计算机图形学

第七章 消隐

颜波

复旦大学计算机科学技术学院
byan@fudan.edu.cn

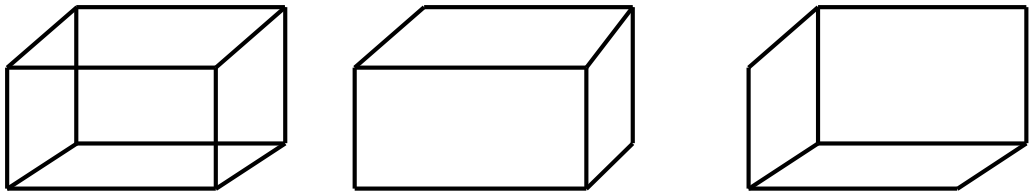
本章概述

消除隐藏面

- 基本概念
- Z缓冲区(Z-Buffer)算法
- 扫描线Z-buffer算法
- 区间扫描线算法
- 光线投射算法

基本概念

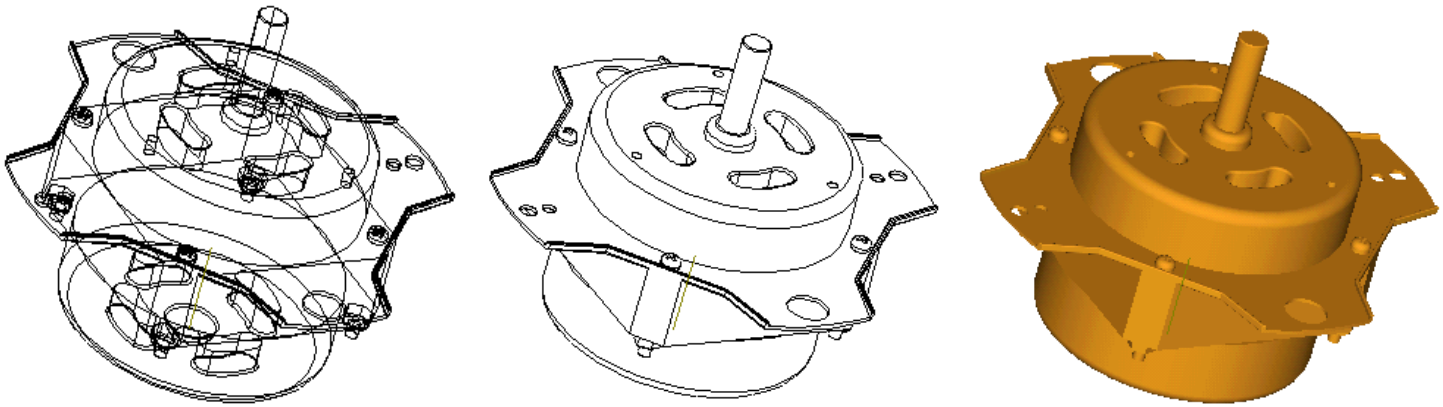
- 投影变换失去了深度信息，往往导致图形的二义性
- 要消除二义性，就必须在绘制时消除被遮挡的不可见的线或面，习惯上称作消除隐藏线和隐藏面，简称为消隐。
- 经过消隐得到的投影图称为物体的真实图形。



长方体线框投影图的二义性

基本概念

- 消隐的对象是三维物体。
- 消隐结果与观察物体有关，也与视点有关。



消隐的分类

- 按消隐对象分类
  - 线消隐：消隐对象是物体上的边，消除的是物体上不可见的边。
  - 面消隐：消隐对象是物体上的面，消除的是物体上不可见的面。

面消隐

面消隐算法的分类：

- Z缓冲器算法
- 扫描线Z缓冲器算法
- 区间扫描线算法
- 光线投射算法

面消隐算法的分类

- 消隐算法的分类
  - 第一类（图像空间的消隐算法）
    - 以窗口内的每个像素为处理单元；
    - 如Z - buffer、扫描线算法

    for (窗口内的每一个像素)
    {
        确定距视点最近的物体，以该物体表面的颜色来显示像素
    }
  - 第二类（物体空间的消隐算法）
    - 以场景中的物体为处理单元；
    - 如光线投射算法

    for (场景中的每一个物体)
    { 将其与场景中的其它物体比较，确定其表面的可见部分；
       显示该物体表面的可见部分；
    }

面消隐算法的分类

- 第一类（图像空间的消隐算法）
  - 以窗口内的每个像素为处理单元；

  for (窗口内的每一个像素)

  {确定距视点最近的物体，以该物体表面的颜色来显示像素}

假设场景中有k个物体，平均每个物体表面由h个多边形构成，显示区域中有m x n个像素，则：

算法的复杂度为：O(mnkh)

面消隐算法的分类

- 第二类（物体空间的消隐算法）
  - 以场景中的物体为处理单元；

for (场景中的每一个物体)
{ 将其与场景中的其它物体比较，确定其表面的可见部分；
显示该物体表面的可见部分；
}

假设场景中有k个物体，平均每个物体表面由h个多边形构成，显示区域中有m x n个像素，则：

算法的复杂度为：O((kh)*(kh))

