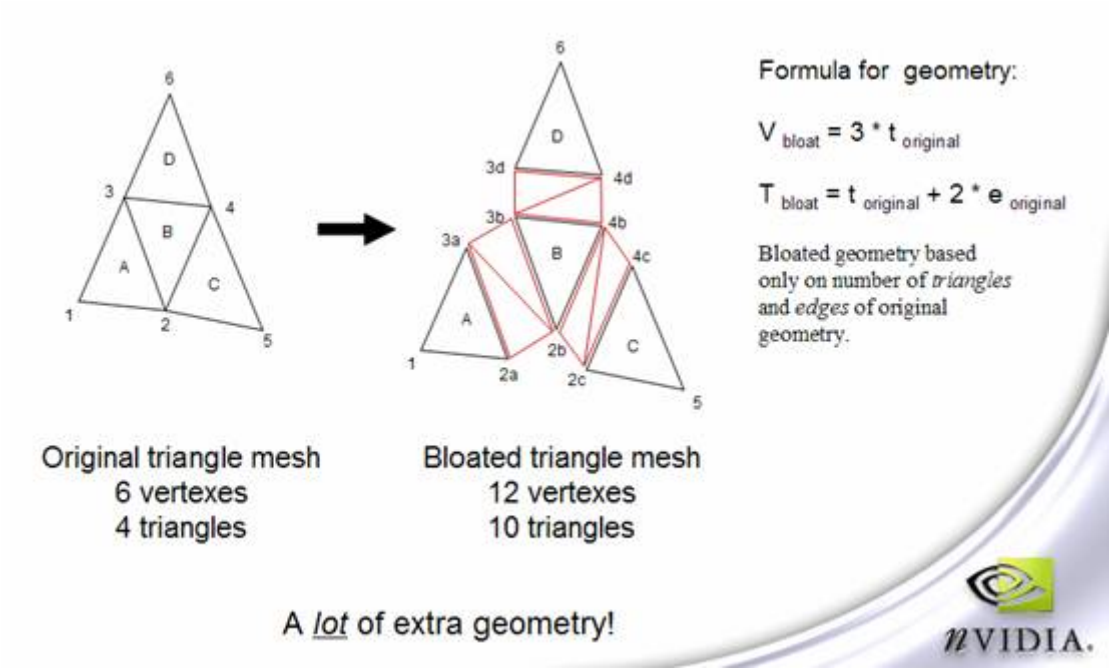
- 根据光源方向 ，利用 CPU 或者 vertex shader 将这些 silhouette edge 投射出去形成 shadow volume.

值得一提的是，这种方法正是 DOOM3所采用的方案，但是其中有一个问题就是 silhouette edge是由光源和物体的相互位置确定的，也就是说这二者之间有一个的位置发生了变化， silhouette edge就要重新计算，更新的数据也要传回显卡才能渲染 shadow volume ，这对 CPU 的计算能力以及 AGP的带宽不能不说是一个不小的考验。

GPU based method（基于GPU建立方法）：

Vertex shader一出现人们就在思考能不能利用它来加速 shadow volume 的渲染速度。但即使是现在最先进的 vertex shader 3.0也不具备创建新的几何物体的能力。简单点说 vertex shader 只能接受一个顶点，修改这个顶点的属性 (位置，颜色，纹理坐标，etc), 之后输出这个顶点到光栅化部分，继而进行 pixel shader 运算。碰到需要创建新顶点的地方，就只有依靠 CPU 直接操作vertex buffer 了。

另外一个方法就是事先把 shadow volume需要的空间留出来，然后再通过 vertex shader的运算使之外形达到我们需要的样子。这就好比我要存储一串数据，但又不很确定具体的规模是多大，只好事先分配一块很大的区域，这样不免会造成很大浪费，但也是不得以而为之。



由于物体上的每条边都有可能成为 silhouetteedge ，所以我们需要事先插入 degenerate quad(上图的红色三角形), 这些 quad的面积为零，不作任何变换的话是不可见的，不会造成视觉瑕疵。但是在需要的地方，可以把这些 quad 拉伸成为 shadow volume 的侧壁。

显然，插入冗余的顶点会造成极大的浪费。因为大部分的边最终 并不会成为 silhouette edge ，也就是说插入的 degenerate quad是无用的。不过这样做的好处是几何数据只需要传输到显卡一次，之后无论光源的位置在哪里，预处理过后的几何体都可以用来生成 shadowvolume ，不像刚才解释过的方法那样一旦光源和物体的相对位置发生变化，就需要重新用 CPU 计算 silhouette edge，之后再把结果 传送给显卡。

