阴影技术

阴影在渲染场景中的具有非常重要的作用，对于那些具有真实感的场景尤其重要。场景中的阴影带给我们的不仅仅是物体在三维空间中的位置关系，而且能够带给我们真实的体验。但是，阴影技术是三维渲染中的一个最重要的难题之一，当前对这一领域的研究也非常火热。现在已经有了多种技术来渲染阴影，但是没有一个是完美的，各个方案总是伴随着优点和缺点。

一、目前的实现方法

当前的阴影的实现方法从阴影的形状如何生成来分，主要可以归为两类：一类是模板阴影（stencil shadows），另一类是纹理阴影（texture shadows）。

首先我们需要知道阴影产生的条件，产生阴影首先必须要有光线（light），漆黑的没有月光和灯光的夜晚我们是看不见阴影的；其次还需要两个重要的参与者，遮挡物（occlusion）和阴影接收者（shadow receiver）。顾名思义，阴影产生者是物体挡住了光线的传播路径，形成了一片阴影区域；阴影接收者则是处于阴影产生者产生的阴影区域中的物体。所以，阴影产生的三个比较条件是：光线，阴影产生者和阴影接收者。

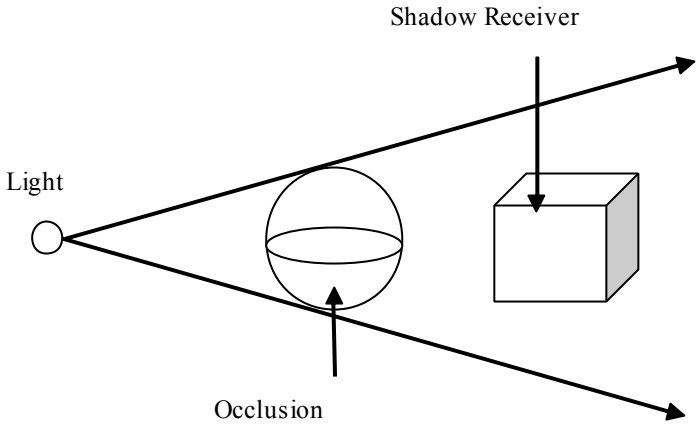


图 1　阴影三要素示意图

模板阴影技术利用模板缓存来实现，我们知道帧缓冲区（frame buffer）中有一个非常重要的缓冲区叫做模板缓存（stencil buffer），模板缓存和颜色缓存（color buffer）是通过像素一一对应。模板阴影技术通过多个绘制通道（pass）来实现，其中一种方法是：pass 1 绘制出没有阴影效果的场景，数据保存在颜色缓存中；然后 pass 2 通过一定的技术在模板缓存中以标志（mask）的形式绘制出阴影的形状；最后 pass 3 利用模板缓存中的数据对颜色缓存进行调整，或者是用黑色数据替代原有数据，或者是对原来的数据进行调整，这是后面将要详细介绍的阴影体（shadow volume）方法，另一种方法是阴影贴图（shadow mapping），在后面也会进行介绍。

纹理阴影和模板阴影的思路大不相同，它是通过将视点移动到光源的位置，然后形成一张投影纹理贴图（projection texture），最后把这张纹理贴图应用到 shadow receiver 上。

和模板阴影技术相比，纹理阴影的一个优点是速度快，大多数的操作都是由图形显卡来完成的。因为纹理阴影不需要进行大量的几何计算，这和模板阴影正好相反，所以不论显示的模型的几何复杂程度如何，纹理阴

影的开销都变化不大；纹理阴影的另一个优点是用户定制化程度较高，当获取到阴影的纹理贴图后，用户可以随需要进行任意的操作，可以在 shader 程序中对纹理进行过滤处理来形成软阴影（soft shadow）或者其他特殊的效果。但是，纹理阴影的缺点也很明显，就是它不够精确。因为我们得到的仅仅只是一副纹理贴图，我们在获得它的时候它的分辨率就已经确定了，当我们需要拉伸的时候就显得非常不方便。

相比较而言，模板阴影则更加精确，因为是对像素进行操控。但是模板缓存的缺点也很突出，就是伴随着显示模型中阴影产生者的几何复杂程度的增加，模板阴影的计算量也会随着提升。模板阴影需要对每个模型中的每个图元（primitive，一般是三角形）进行计算，找到模型的边缘以构建边缘轮廓（silhouette）。并且模板阴影的另一个缺点是“硬”阴影（hard shadow），就是说阴影和光照区域的分界十分明显。

