消除隐藏面算法总结报告

魏少杭 学号：20373594 学院：人工智能研究院

一、消除隐藏面算法综述

1.背景

当使用屏幕显示由计算机图形系统生成的三维场景时，必须要经过某种投影变换才能在二维的显示平面上绘制出来。投影的过程丢失了图形的深度信息，使得生成的图形具有二义性。必须在绘制图形时消除被遮挡的不可见的线面，这个过程叫消除隐藏线或消除隐藏面。经过消隐得到的投影图形就可以称为物体真实的图形。在此基础上，可以得到富有真实感的图形。在使用光栅图形显示器绘制物体的真实图形的时候，必须先消除隐藏面。下面，主要围绕消除隐藏面算法进行讨论。

2.常见的方法

画家算法、区域排序算法、深度缓存（Z-Buffer）算法、扫描线 Z-Buffer 算法、区间扫描线算法、光线投射算法等。

二、画家算法

1.原理

先把屏幕置为背景色，再把物体的各个面按照其距离视点的远近进行排序，形成一个深度优先级表，表头是离视点最远的面（即 z 最小者），表尾是离视点最近的面（即 z 最大者）。然后按照表头到表尾的顺序绘制各个面。由远及近地绘制物体各个面，最终结果就等价于消除隐藏面。这种算法类似于画家作画时先画远景，再画中景，最后画近景类似，故称之为画家算法。

画家算法又称为列表优先算法。该算法**核心在于建立深度排序**。只要建立了各面的排序顺序，就可以根据这个拓扑结构进行一系列渲染。下面是建议的一种构建深度排序的步骤。

2.一种构建深度排序的算法步骤

先根据每一个多边形顶点 z 坐标的极小值z_{min}来把多边形进行初步排序。即设所有多边形的z_{min}中最小者为多边形 P。固定 P 之后，遍历所有其他的多边形，记为 Q。我们要做的工作就是判断是否 P 会遮挡 Q，我们理想的情况就是要 P 不遮挡 Q，那么深度列表中多边形 P 一定排在多边形 Q 之前。P 一定不遮挡 Q 的情况有以下几种：

（1）若$z_{min}(P) < z_{min}(Q)$同时$z_{max}(P) < z_{min}(Q)$，则 P 一定不能遮挡 Q。

（2）若$z_{min}(P) < z_{min}(Q)$但$z_{max}(P) > z_{min}(Q)$，则分为以下几种情况任一种，就可判断 P 不遮挡 Q。

①P 和 Q 在 xy 平面上的投影的包围盒在 x 或 y 方向上不相交。

②P 和 Q 在 xy 平面上的投影不相交。

③在沿 x 轴方向存在某些 x 值使得 PQ 在 xy 平面有投影，且这些 x 值对应的深度 z 值总是保持 z(P)<z(Q)，则 P 不会遮挡 Q。

如果以上（1）（2）测试均不通过，则必须对两个多边形在 xy 平面上的投影作求交运算，在交点处能够比较 PQ 的深度从而得出前后顺序即可。若遇到多边形相交或循环重叠的情况，还必须在相交处分割多边形，然后进行判断。

3.算法局限性

首先，画家算法只能处理互不相交的面；其次，深度优先级表中面的顺序也可能会错误；第三，如果两个面相交，三个以上的面重合情况，无法得到一个精确的深度排序表，只能先将有关的面进行分割后再排序。第四，画家算法的深度排序计算量大，且排序后还需要再检查相邻的面，以保证排序的绝对正确。

三、区域排序算法

1.原理

首先，与画家算法类似，在图像空间中，将待显示的所有多边形按照深度值从小到大排序，用前面的可见多边形切割后面的多边形，最终使得每一个多边形要么是完全可见，要么是完全不可见。这个算法需要用到多边形裁剪算法，能够很好地处理凸多边形、凹多边形以及内部有空洞的多边形。