实际编程的时候，可以做一下改进，由于平坦的表面是不会产生阴影的，所以在这些表面所包含的边上就没必要插入 degenerate quad。而且所有的预处理应该在软件开发过程中完成，用户启动程序以后直接调用的就是插入过 quad 的模型，不需要 CPU 再进行计算。

建立/渲染 shadow volume 的 shader 代码：

```
// c0     : Light position in object space
// c1     : 1, 1, 1, 0
// c2- c5   : Light * View * Proj = LightClip
// c6- c9   : WorldInvLight matrix
// c10    : Color for exposing the shadow volume
vs.2.0
mov oD0, c10        // 输出特定的颜色使 shadow volume 可见
sub r1, v0, c0        // 光源方向
```

```
m4x4 r4, v0, c[6]        // 将顶点变换到光源坐标系
nrm r1, r1        // 光源向量归一化，这是为了 shadow volume 的各个边一样长
mov r10, c1
dp3 r10.w, v1, r1        //dp3 顶点法向量和光源向量，确定顶点的朝向
slt r10, c1.w, r10        // 根据 dp3 的结果设置 r10 寄存器的第四个单元
mul r4, r4, r10        // 设定 r4 的 w 位
m4x4 r5, r4, c[2]        // 输出顶点到 clip space
mov oPos , r5
```

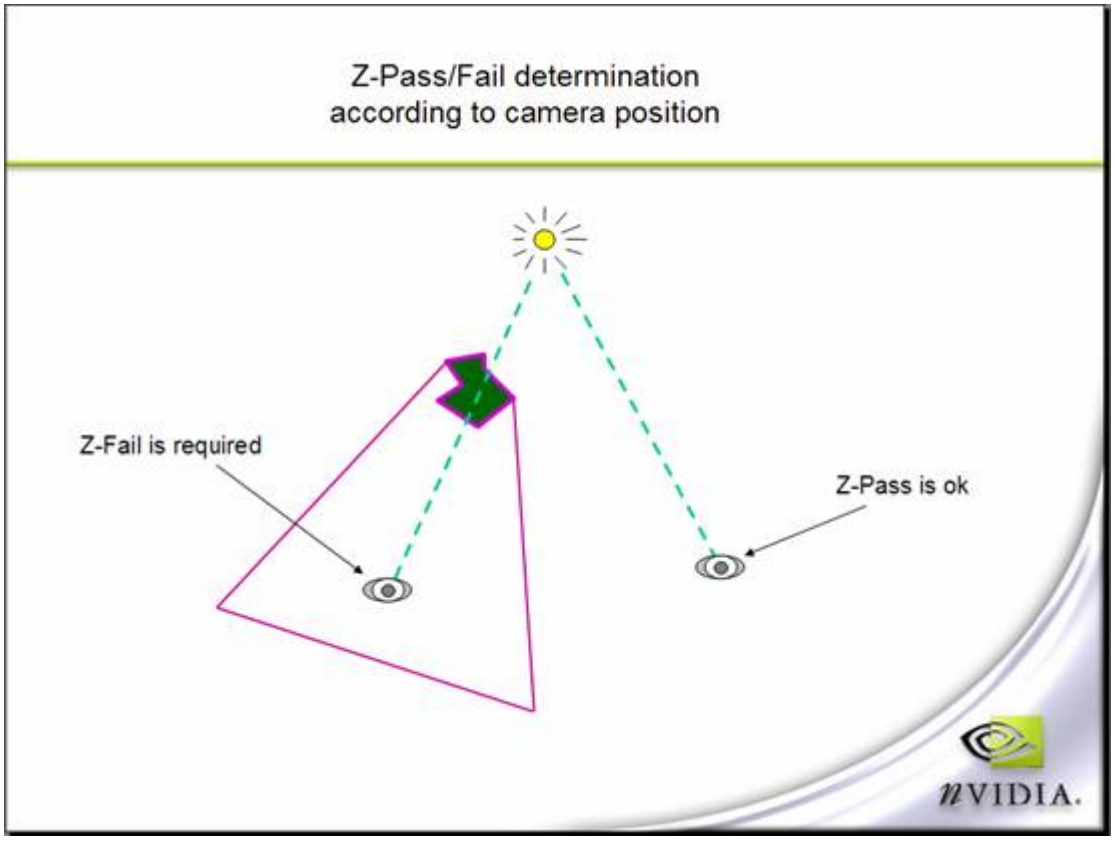
Shadow volume 的算法优化（一）

Shadow volume 的基本算法讲到这里就基本完成了，下面说一下现在比较常用的一些优化算法。

（一）Z-Pass .VS. Z-Fail

前面提到过，Z-Pass 比 Z-Fail 速度要快，因此我们可以在不会产生问题的场合下适当使用 Z-Pass 来提高性能，但是如何确定何时 Z-Pass 不会带来问题呢？ Z-Pass 失效主要是由于两种原因 ：