Z-Buffer算法

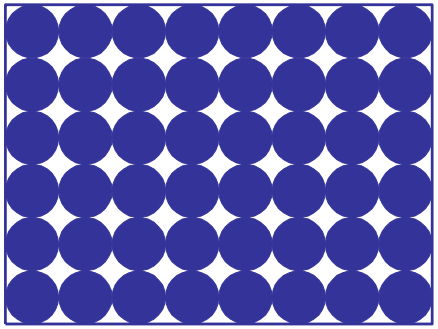
- 由来

  帧缓冲器：保存各像素颜色值

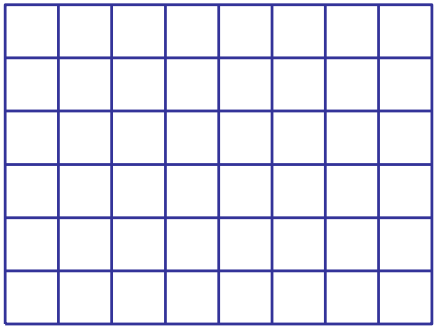
  Z缓冲器：保存各像素处物体深度值

Z缓冲器中的单元与帧缓冲器中的单元一一对应

屏幕

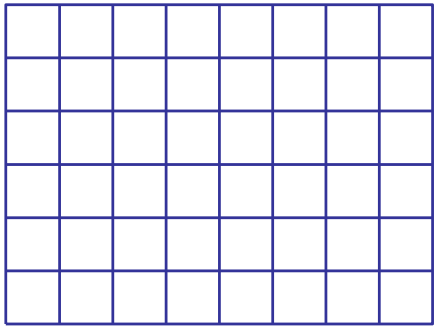


帧缓冲器



每个单元存放对应象素的颜色值

Z缓冲器



每个单元存放对应象素的深度值

Z-Buffer算法

思想：

① 先将Z缓冲器中每个单元的初始值置为最小值。

② 当要改变某个像素的颜色值时，检查当前多边形的深度值是否大于该像素原来的深度值（保存在该像素所对应的Z缓冲器的单元中），

a) 如果大于，说明当前多边形更靠近观察点，用它的颜色替换像素原来的颜色；

b) 否则说明在当前像素处，当前多边形被前面所绘制的多边形遮挡了，是不可见的，像素的颜色值不改变。

Z-Buffer算法-算法描述

```
{   帧缓存全置为背景色
    深度缓存全置为最小Z值
    for(每一个多边形)
    {  for(该多边形所覆盖的每个象素(x,y) )
       {     计算该多边形在该象素的深度值Z(x,y);
             if(Z(x,y)大于Z缓存在(x,y)的值)
             { 把Z(x,y)存入Z缓存中(x,y)处
               把多边形在(x,y)处的颜色值存入帧缓存的(x,y)处
         }
       }
     }
}
```

Z-Buffer算法

- Z缓冲器算法是所有图像空间算法中最简单的一种隐藏面消除算法。
  - 它在象素级上以近物取代远物，与形体在屏幕上的出现顺序无关。
- 优点：

  ①简单稳定，利于硬件实现

  ②不需要整个场景的几何数据
- 缺点:

  ①需要一个额外的Z缓冲器

  ②在每个多边形占据的每个像素处都要计算深度值，计算量大

Z-Buffer算法-改进算法

- 只用一个深度缓存变量zb的改进算法。
  - 一般认为，Z-Buffer算法需要开一个与图象大小相等的缓存数组ZB，
  - 实际上，可以改进算法，只用一个深度缓存变量zb。

Z-Buffer算法-改进算法过程

```
{  帧缓存全置为背景色
   for(屏幕上的每个象素(i,j))
   {  深度缓存变量zb置最小值MinValue
            for(多面体上的每个多边形Pk)
             {      if(象素点(i,j)在pk的投影多边形之内)
                    {       计算Pk在(i,j)处的深度值depth;
                            if(depth大于zb)
                            {       zb = depth;
                                    indexp = k;
                             }
                    }
            }
     if(zb != MinValue)
     在交点 (i,j) 处用多边形Pindexp的颜色显示
    }
}
```

Z-Buffer算法-改进算法

- 关键问题：判断象素点(i,j)是否在多边形Pk的投影多边形之内
- 计算多边形Pk在点（i，j）处的深度。设多边形Pk的平面方程为：

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

$$depth = -\frac{ai + bj + d}{c}$$

扫描线Z-buffer算法

- 由来：Z缓冲器算法中所需要的Z缓冲器容量较大
- 为克服这个缺点可以将整个绘图区域分割成若干个小区域，然后一个区域一个区域地显示，
- 这样Z缓冲器的单元数只要等于一个区域内像素的个数就可以了。
- 如果将小区域取成屏幕上的扫描线，就得到扫描线Z缓冲器算法。