该裁剪算法要求多边形的边都是有向边，设多边形的外环总是沿着顺时针方向，并且沿着边的走向，边左侧为多边形外部、边右侧为多边形内部。若两个多边形相交，新的多边形可以用“相遇后右拐跳到另一多边形”的规则生成。于是，被裁剪多边形（深度值大者）就被（深度值小的）裁剪多边形分为两个或多个多边形。

2.步骤

（1）进行初步深度排序，例如前述画家算法的初步排序方法：按各个多边形的 z 坐标最小值进行排序。

（2）选择当前离视点最近的多边形作为裁剪多边形。

（3）利用（2）中裁剪多边形，对其他离视点更远的多边形进行裁剪。

（4）比较裁剪多边形和各个内部多边形的深度，检查裁剪多边形是否是离视点最近的多边形。如果裁剪多边形深度大于某一个内部多边形的深度，则需要恢复被裁剪的各个多边形的原有形状，选择新的裁剪多边形，并回到步骤（3）再做裁剪，否则进行步骤（5）。

（5）选择下一个深度最小的多边形作为裁剪多边形，从步骤（3）开始做，直到所有多边形都处理过为止。得到的多边形中，除内部多边形是不可见的，其余所有多边形是可见多边形。

3.性能优化

区域排序算法复杂度与输入多边形的个数密切相关。可以参考分治思想，将输入多边形按照 xy 投影平面进行分区，这样在每一个区里先各自进行消隐，然后再对全局进行消隐。这样做可以降低算法时间复杂度。

四、Z-Buffer 算法

提出该算法的目的：避免画家算法等深度排序计算量大的问题。

1.原理

需要帧缓存存放每一个像素的颜色值、深度缓存来存放每一个像素的深度值。Z 缓冲器中的每一个单元的值是对应像素点所反映对象的 z 坐标值。初值取 z 的极小值；帧缓冲器每一个单元的初值存放对应背景颜色的值。

给帧缓冲器、Z 缓冲器中相应的单元填充值的过程就是消隐过程。把显示对象的每一个面上的每一个点的属性值填入帧缓冲器相应单元之前，需要拿这个点的 z 值与 Z 缓冲器中相应单元的值进行比较，当且仅当 z 小于 Z buffer 对应单元值时，即当前点比 Z buffer 单元值更接近观察点时，才改变帧缓冲器的该单元值，同时更新 Z buffer 的值为当前点的 z 值。当所有点都进行了上述处理后，就得到了消除隐藏面的结果。

2.基本 Z-Buffer 算法步骤

（1）帧缓存全置为背景色，Z buffer 全置为最小 Z 值。

（2）对于每一个多边形，先进行扫描转换。

（3）对于该多边形覆盖的每一个(x,y)像素值，

①首先计算多边形在该像素的深度值 Z(x,y)；

②比较 Z(x,y)与(x,y)在 Z buffer 中的值

③若 Z(x,y)>(x,y)，则更新 Z buffer 在(x,y)的值；多边形在(x,y)处的颜色值存入帧缓存的(x,y)处。

（4）若未遍历完所有多边形，回到（2）；否则，结束。

3.基本 Z-buffer 算法性能分析

本质上，Z buffer 是以空间换时间的算法，在像素级上以近物体出现而隐蔽远物体。根据算法步骤可以看到其空间消耗度非常大，Z buffer 至少要存图像像素数量多个值。为了减小空间开销，可以变换扫描思路，直接对屏幕像素进行扫描，然后判断是背景还是某个最近的多边形的像素值作为显示。

4.改进的 Z-bugger 算法步骤

（1）帧缓存全置为背景色。

（2）对于每一个像素点(x,y)，扫描，并设置当前(x,y)的深度缓存变量 zb 为最小值 MinValue，以供后面更新。

（3）对于多面体上每一个多边形 P_k

①**判断是否 P_k 的投影多边形包含了像素点(x,y)**。若是，则进行下面步骤，否则继续遍历。

②计算 P_k 投影多边形在(x,y)的深度值 depth，将其与 zb 进行比较，若 depth 大于 zb 则更新 zb，并记录最大 zb 对应的多边形的编号 index=k。否则，找下一个多边形。