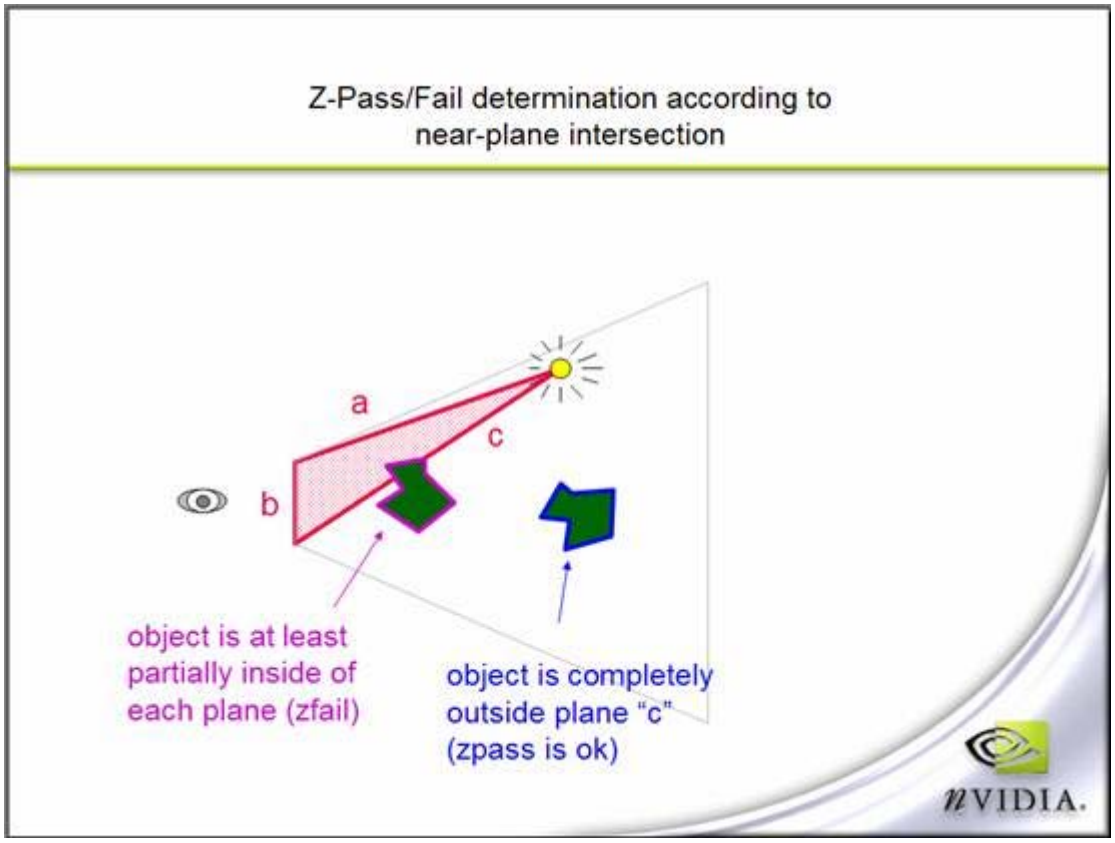
原因一：视点进入 shadow volume 内，比如下图：



只要能探测出这两种情况，就能在需要的时候切换到 Z-Fail 算法。条件 A 的判定可以参照下图，在视点和光源之间做一条连线，如果这条线和遮蔽物相交，那么可以肯定视点在 shadow volume 内，将切换到 Z-Fail 算法。

原因二：shadow volume 与近 剪裁面 相交

至于情况 B 的判定可以利用光源和近 剪裁面 形成的light-pyramid(红色阴影部分) 与遮蔽物的交汇关系。如果遮蔽物完全在 light-pyramid 之外，则由它生成的shadow volume 不会和近 剪裁面 相交，可以使用 Z-Pass 算法，否则将只能使用 Z-Fail 算法。