扫描线Z-buffer算法

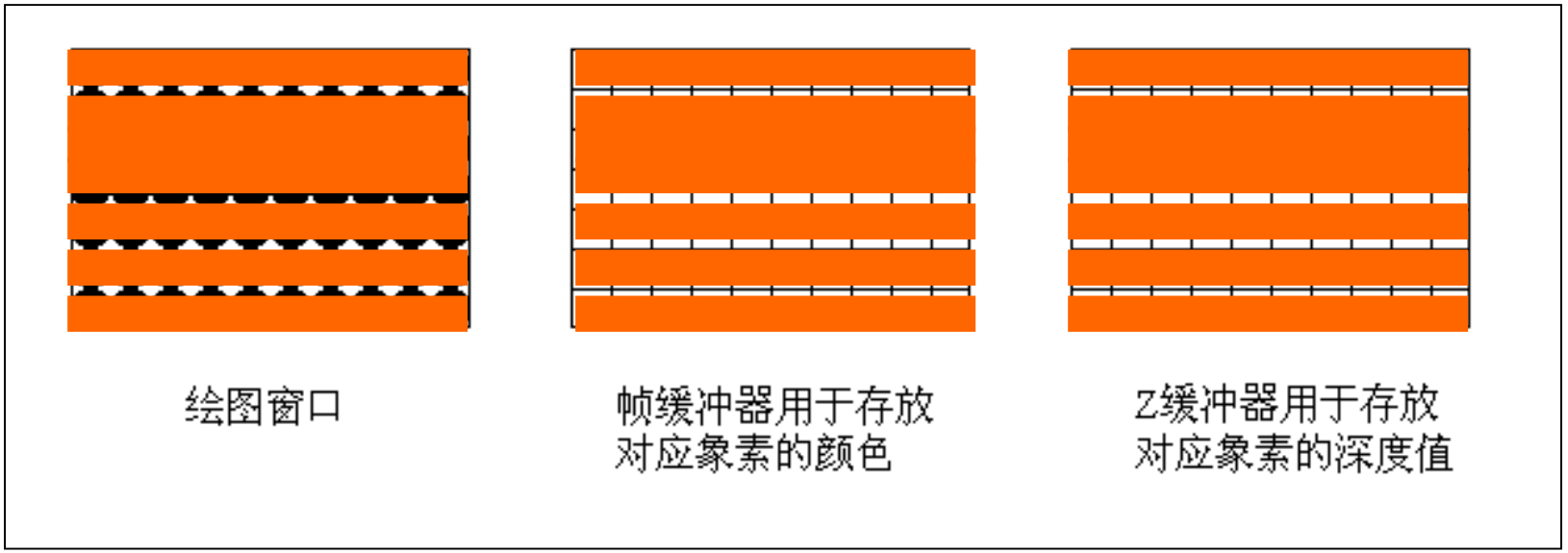
- 算法思想:
  - 在处理当前扫描线时，开一个一维数组作为当前扫描线的Z-buffer。首先**找出与当前扫描线相关的多边形，以及每个多边形中相关的边对。**
  - 对每一个边对之间的小区间上的各象素，计算深度，并与Z-buffer中的值比较，找出各象素处可见平面。
  - 写帧缓存。采用增量算法计算深度。

扫描线Z-buffer算法

```
for ( v= 0;v<vmax;v++)
{    for (u= 0; u<umax; u++)      //初始化当前扫描线
     {  将帧缓冲器的第(u,v)单元置为背景色;
        将Z缓冲器的第u单元置为最小值 }
     for  (每个多边形)
     {  求出多边形在投影平面上的投影与当前扫描线的相交区间
        for   (该区间内的每个像素(u,v) )
        {   计算多边形在该像素处的深度值d;
            if (d > Z缓冲器的第u单元的值)
            {  置帧缓冲器的第(u,v)单元值为当前多边形颜色;
               置Z缓冲器的第u单元值为d;
            }
        }
   }
//处理下一条扫描线
}
```