二、模板阴影体（stencil shadow volume）

2.1 模板阴影体基本概念

使用阴影体的方式构建模板阴影是在图形引擎中使用得相当广泛，称为 stencil shadow volume。Stencil shadow volume 利用模板缓存来辅助构建阴影，模板缓存中阴影数据的构建是通过阴影体技术来实现。Shadow volume 是 Frank Crow 在 1977 年在其论文《SHADOW ALGORITHMS FOR COMPUTER GRAPHICS》提出的一个理论观点，Silicon Graphics 公司的 Tim Heidmann 利用模板缓冲区的技术将该理论付诸实现。

Stencil shadow volume 的基本思想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。检测的方法也很容易理解，如果物体在阴影中，那么在绘制 shadow volume 的时候，一定有一面会通过深度测试而另一面不能通过深度测测试。下图是 shadow volume 的示意图，其中的灰色区域就是阴影体（shadow volume）。

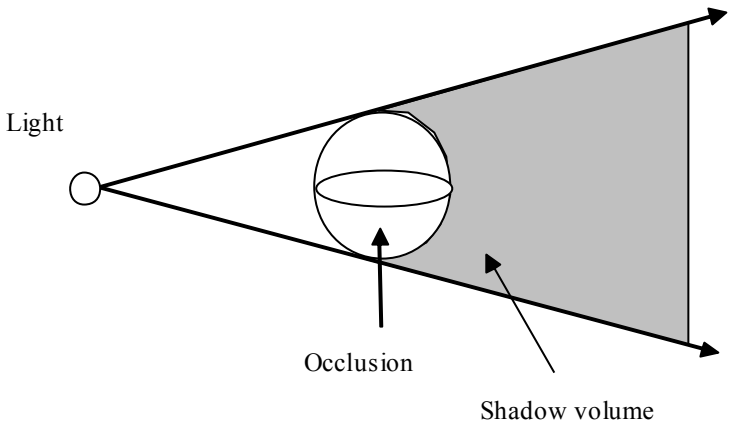


图 2　shadow volume 示意图

和我们现实中的情况一样，那些遮挡光线传播从而投射出阴影区域的物体叫做遮挡物（occlusion），如下图所示。其中轮廓线（silhouette）是物体上是否能够接收到光照照射区域的分界线，在下图中时一个圆环形状，shadow volume 就是通过从光照点向轮廓线上的点引出延长线而得到的区域。

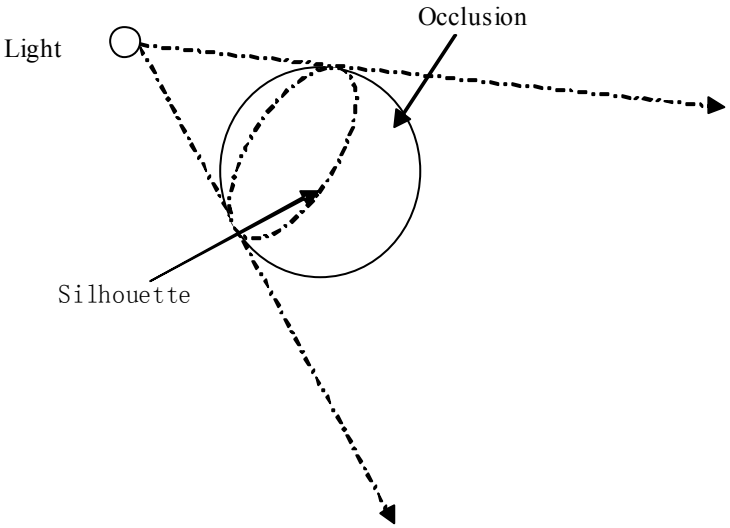


图 3　遮挡物轮廓线示意图

值得注意的是，对于不同种类的光源，shadow volume 的形状也不同。对于电光源来说，就是对点到点的连线延长就好，最终这些延长线之间的距离会原来越远；对于无限远的方向型光源来说，这些延长线最终需要在一个点出汇合。前者形成的 shadow volume 是无限远的，成为无穷阴影体（Infinite Shadow Volume），后者成为有限阴影体（Finite Shadow Volume）。