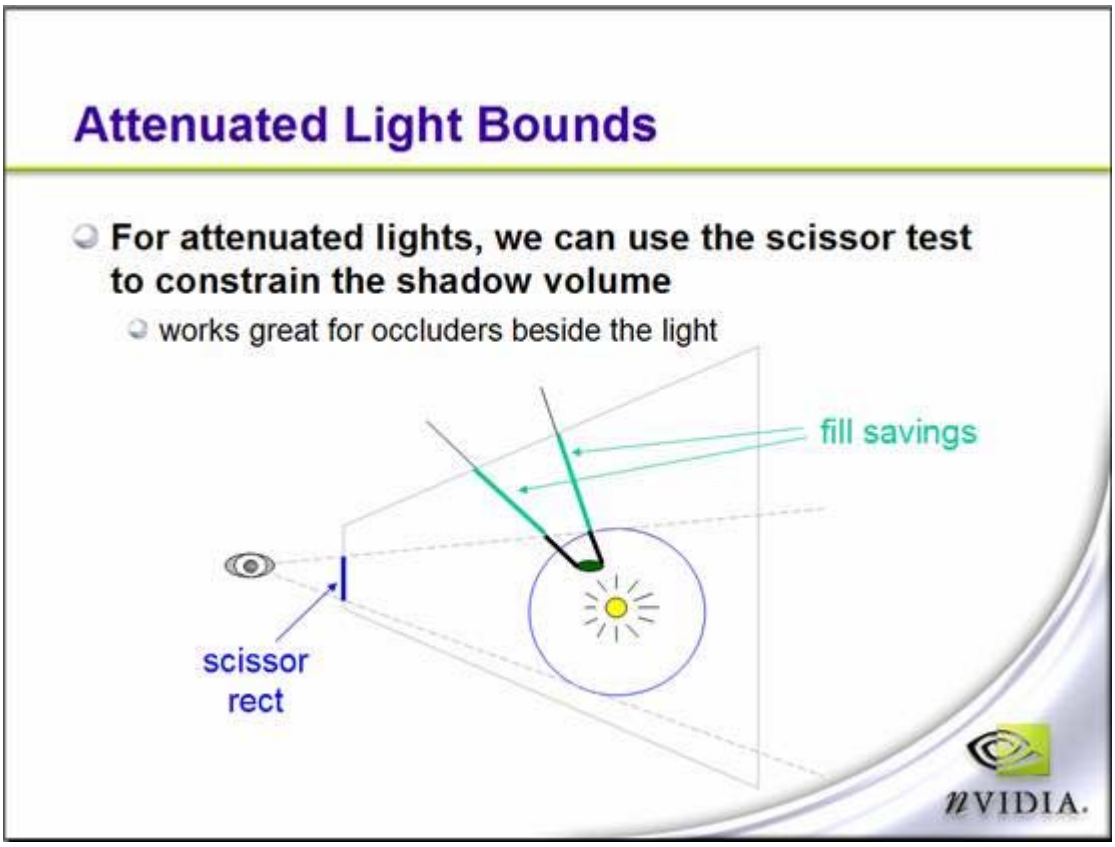
Shadow volume 的算法优化（二）

（二）tricks to save fillrate :

前面提到过，shadow volume算法里面两个最耗时的步骤就是 silhouette edge determination 和 shadow volume rendering。其中 shadow volume rendering 是完全考验 GPU 填充率的步骤，虽然现在的显卡动辄就有几十 G fragment/s的填充率能力，但是遇到复杂的场景，流水线也不免不堪重负。此外，频繁的 stencil buffer操作也会占据一部分显存带宽，如果能够找出一些办法尽量减小 shadow volume 的尺寸，将会是效果很明显的一种优化方法：

限定光照的范围（Attenuated Light Bounds）：

如果所用的光源有衰减效应，则可以利用 scissortest 将渲染的范围限定在光源的作用范围之内，因为超出了这个范围就不会有阴影存在，自然用不着去渲染那部分的 shadow volume了。所谓 scissor test 就是人为地在屏幕坐标系下面定义一个矩形，只有坐标处在这个矩形范围内的 fragment才能够通过测试，其内容才能被写入 帧 缓存。