扫描线Z-buffer算法

- 改进之一：将窗口分割成扫描线



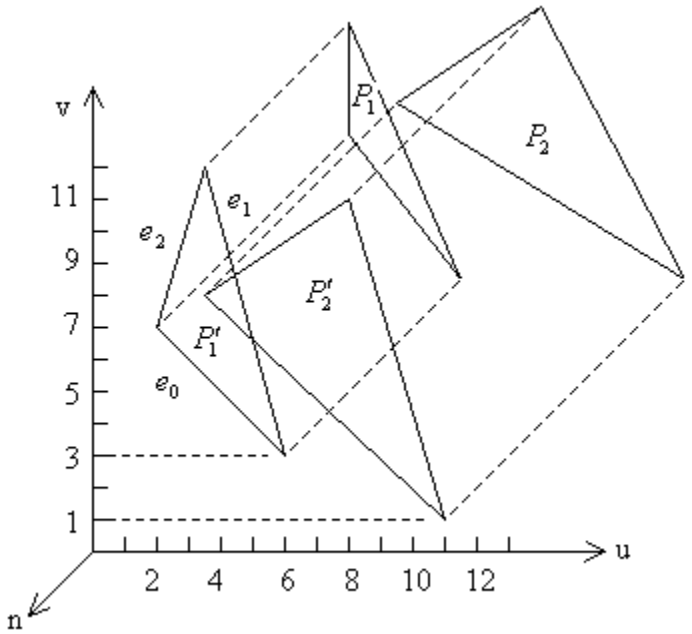
- Z缓冲器的单元数只要等于一条扫描线内像素的个数就可以了。

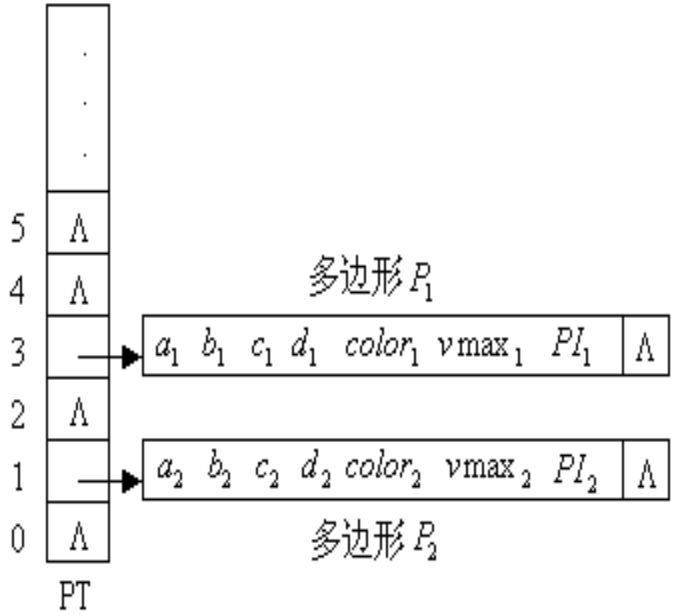
扫描线Z-buffer算法

- 改进之二：采用多边形分类表、活化多边形表避免**多边形**与扫描线的盲目求交

扫描线Z-buffer算法-多边形分类表

- 多边形分类表（PT）：对多边形进行分类的一维数组，长度等于绘图窗口内扫描线的数目。
- **若一个多边形在投影平面上的投影的最小v坐标为v，则它属于第v类。**