2.2 shadow volume 算法实现

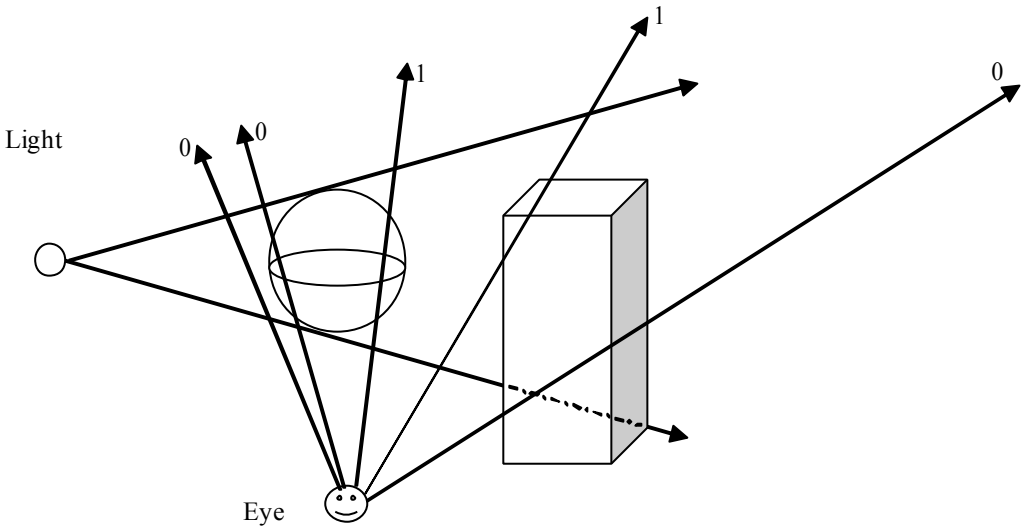


图 4　Depth-Pass 算法示意图

上图演示一个观察者对场景中不同方向的观察情况，视线箭头后面的数字表示渲染完 shadow volume 后模板缓存中的值。模板缓存中值为非 0 的片段（fragments）被认为是在阴影中的部分，而为 0 的片段则不处于阴影中。如前所述，shadow volume 是通过模板缓存来实现的，具体如何实现现在一般存在两种方法，都是为

了实现这样的想法：渲染 shadow volume 前将 stencil buffer 清零；当物体在 shadow volume 内的时候，渲染完 shadow volume 后 stencil 值大于 0；当物理不在 shadow volume 内的时候，则渲染完毕后 stencil 值保持为 0 不变。这样最后根据 stencil 的值来绘制出阴影效果。

在介绍具体的算法实现之前，我们先介绍具体的场景渲染过程，这样便于我们理解后面的算法在具体绘制过程中所起的作用。在渲染具有阴影效果的场景时，通常需要多个渲染通道（pass），具体如下：

Pass 1：enable depth-buffer write，渲染整个场景。这样会在帧缓冲区（framebuffer）中得到没有阴影效果的场景，和场景的深度缓存信息，这个对后面有用；

Pass 2：disable depth-buffer write, disable color-buffer(framebuffer) write, enable stencil buffer write，渲染 shadow volume。这个地方就是我们下面将要介绍的 Depth-Pass 算法和 Depth-Fail 算法将要起作用的地方了。渲染完成之后，我们便在 stencil buffer 中得到了整个场景的阴影信息。

Pass 3: pass 2 完成之后，根据每个像素的 stencil 值来判断该像素是否在阴影中（stencil value 大于 0 则像素在 shadow volume 内，否则在 shadow volume 外），然后根据这个来绘制阴影效果，具体的绘制效果和纹理混合的操作类似，可以选择替换也可以选择对原像素进行调整等，这里不做介绍。

下面介绍 pass 2 中需要使用的两个算法，Depth-Pass 算法和 Depth-Fail 算法。

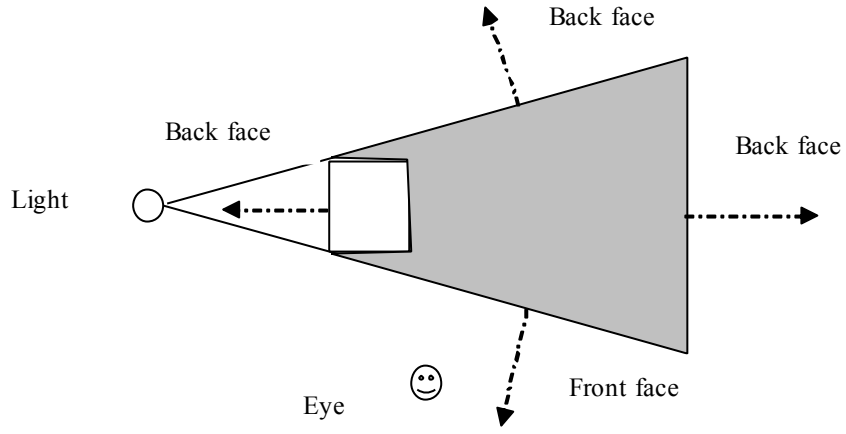
Depth-Pass 算法：

1. 渲染 shadow volume 的正面（front face）：如果通过了深度测试，stencil value 加 1，否则 stencial value 不变；

2. 渲染 shadow volume 的背面（back face）：如果通过了深度测试，stencil value 不变，否则 stencial value 减 1；

我们可以这样来理解 Depth-Pass 算法：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 1，而当这条射线离开 shadow volume 的时候，stencil 值减 1。如果 stencil 值为 0，则表示视线进入和离开 shadow volume 的次数相等，说明物体不在 shadow volume 内，否则物体便处于阴影之中。