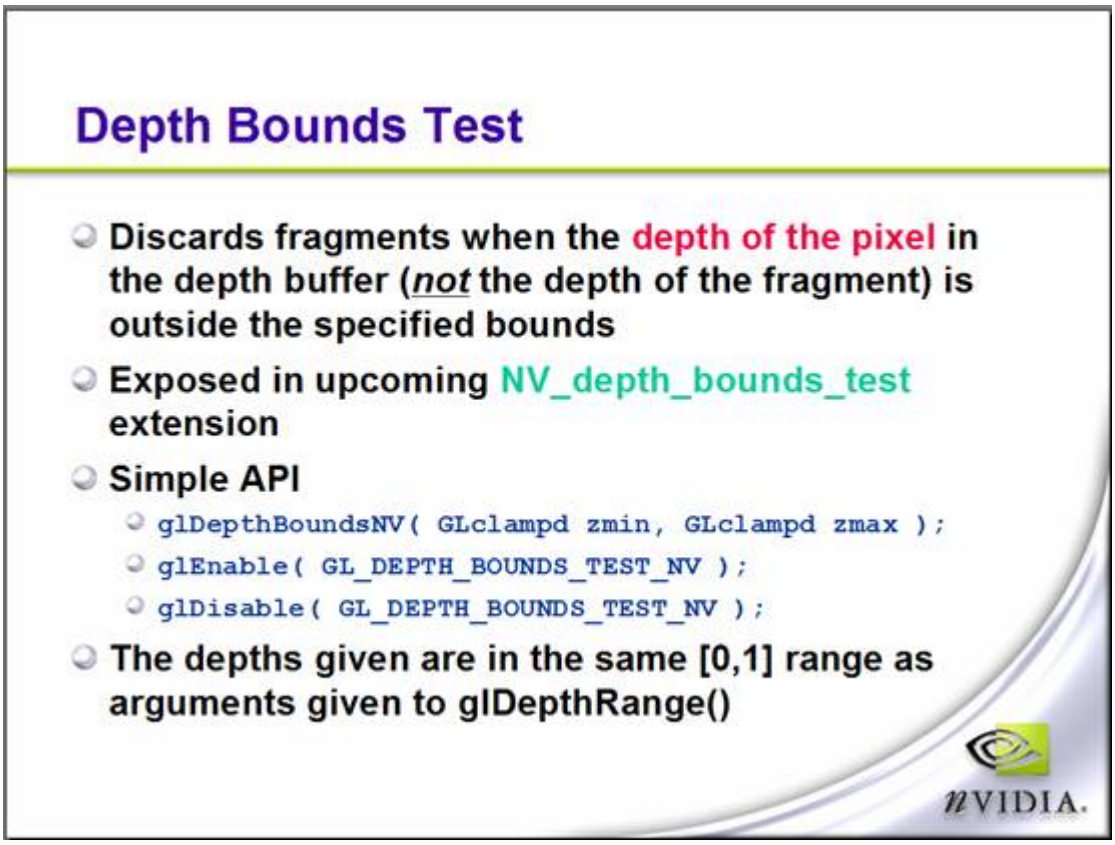


NVIDIA的阴影加速技术（ultra shadow）：

ultra shadow这项技术是随着NV35 的发布而浮出水面的，进而在 NV36/38 中得到了继承，我们基本上可以在 NVIDIA 今后的产品中，这项技术会得到持续的应用。

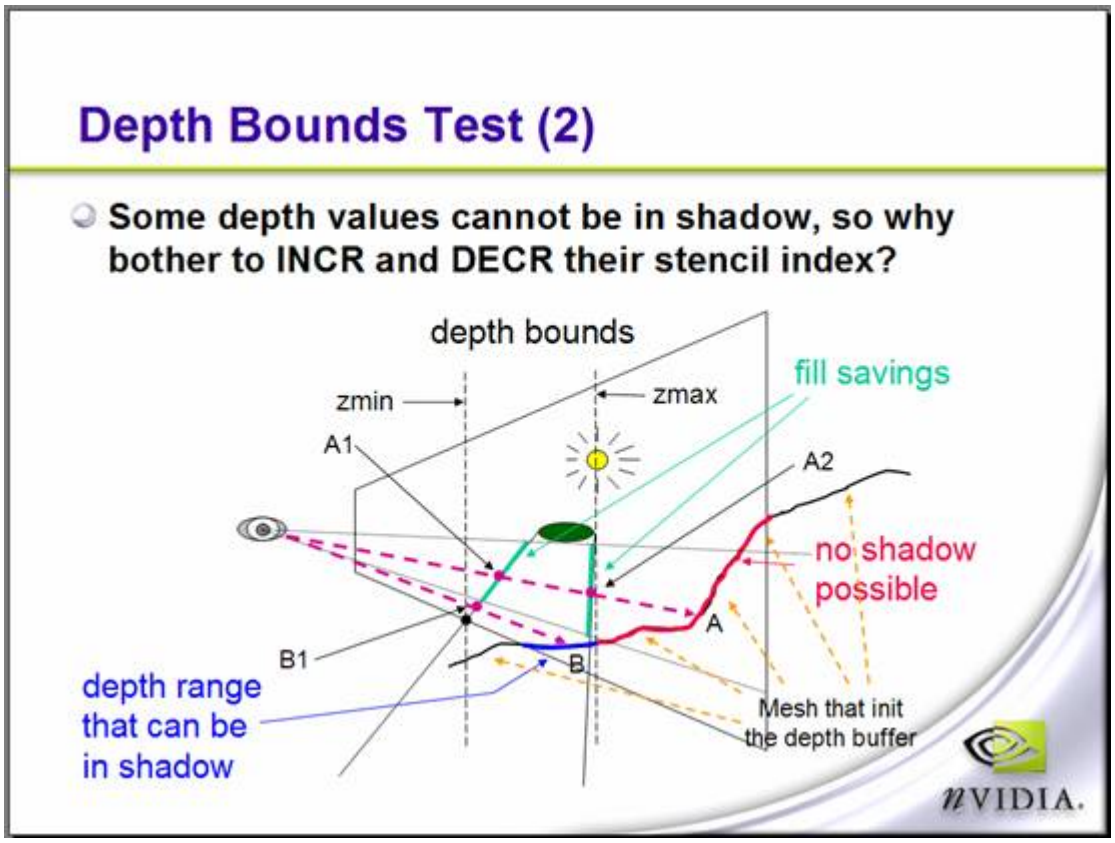
id software 的当家程序员 JohnCarmack 曾经说过 NV35 是为 DOOM3 量身打造的 GPU，我们在这里有理由怀疑 Carmack说这番话的原因很有可能就是由于 NV35 中集成了 ultra shadow 阴影加速技术（近日GeForceFX系列已经成为DOOM3的推荐GPU），那么 ultra shadow 究竟是什么，它如何加速阴影的渲染速度呢？