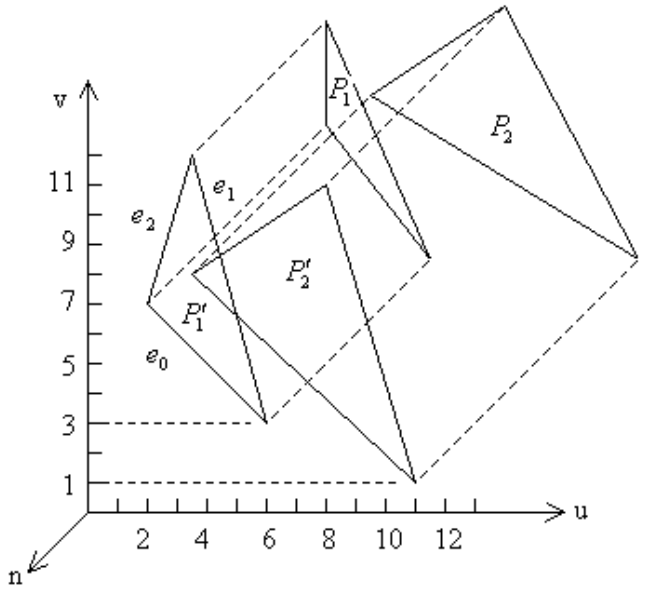


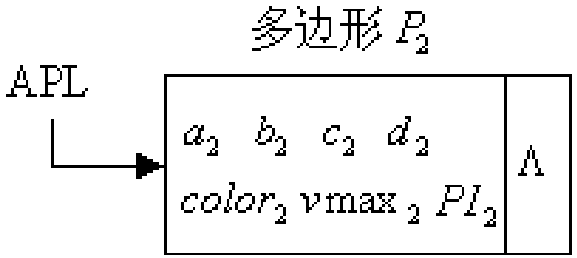


扫描线Z-buffer算法-活化多边形表

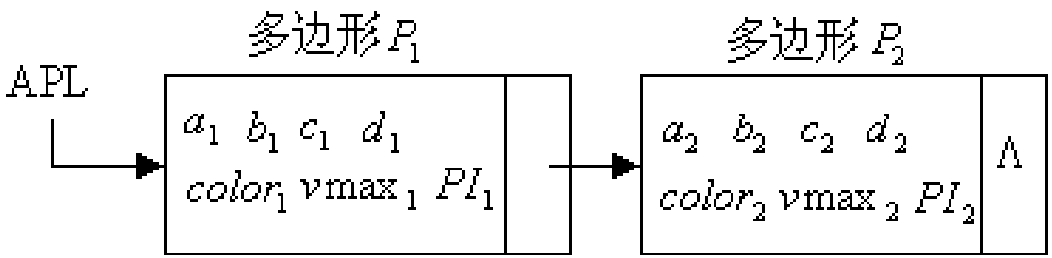
- 活化多边形表（APL）

  记录投影与当前扫描线相交的多边形。





(a) v=2



(b) v=4

扫描线Z-buffer算法-多边形

多边形的数据结构如下：

① a,b,c,d：多边形所在平面方程f(u,v,n)=au+bv+cn+d=0的系数。

② Color：多边形的颜色

③ Vmax：多边形在投影平面上的投影的最大v坐标值。

④ PI：多边形的序号

⑤ nextP：指向下一个多边形结构的指针

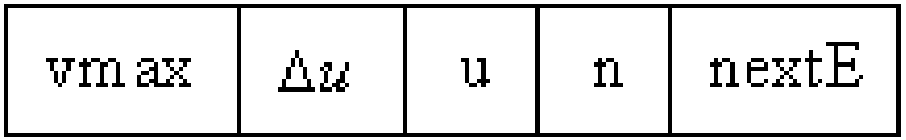
a, b, c, d	color	vmax	PI	nextP

扫描线Z-buffer算法

- 改进之三：利用边、边的分类表、边对、活化边对表避免边与扫描线的盲目求交

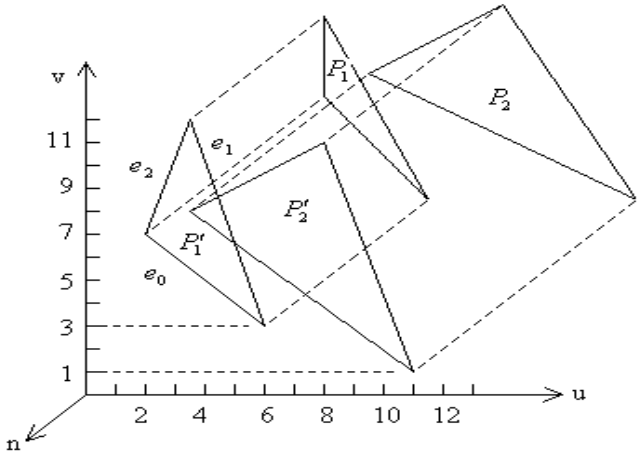
扫描线Z-buffer算法-边数据结构

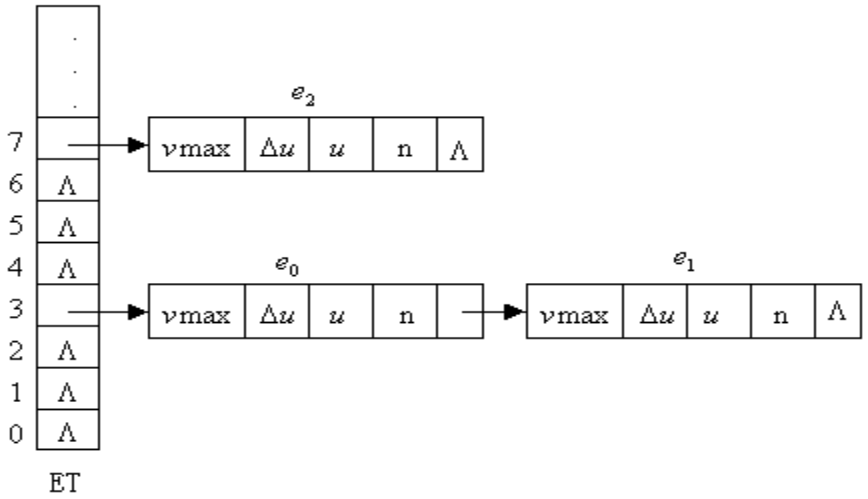
- 边：用来记录多边形的一条边，其中vmax：边的投影的上端点的v坐标。
- u：边的下端点的u坐标
- n：边的下端点的n坐标（深度）
- 增量Δu：在该边上v值增加一个单位时，u坐标的变化量Δ
- nextE：指向下一条边结构的指针。



扫描线Z-buffer算法-边的分类表

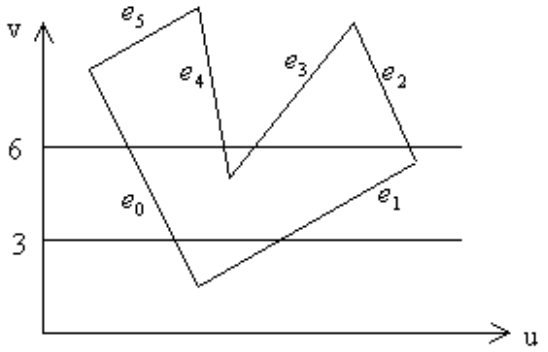
- 边的分类表（ET）：当一个多边形进入活化多边形表时，需为其建立一个边分类表（ET）。
- ET与其在扫描转换多边形的扫描线算法中的含义相同，是对多边形的非水平边进行分类的一维数组，长度等于绘图窗口内扫描线的数目。
- 若一条边在投影平面上的投影的下端点的v坐标为v，则将该边归为第v类。





扫描线Z-buffer算法-边对

边对：在一条扫描线上，同一多边形的相邻两条边构成一个边对。边对中包含了如下信息：



① PI： 多边形的序号

② u_l： 边对的左侧边与扫描交点的u坐标

③ v_{lmax}： 左侧边投影的上端点的v坐标。

④ v_{rmax}： 右侧边投影的上端点的v坐标。

⑤ Δu_l： 当沿左侧边v坐标递增一个像素时，u坐标增量

⑥ u_r： 边对的右侧边与扫描交点的u坐标

⑦ Δu_r： 当沿右侧边v坐标递增一个像素时， u坐标的增量

⑧ n_l： 左侧边与扫描线交点的n坐标

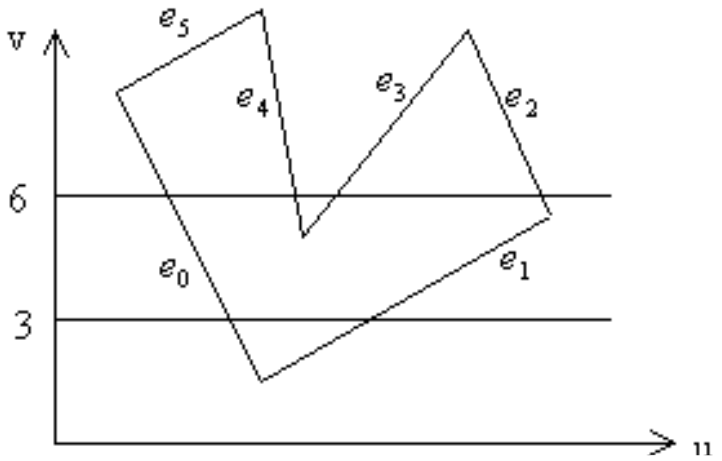
⑨ Δn_u： 当沿扫描线u递增一个像素时，多边形所在平面n坐标的增量，对方程au+bv+cn+d=0来说，$\Delta n_u = -a/c$