这里的正面和背面指的是 shadow volume 的片元（primitive）和视点的关系，若片元正对视线则表示是正面（front face），否则表示背面。所以这里我们需要一个条件，就是在绘制 shadow volume 的时候，所有片元的法线都是指向 shadow volume 外面的，所以绘制这些片元的时候需要注意片元的顶点顺序。如下图所示：



：片元法线方向

图 5　shadow volume face 示意图

这个算法由于是在通过深度测试的时候给 stencil value 加一，所以取名为 Depth-Pass，也可以称之为 Z-Pass 算法。

上述算法看似简单，但是在大多数情况下都能非常好的工作，我们可以用图 4 来说明。虽然前面在渲染过程中我们已经提到了，但是还是要提醒下。在使用该算法操作 stencil buffer 时，我们已经将物体渲染到 framebuffer 中了（pass 1），并且为了实现深度测试（depth-test,or z-test)我们已经设置好了深度缓存（depth buffer）。图 4 中最左边的两条视线，由于没有和 shadow volume 的任何部分接触，因此 stencil value 保持为 0，因此该视线对应的片段（fragment)不需要阴影处理；从左边数的第三条光线，在 front face 部分，通过了深度测试（shadow volume 的图元深度比物体的深度浅），而 back face 部分则没有通过深度测试，因此最后的 stencil value 等于 1，意味着该视线对应的片段（fragment）需要加上阴影效果；从右边数的第二条视线和第三条视线的情况相同，因此 stencil value 等于 1，需要阴影处理；最后一条光线，由于 front face 和 back face 的深度测试都没有通过，因此也没有阴影。

那么 Depth-Pass 算法在多个 shadow volumes 的情况下还能正常工作吗？答案是，能。我们可以通过下图

来例证。

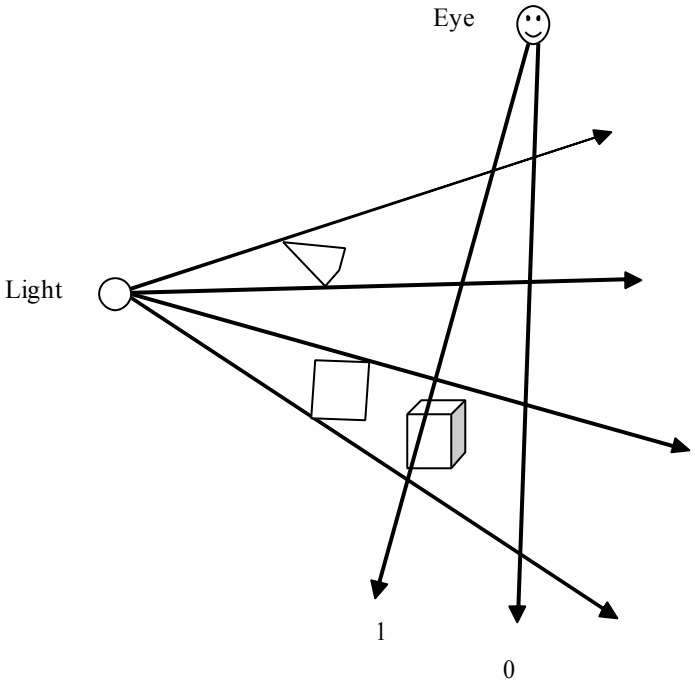


图 6 多个 shadow volume 情形示意图

但是 Depth-Pass 算法在一定的情况下会失效，当视点（view point）是在阴影内部时，该算法就不能工作了，如图 7 所示。

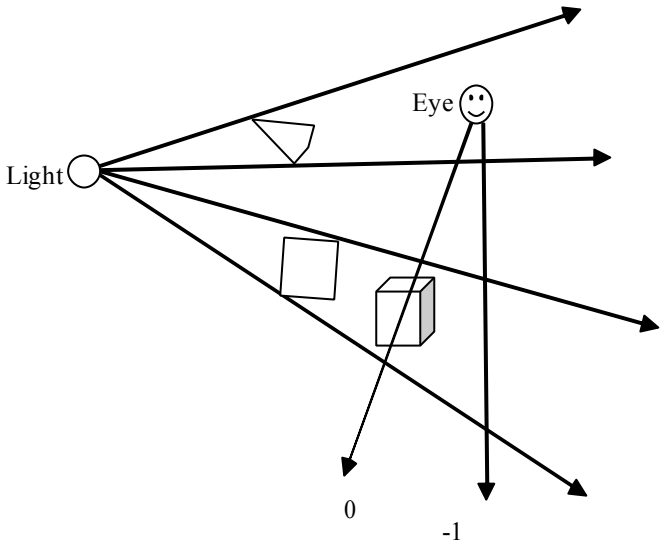


图 7 Depth-Pass 失效情形示意图

所以 Depth-Fail 算法应运而生，该算法是由 John Carmack，Bill Biodeau 和 Mike Songy 各自独立的发明的，该算法的提出就是为了解决当视点进入到阴影区域的时候的 stencil value 的计算。