其实 ultra shadow 技术仅仅利用了一个 NVIDIA 新近提交的 OpenGL 扩展—— EXT_depth_bounds_test，我们先来看一下 NVIDIA 官方在 GDC2003 上对这个扩展的介绍：



首先注意一下名称的问题，GDC2003在三月举行，那时这个扩展还只是 NVIDIA 独家的东西，到了 4 月这个扩展更名为 EXT_depth_bounds_test 。 EXT开头的扩展表示有多家厂商在开发这项技术，也许不久以后我们就会看到 ultra shadow 在 ATI 的 GPU 上面实现。

Depth bounds test 的作用是比较由当前 fragment 的屏幕坐标（ xw， yw ）指定的 depth buffer 中的 z 值与用户通过 glDepthBoundsNV（GLclampd zmin , GLclampd zmax)所指定的 [zmin,zmax], 如果 z 值在次范围之外，则将当前的 fragment 从流水线中剔除掉，不进行此处的 stencilbuffer 操作。注意这里比较的并不是 fragment(shadow volume) 的 z 值，而是前一个 path 中已经渲染过的shadow receiver 的 z 值。具体情况请看下图：

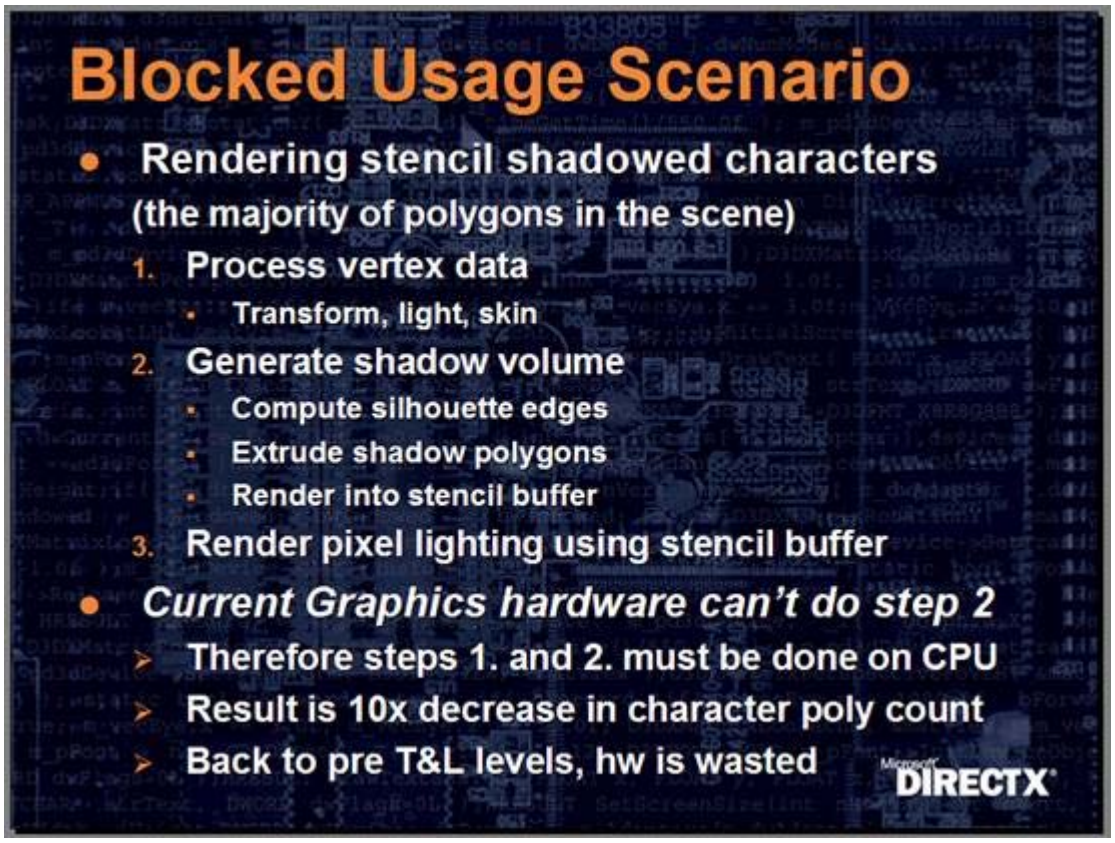


可以看到，由于 A 点的 z 值在 [zmin,zmax] 范围之外，此点没有可能被阴影遮住，因此 A1/A2 点处的 fragment 就可以被丢弃。而 B 点的 z 值在 [zmin,zmax] 之外，所以 B1 点处的 fragment 就必须进行 stencil buffer 操作。

（详细的技术介绍请看：《NVIDIA的复仇计划 GF FX 5900 Ultra》）

阴影渲染实现技术的展望

shadow volume是近阶段实现统一光照模型比较好的一种技术，现在主要的问题是基于 CPU 的方法对处理器依赖比较重，在 AI/ 物理运算较多的场景中 CPU的运算能力可能不足，而基于 GPU 的方法效率太低，会产生大量的冗余顶点，其原因还是由于现在的 GPU(包括即将发布的 NV40/R420)都不具备在芯片内部产生新顶点的能力。 Microsoft 意识到了这一点，在 DirectX Next的发展规划中将这种能力列为了要实现的目标之一：