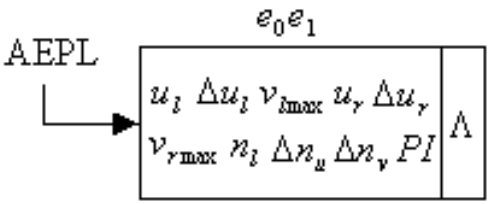
⑩ Δn_v： 当沿扫描线v递增一个像素时，多边形所在平面n坐标的增量，类似，$\Delta n_v = -b/c$ (c!=0)

⑪ nextEP：指向下一个边对结构的指针。

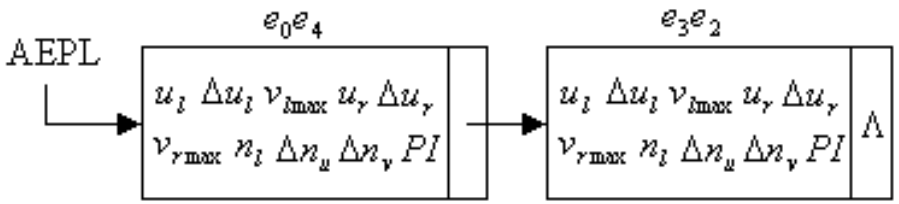
扫描线Z-buffer算法-活化边表对

- 活化边表对（AEPL）：记录了活化多边形表中与当前扫描线相交的边对，边对在AEPL中的顺序无关紧要。





(a) v=3

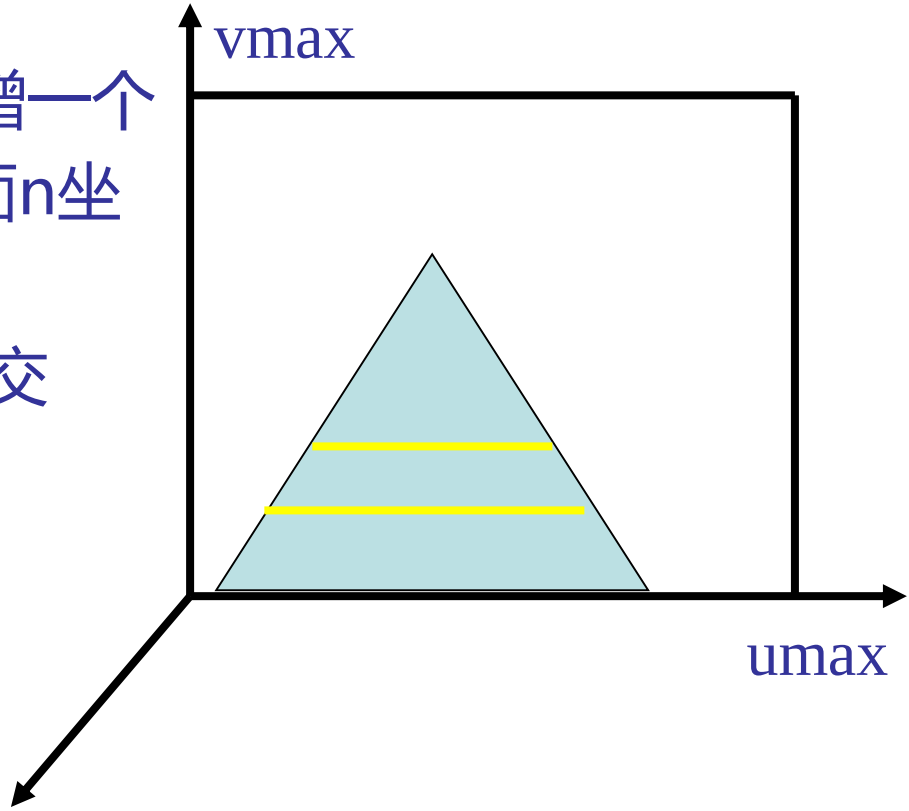


(b) v=6

扫描线Z-buffer算法

- 改进之四：利用连贯性计算深度
  - 水平方向：沿扫描线u递增一个像素时，多边形所在平面n坐标的增量，对方程au+bv+cn+d=0来说，$\Delta n_u = -a/c$
  - 竖直方向：沿扫描线v递增一个像素时，多边形所在平面n坐标的增量，$\Delta n_v = -b/c$
- 下一条扫描线与边对左侧边交点处的深度值：

nl=nl+Δnu Δul+Δnv



扫描线Z-buffer算法

1. 建多边形分类表：对每一个多边形，若它在投影平面上的投影的最小v坐标为v，则它属于第v类。
2. 置活化多边形表APL为空，置活化边对表AEPL为空。
3. 对每条扫描线v，执行下列步骤：
   ① 置帧缓存器第v行中的各单元为背景色。
   ② 置Z缓冲器各单元的值为最小的深度值。
   ③ 检查PT(多边形分类表)的第v类是否非空，如果非空，将该类中的多边形取出加入APL(活化多边形表)中。
   ④ 对新加入APL(活化多边形表)中的多边形，为其建立边的分类表ET。
   ⑤ 对新加入APL (活化多边形表)中的多边形，若它的ET（边的分类表）中的第v类非空，将其中的边配对插入AEPL（活化边对表）中；