Depth-Fail 算法：

1. 渲染 shadow volume 的背面（back face）：如果没有通过深度测试，则 stencil value 加 1，否则 stencil value 不变。

2. 渲染 shadow volume 的正面（front face）：如果没有通过深度测试，则 stencil value 减 1，否则 stencil value 不变。

该算法由于是对 Depth-Pass 算法的一个改进，并且是在深度测试失败时对 stencil buffer 进行操作，因此称为 Depth-Fail，也称为 Z-Fail，有时也叫做“Carmack's Reverse”。

我们可以通过下图来验证当视线进入 shadow volume 时，Depth-Fail 算法的工作情况。

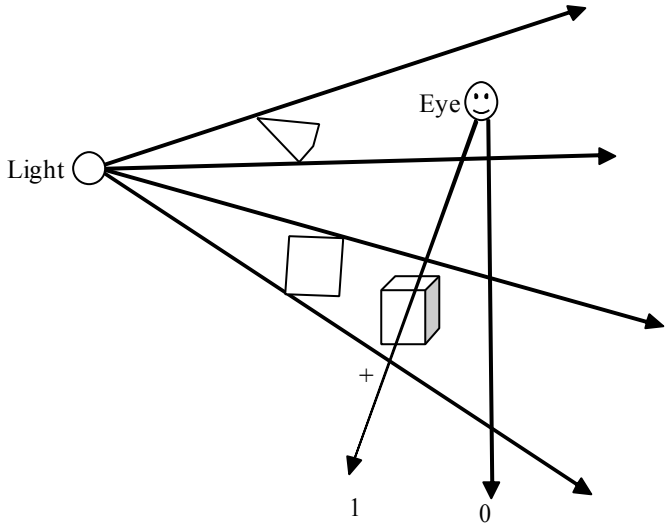


图 8 Depth-Fail 算法示意图

当视点不在 Shadow volume 里面的时候，Depth-Pass 算法和 Depth-Fail 算法都能够很好的工作。但是，在某些场景下还是会失败，这个在后面会有描述。

2.3 构造 shadow volume 的侧面

构造遮挡物（occluder）的 shadow volume 的时候，第一步就是要计算遮挡物的轮廓线。Stencil shadow 算法要求遮挡物必须是封闭的三角形网络，这意味着模型中的每条边都是被两个三角形所共享，同时不允许模型的中间有空洞的存在。我们计算轮廓线的时候，我们仅仅关心那些那些共享一条边但是和朝向光源的方向不同的边，比如说边 AB 相邻的两个面分别是面 a 和面 b，若面 a 朝向光源而面 b 逆向光源，或者说面 a 能够接收到光源照射，而面 b 接收不到，则面 a 和面 b 所共享的这条边便是这个物体相对于该光照的一条轮廓线。而当两个边同时朝向光源或者同时逆向光源的时候，则共享的边不是物体相对于该光照的轮廓线。构造遮挡物 的 shadow volume 的时候，就是要找出物体相对于光源的所有轮廓线，通过这些轮廓线和光源的连线来构造 shadow volume。

判断三角面和光源的面向关系非常简单，分别令三个点为点 A,点 B 和 C，且正面的顺序为 A,B,C，令光源为 L。则利用向量的乘法便可算出：（（A+B+C)/3 - L)·((B-A)×(C-B))，若结果大于 0 则表示面向光源，否则表示逆向光源。

这里介绍两种寻找轮廓线的简单方法：

第一种方法：

1. 遍历所有的三角形面，计算出这些面的朝向；
2. 遍历每条边，对相邻的面进行判断，朝向不同就选择，否则放弃；

第二种方法：

1. 遍历所有的三角形面，计算出面的朝向；
2. 当面朝向光源时，将该面的三条边插入到一个边的栈中；
3. 对插入进去的边进行检查，看它是否已经在栈中；
4. 若在栈中，则从栈中除去这两条相同的边；
5. 最后留在栈中的边就是组成轮廓线的边。

这里有一点必须要注意，选定轮廓线操作是构建 shadow volume 时代价最昂贵的操作之一，另一个是更新模板缓存时为每个光照进行的一次 shadow volume rendering pass 操作。如果想要优化阴影的处理，在这两个方面进行改进是最值得尝试的。

选定轮廓线之后，我们便可以从光源位置向轮廓线连线，并进行延长，构成一个四边形，这个四边形便是我们构造的 shadow volume 的一部分，当所有四边形都构造完毕之后，由于他们两两相邻，便构成了 shadow volume 的侧面。如下图所示，V0，V1 是我们选定的轮廓线的边，V2，V3 是从光源分别向 V1,V0 引线并向无穷远处延长所得。