（4）判断(x,y)像素值对应的 zb 值是否仍然是 MinValue，如果不是，则帧缓冲在(x,y)像素处保存多边形 P_index 值。否则，帧缓冲颜色仍然是背景色。回到第（2）步。

关于（3）①要求进行像素点是否包含在多边形 P_k 中即包含性检测，且要计算多边形 P_k 投影点(x,y)处的深度。

包含性检测可以使用射线法或弧长法，判断一个给定的点是否在一个多边形内。

平面多边形的**深度计算**可以使用平面方程来实现。平面方程：

$$ax + by + cz + d = 0$$

①若$c \neq 0$，则把(x,y)代入方程可以得到$depth = -\frac{ax+by+d}{c}$。

②若$c = 0$，则说明多边形贯穿了 z 轴方向，在 xy 平面上投影为一条直线，在 Z-buffer 消隐算法中不考虑此类多边形。

五、扫描线 Z-Buffer 算法

1.原理

利用扫描线（比如在 xy 平面中遍历 y 值，以 y=c（c 变化）的直线为扫描线）**扫描屏幕平面**，开一个**一维数组作为当前扫描线的 Z-buffer，描述这一条扫描线各个像素值的最大深度。**

首先要找出与当前扫描线相关的多边形，以及每一个多边形中相关的边对；然后计算每一个边对之间的小区间上的各个像素的深度，这些深度与 Z-buffer 中的值比较，来找出各个像素对应的可见平面，并计算颜色，写入帧缓存。

2.数据结构

（1）多边形 Y 表

指针数组。以 Y 坐标值为数组下标，每一个元素是一个指针，这个指针指向若干个多边形组成的链表。建立时，根据每一个已编序号的多边形顶点的最小值 y 对应到多边形 Y 表的数组索引值，并将这个多边形的序号、所有顶点中最大 y 值插入到对应链表中。

多边形 Y 表可以方便查找多边形序号，这个序号作为索引可以找到多边形的信息：如多边形平面方程系数（用于深度计算）、多边形的边、顶点坐标、颜色值（用于帧缓存操作）等。

（2）活化多边形表

扫描时，与当前的扫描线相交的多边形存在活化多边形表 APT 中，这是一个动态的链表。保存的信息如多边形 Y 表中链表结构内的各个结点一样，只需多边形序号、顶点最大 y 值。APT 表代表着当前扫描线所需要扫描的所有多边形。

（3）边表

在活化多边形表中，每一个多边形有边表 ET。边表存放了每一条边的端点中较大的 y 值、增量Δx、y 值较小一端的 x 和 z 坐标。

根据以上保存的信息可以计算得到每一条边的线段方程。

（4）活化边对表

扫描时，同一条扫描线上的同一个多边形中某两条边构成边对。当前多边形与当前扫描线相交的各边对信息保存在活化边对表 AET 中。每一个结点内容包含边对的信息：左侧边与右侧边分别保存①该边与扫面线交点的 x 坐标、左侧边在扫描线加 1 时的 x 坐标增量Δx、边的两个端点的最大 y 值；左侧边与扫描线交点处的多边形深度值z_l；多边形序号 IP；沿着扫描线 x 增加 1 时，多边形平面深度的增量Δz_a；扫描线加 1 时，多边形平面深度的增量Δz_b。