扫描线Z-buffer算法

⑥ 对AEPL（活化边对表）中的每一个边对，执行下列步骤：

```
深度值n=nl;
for  (u=ul;u<=ur;u=u+1 )
{
  if (n > Z缓冲器的第u单元的值)
  {    置帧缓冲器的第(u,v)单元值为当前多边形颜色;
       置Z缓冲器的第u单元值为n;
   }
   n=n+ Δnu;//计算下一个像素(u+1,v)处多边形的深度值
}
```

扫描线Z-buffer算法

⑦ 检查APL（活化多边形表），删除那些满足vmax=v的多边形，释放多边形的ET，并从AEPL（活化边对表）中删除属于该多边形的边对。

⑧ 检查AEPL（活化边对表）中的每一个边对，执行下列步骤

a) 若vlmax=v或vrmax=v，删除边对中的左侧边或右侧边。

b) 若左侧边和右侧边都从边对中删除了，则从AEPL（活化边对表）中删去该边对；若边对中仅有一条边被删去了，则从该边对所属的多边形的ET中找到另一条边与余下的边配对，组成新的边对，加入AEPL（活化边对表）；

c) 计算下一条扫描线与边对两边交点的u坐标：

d) ul=ul+Δul；ur=ur+Δur

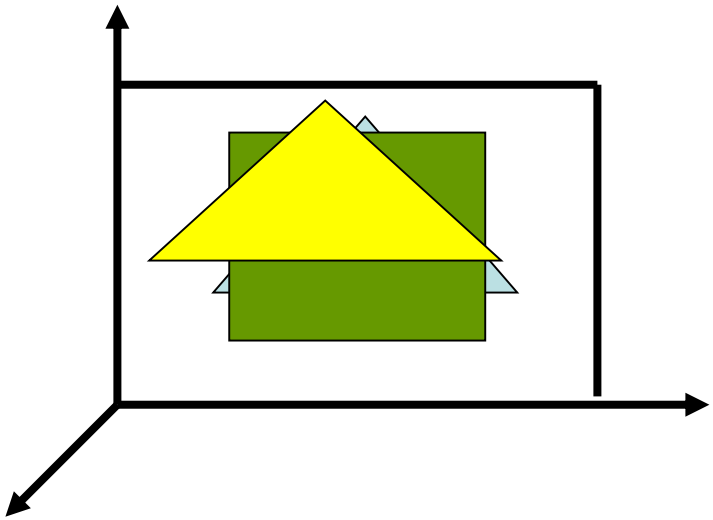
e) 计算下一条扫描线与边对左侧边交点处的深度值：

f) nl=nl+ΔnuΔul+Δnv

⑨ 将扫描线递增一个像素，v=v+1

扫描线Z-buffer算法

- 缺点
  - 在每一个被多边形覆盖像素处需要计算深度值
  - 被多个多边形覆盖的像素需要多次计算深度值

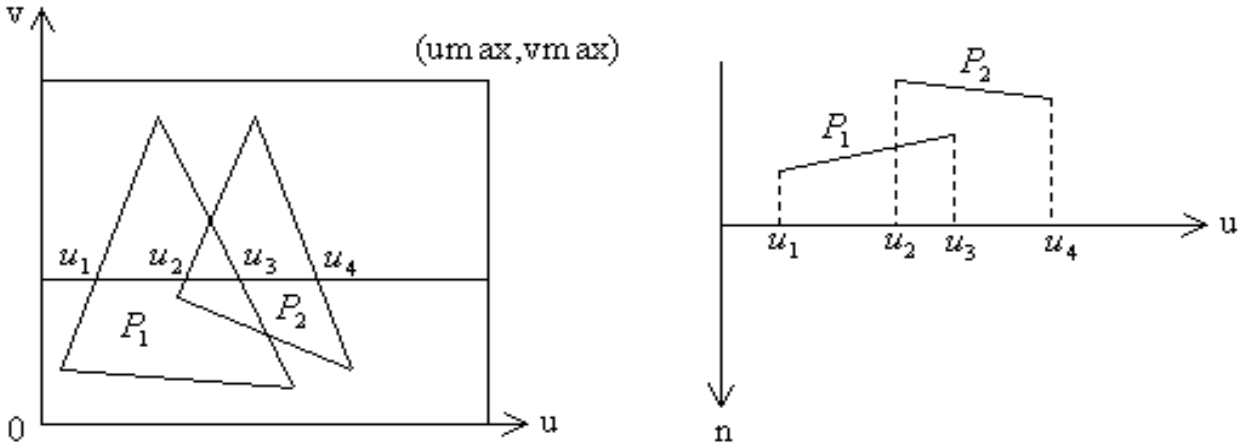


区间扫描线算法

- 与Z-Buffer算法相比，扫描线算法有了很大改进，比如所需的Z-Buffer大大减小，计算深度利用了面连贯性等；
- 缺点：每个像素处都计算深度值，甚至不止一次的计算，运算量仍然很大。
- 改进：在一条扫描线上，每个区间只计算一次深度，即区间扫描线算法，又称扫描线算法。