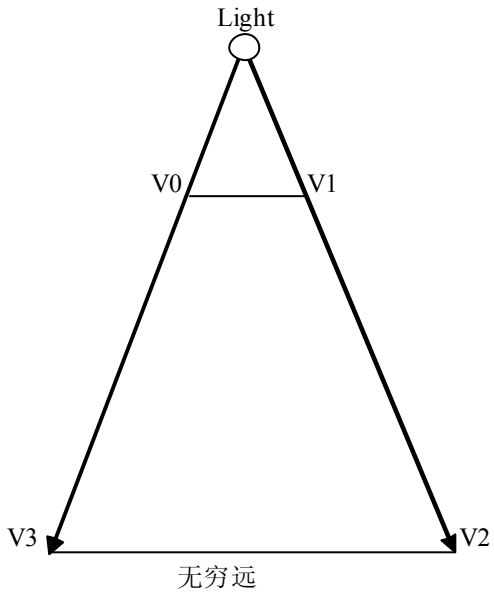


图 10 shadow volume 侧面构造方法示意图

还有一个问题需要解决，在图中我们看到 V2，V3 下面是无穷远处（Infinity），这是一个让计算机非常难办的事情，到底什么是无穷远呢？这里我们要利用齐次坐标来帮助我们解决这个问题，大家可能都知道（不知道的可能需要去翻阅相关三维数学基础的书籍）在图形内部的处理过程中，所有的点都是被当做具有 4 个坐标的三维齐次坐标来处理的，每个列向量（x，y，z，w）^{T}如果至少存在一个非零的元素，那么它就表示一个齐次顶点。只要 w 不等于 0，齐次顶点（x，y，z，w）^{T}就对应于三维点（x/w，y/w，z/w）^{T}；如果 w=0.0，那么它不再对应于任何欧几里德点，而是对应于一些理想化的“无穷远点”。例如（1，0，0，0）点表示沿着 X 轴正方向的一个无穷远点。计算机在处理这个点时由于要经历投影的过程，这个点会被投影到正确的位置。那么这个无穷远点到底该如何表示呢？我们通过下图来讨论。