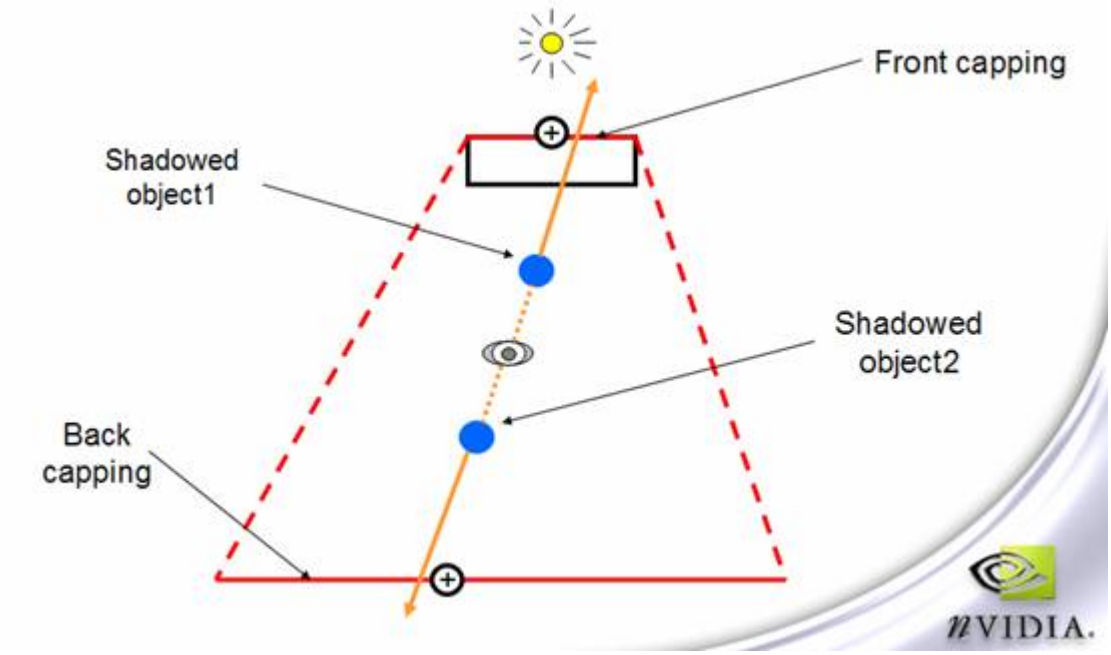


从更长远的角度来说，基于真实物理模型的光照模型（比如spherical harmoniclighting、ray-tracing、radiosity）才是发展的方向，那时我们没有必要设计单独的算法来实现阴影，所有的光照/阴影效果都被包扩在了一个统一的光照模型之中，任何效果实现起来都是自然而然的，就像它们在真实世界中的情况一样。当然，所有这些设想都要基于半导体生产技术的支持才行，我们在近期（5-10年）将不会看到它们在硬件上的实现。

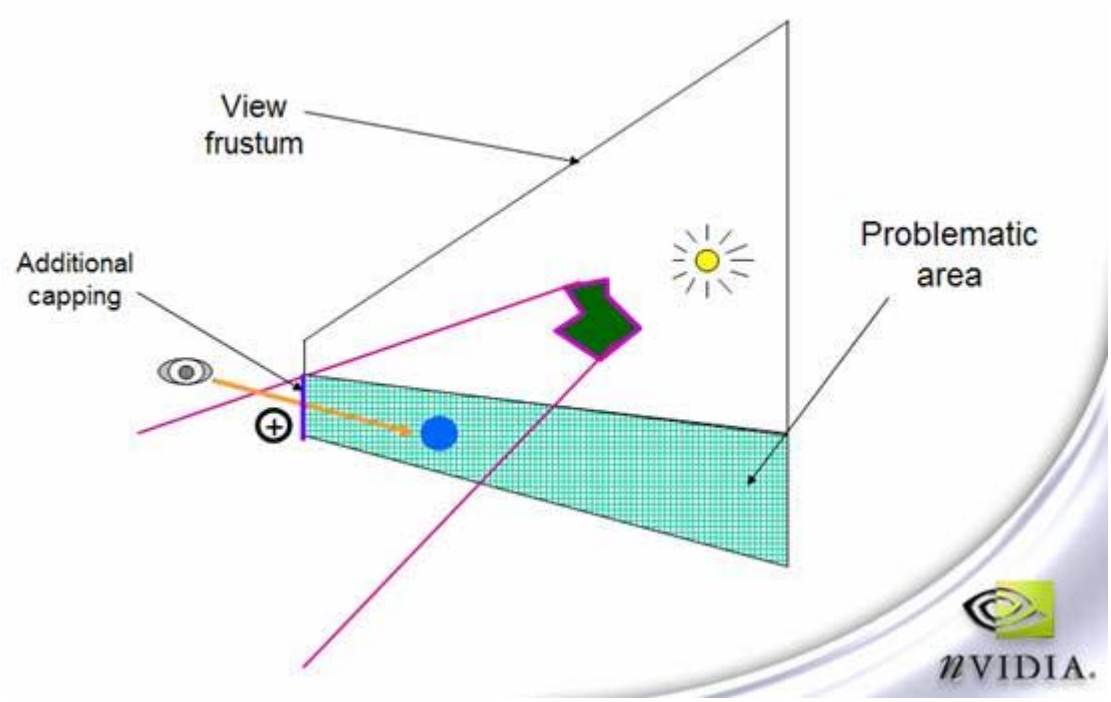
使用 z-Fail 算法的条件

Capping For Z-Fail

由于 Z-Fail 算法依靠计算 shadowvolume 不能通过 Z-test 的部分来确定 stencil buffer 的值，所以要求 shadow volume是闭合的。下面的那张图里面红色的实线表示 capping, 可以想象，假如不人为的添加 capping, 那么 shadow object1/2 的 stencil 值都会是 0，而实际上正确的 stencil 值应该是 1，因为它们都在阴影内。



Z-Pass 和近剪裁面的关系：



在 Z-PASS 算法中，当 shadowvolume 和视图体 (view frustum) 发生剪切关系的时候，需要附加的 capping 才能保证最后的结果正确。因为经过view frustum 的剪裁作用以后，shadow volume 的一部分有可能变成敞开的，比如在图中 additionalcapping 的位置，假如不人为的附加一部分多边形，在渲染 shadow volume 的时候 stencil buffer 就不会发生+1 的操作 (因为这里没有任何多边形，自然也就不会和原来的 depth map 比较)，最后的结果显然是不对的。