3.步骤

（1）建立多边形 Y 表。

（2）构建活化多边形表 APT，初始化为空表（因为尚未扫描）。

（3）循环：i 从小到大遍历，对于每一条扫描线 i

①帧缓存置为背景色；

②深度缓存（一维数组）置为负无穷大；

③对应扫描线 i 的，且处于多边形 Y 表中的多边形加入到活化多边形表 APT 中。

④对于新加入的多边形，生成器对应的边表 ET。

⑤对于 APT 中每一个多边形，若其边表中对应扫描线 i 增加了新的边，将新的边配对，加到活化边对表 AET 中。

⑥更新缓存：对于活化边对表 AET 中的每一对边：

A.按照增量公式计算各个像素深度值。

B.计算颜色值：先比较是否 depth>深度缓存(x)值，若是，则更新深度缓存(x)值=depth，并计算颜色值写入帧缓存。

⑦删除以后不再扫描的多边形。

⑧更新 AET：

A.删除边对中左右两边满足最大 y 值小于等于扫描线 y 值的边。若一边对中只删除了其中一条边，则需要对该多边形进行重新配对。

B.使用增量公式计算新的边对左右边与扫描线交点的 x 值和左边与扫描线交点的深度值。

4.性能分析

相比基本 Z-buffer 算法，将整个绘图窗口内的消隐问题分解到每一条扫描线上进行。这是利用了不同 y 值之间处理深度信息的无相关性特点。这使得 Z buffer 需要的存储量大幅度减小。

利用了平面的连贯性，使用增量公式计算各个像素点的深度。

由于在步骤（3）⑥更新缓存时，仍然计算了一个区间内所有像素点的深度值，对于很多被多个多边形覆盖的点被进行多次计算，仍然计算量很大。

六、区间扫描算法

1.原理

基于上述的扫描 Z-buffer 算法作出修改。将当前扫描线与各个多边形在投影平面的投影的交点的深度进行排序之后，将扫描线分为若干个子区间，子区间内的颜色就使用该面的颜色来显示，而不必不断多次地扫描区间内的像素值，做无用功。

2.区间着色方法

首先，通过找扫描线与多边形的投影交点得到**若干个子区间**。子区间颜色可以根据三种情况分别**着色**：

（1）若小区间内无多边形，则该区间线段着色为背景色；

（2）若小区间内仅有一个多边形，则该区间线段全部着色为对应的多边形颜色；

（3）若小区间内不少于两个多边形，则该区间内需要进行深度测试判断哪个多边形在前，着色为在前的多边形。此处会出现贯穿情况，需要他们在扫描平面 zx 平面的交点，用这些交点将小区间再细分为若干子区间，然后再决定子区间哪个多边形是可见的。

七、区域子分割算法（Warnack 算法）

1.原理

将物体投影到全屏幕窗口内。**递归分割窗口**，使得窗口内最多包含一个多边形；否则进行左右上下等分成四个子窗口，进行循环；如果循环到某一步窗口达到像素级别，但仍有不少于两个以上的面，则停止分割，取窗口内最近的可见面的颜色或所有可见面的平均颜色作为该像素值。这种方法也被称为四叉树算法。

八、光线投射算法

1.原理

类似改进版的 Z-buffer 算法，但这是**基于物体空间各个面之间的遮挡关系**。先扫描屏幕像素，然后以该像素作为视点发射垂直于屏幕的射线射入场景；找出场景中与该射线相交的物体。如果有交点，则面片颜色为该像素的颜色；否则，说明没有多边形会投影到该像素，像素颜色为背景色。

2.性能分析

由于与改进版的 Z-buffer 流程基本一致，整体算法时间复杂度相似。但不需要 Z-buffer。为了提升光线投射算法的效率，可以使用包围盒技术、空间分割技术等来加速检索物体。

九、总结和感悟

在使用 OpenGL 实现大作业阴影映射的时候，设置 Z buffer 获取深度信息，然后进行深度贴图。计算阴影部分在片段着色器中就是使用了类似于 Z buffer 算法中，将每一个片段调整到（以光源为视角的）屏幕范围内得到对应 x,y 坐标，并计算其深度值，再比较之前预先缓存的离视角最近点的深度值 closestDepth。若当前片段深度值更大，则获得阴影，否则无阴影。

以上的算法主要分为物理空间的消隐、图像空间的消隐以及两种空间融合的消隐。

物理空间的消隐算法基于场景中的实际物体的各个面来判断深度，有光线投影算法等。而图像空间的消隐则从屏幕上的各个像素作为参考进行判断，决定哪些多边形在当前像素可见，如 Z-buffer 算法、扫描算法、区域子分割算法等。而画家算法就是两种空间融合的消隐算法，它在物体空间中预先计算面的可见性优先级，然后在图像空间中生成消隐图。