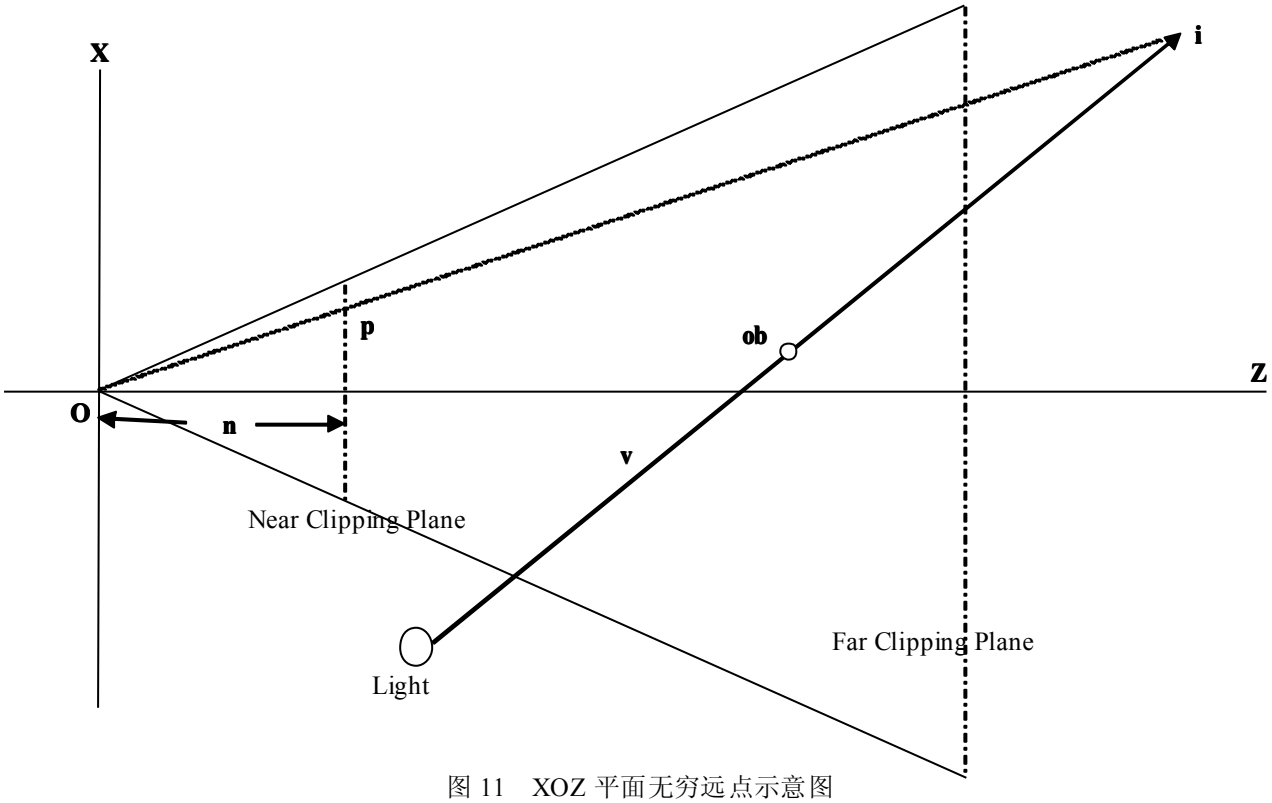


图 11　XOZ 平面无穷远点示意图

如上图所示，遮挡物顶点 P 沿着光线向无穷远处延伸至 **i** 点，**i** 点在近投影面的投影点是 **p** 点，现在我们要计算的是 I 点在 P 的投影坐标（$\mathbf{x_p}$，$\mathbf{y_p}$，$\mathbf{z_p}$），从光源到 **ob** 点的向量是 **v**。有下述公式成立：

$$\frac{x_p}{n}=\frac{x_{ob}+v_x t}{z_{ob}+v_z t}=\frac{\left(\frac{x_{ob}}{t}+v_x\right)t}{\left(\frac{z_{ob}}{t}+v_z\right)t}=\frac{\frac{x_{ob}}{t}+v_x}{\frac{z_{ob}}{t}+v_z}$$

其中 **t** 表示沿着光源向遮挡物延伸的比例，当 **i** 趋向于无穷远时，**t** 趋向于无穷大。所以有下面的公式成立：

$$\frac{x_p}{n}=\lim_{t\to\infty}\frac{\frac{x_p}{t}+v_x}{\frac{z_p}{t}+v_z}=\frac{v_x}{v_z}\Rightarrow x_p=n\frac{v_x}{v_z}$$

同理可得：$y_p = n\frac{v_y}{v_z}$

所以 **p** 点坐标可以表示为：$p(n\frac{v_x}{v_z},n\frac{v_y}{v_z},n,1)$

于是点坐标可以表示为：$i(v_x,v_y,v_z,0)$

v 向量的是通过物体的坐标减去光源的坐标点获取的，这里有一点需要提醒，就是在计算 **v** 向量时我们需要将光源你的坐标移动到绘制物体的局部坐标系中来，因为我们绘制物体和 shadow volume 都是在绘制物体的局部坐标系中来完成的。而为了获得光源相对于局部坐标系的坐标，我们需要获得物体的模视矩阵的逆矩阵然后乘以光源的位置向量来得到局部坐标。

2.4 构造 shadow volume 的覆盖面（cap）

上一步中，我们构造了 shadow volume 的侧面。但是在使用 Depth-Fail 算法时，由于该方法依赖于 shadow volume 背面的测试失败来为模板缓存进行+1 的操作，必须要求这个阴影体是闭合的，因此这个阴影体必须要求前后两个覆盖面（cap）。如果没有覆盖面，Depth-Fail 算法将会产生错误的结果。如下图所示，如果没有前覆盖面（front cap），则左边的视线计算出的结果是 0，而该片段是在阴影中的，stencil value 应该大于 0；同样对于右边的视线，如果没有后覆盖面（back cap），则计算出的 stencil value 也是 0，同样产生了错误。当加上了前后两个覆盖面（图中的粗线位置）之后，从视点的位置来看，两个覆盖面都是反面（back face），因此当视线经过他们的时候会对 stencil value 进行加 1 操作，所以最后的结果是正确的。

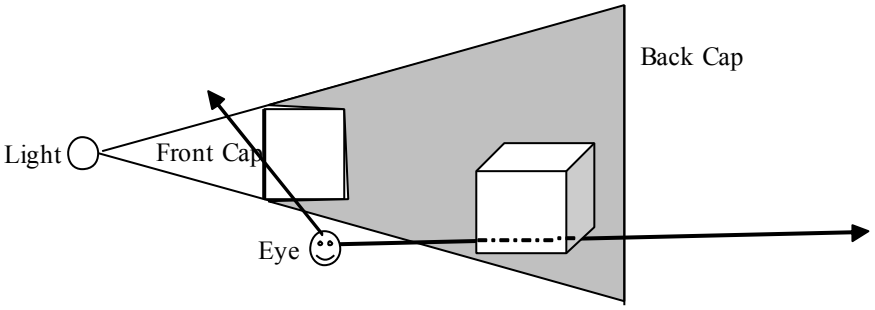


图 9　覆盖面（cap）示意图

但是该如何选择覆盖面呢？对于前覆盖面，Mark Kilgard 给出了一个方法，就是将视点移动到光源的位置，然后让遮挡物在近裁剪面（near clipping plane）上进行投影，得到的便是前覆盖面。我们也可以根据遮挡物几何信息来构造，我们选取遮挡物面对光源的所有图元来构成我们的前覆盖面；然后分别对这些图元沿着光线方向延伸到无穷远处并以反向的顺序进行绘制以构成我们的后覆盖面，按照反向的顺序进行绘制的原因我们前面已经讨论过了，就是要保证 shadow volume 的各个面都是朝向 shadow volume 的外面。

使用 Depth-Pass 算法则没有必要计算前后覆盖面，因为 Depth-Pass 算法不依赖于反面的失败来增加 stencil value。而是用 Depth-Fail 算法构造覆盖面的时候不仅产生了大量的图元（primitive）信息，同时为了计算前后覆盖面需要进行大量的计算。所以在两种算法均可使用的情况下，我们要优先使用 Depth-Pass 算法来实现以节约计算开销。

2.5 存在的问题

1. 延伸 shadow volume 问题

我们在前面讨论过，当使用 Depth-Fail 算法时，需要要给 shadow volume 构造覆盖面，构造覆盖面的目的在于使得我们的 shadow volume 是封闭的，即使他们延伸到无穷远的地方。通常情况下，我们都需要将 shadow volume 得轮廓边延伸到无穷远的地方，因为这样才能保证我们的 shadow volume 对场景中的所有物体生效。当遮挡物非常靠近光源的时候，尤其是如此，如下图所示。当时，如果在你的场景中能够保证这样的情形不会发生，那么也不必向无穷远处延伸。在实际情况中，一个比较大的延长线的值往往就足够了。

但是上述的情况会遇到下面的困难。我们想象以下下面的情景：在一个 FPS（第一人称射击）游戏中，有两个使用砖块隔离的房间，每个房间一个人。整个环境都有环境光，因此每个人都能看到周围的物体，同时左边的房间中中间有一张桌面，桌面上有一个台灯，并且台灯照射的影子正好投射到相邻的墙上。此时从另一个房间中的人在相邻的墙上会看到一个阴影，就是隔壁房间中的人的影子，就像一个“鬼影”一般！这种情况是不符合常理的。但是我们上面讨论的算法是无能为力的。

2. 视景体剪裁的问题

当我们使用 3D 渲染引擎的时候，我们都会有一个个性化的视野，构成这个视野需要两个重要的参数：近裁剪距离（near clipping distance）和远裁剪距离（far clipping distance），这两个参数构成了视景体的两个裁剪面（clipping plane）。所有任何的视野都会有这样的两个裁剪面，而这两个裁剪面对于我们的 shadow volume 算法可谓是“梦魇”。Depth-Pass 算法和 Depth-Fail 算法都会受到裁剪面问题的困扰。当 shadow volume 和视景体的近裁剪面（near clipping plane）发生交集的时候，Depth-Pass 算法便会失效，因为在交集区域，shadow volume 的前向面丢失了，因此 stencil buffer 就没有了加 1 操作；当 shadow volume 和视景体的远裁剪面（near clipping plane）发生交集的时候，Depth-Fail 算法在很多情况下也会失效。下图中，当使用 Depth-Pass 算法时，由于 shadow volume 的 front face 被近裁剪面剪裁，所以无法实现加 1 操作；当使用 Depth-Fail 算法时，由于 shadow volume 得 back face 被远裁剪面剪裁，同样也无法实现加 1 操作。

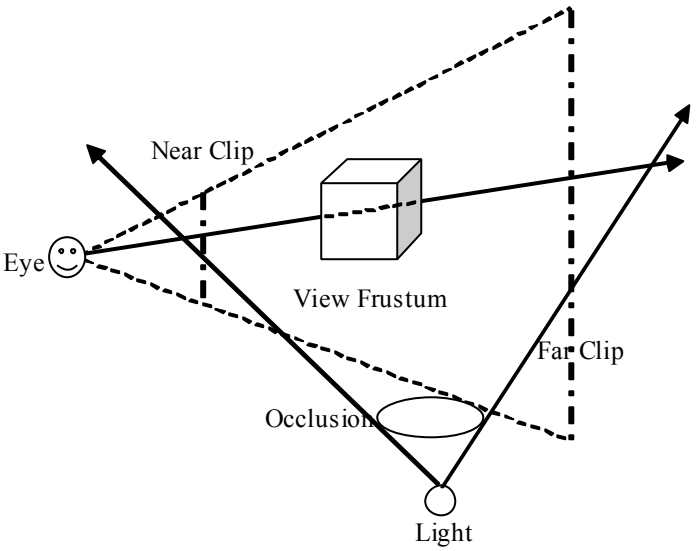


图 11 视景体和 shadow volume 交叉导致算法失败示意图