区间扫描线算法

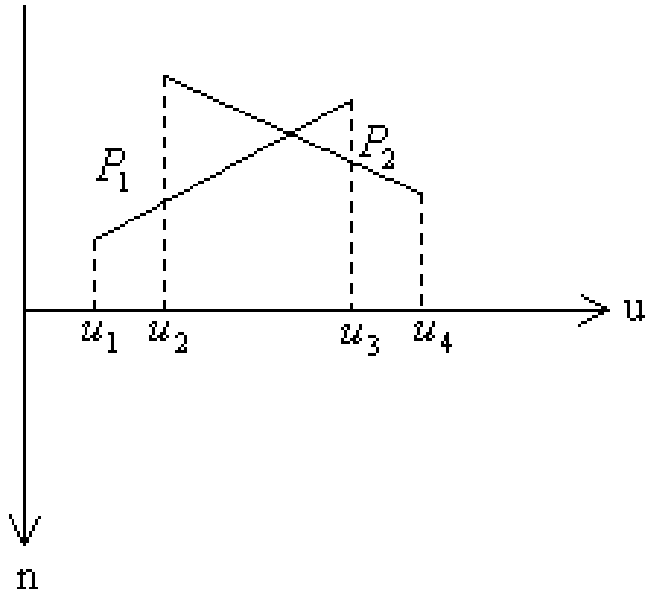
- 基本思想：如下图，多边形P1、P2的边界在投影平面上的投影将一条扫描线划分成若干个区间[0,u1][u1,u2][u2,u3][u3,u4],[u4,umax]，覆盖每个区间的有0个、1个或多个多边形，但仅有一个可见。
- 在区间上任取一个像素，计算该像素处各多边形（投影包含了该像素的多边形）的深度值，深度值最大者即为可见多边形，用它的颜色显示整个区间。



区间扫描线算法

注意：该算法要求多边形不能相互贯穿，否则在同一区间上，多边形深度值的次序会发生变化。

如图：在区间[u1,u2]上，多边形P1的深度值大，在区间[u3,u4]上，多边形P2的深度值大，而在区间[u2,u3]上，两个多边形的深度值次序发生交替。



区间扫描线算法

- 数据结构
  - 多边形分类表
  - 活化多边形表
  - 边的分类表
  - 活化边表
- 类似于扫描线Z-Buffer算法中的数据结构

区间扫描线算法-算法描述

for (绘图窗口内的每一条扫描线)
{ 求投影与当前扫描线相交的所有多边形
求上述多边形中投影与当前扫描线相交的所有边，将它们记录在活化边表AEL中
求AEL中每条边的投影与扫描线的交点；
按交点的u坐标将AEL中各边从左到右排序，两两配对组成一个区间；
for （AEL中每个区间）
{
求覆盖该区间的所有多边形，将它们记入活化多边形表APL中；
在区间上任取一点，计算APL中各多边形在该点的深度值，记深度最大者为P；
用多边形P的颜色填充该区间
}
}

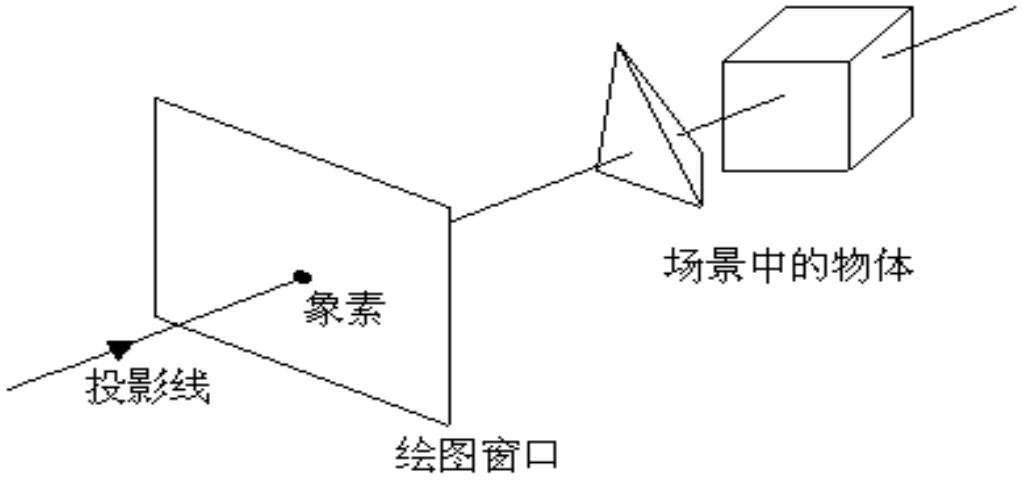
区间扫描线算法

- 相比较扫描线Z-Buffer算法而言，区间扫描线算法做了如下改进：
  - 在一条扫描线上，以区间为单位确定多边形的可见性
  - 不再需要Z-Buffer

光线投射算法

算法思想：

- 将通过绘图窗口内每一个像素的投影线与场景中的所有多边形求交。
- 如果有交点，用深度值最大的交点（最近的）所属的多边形的颜色显示相应的像素；
- 如果没有交点，说明没有多边形的投影覆盖此像素，用背景色显示即可。

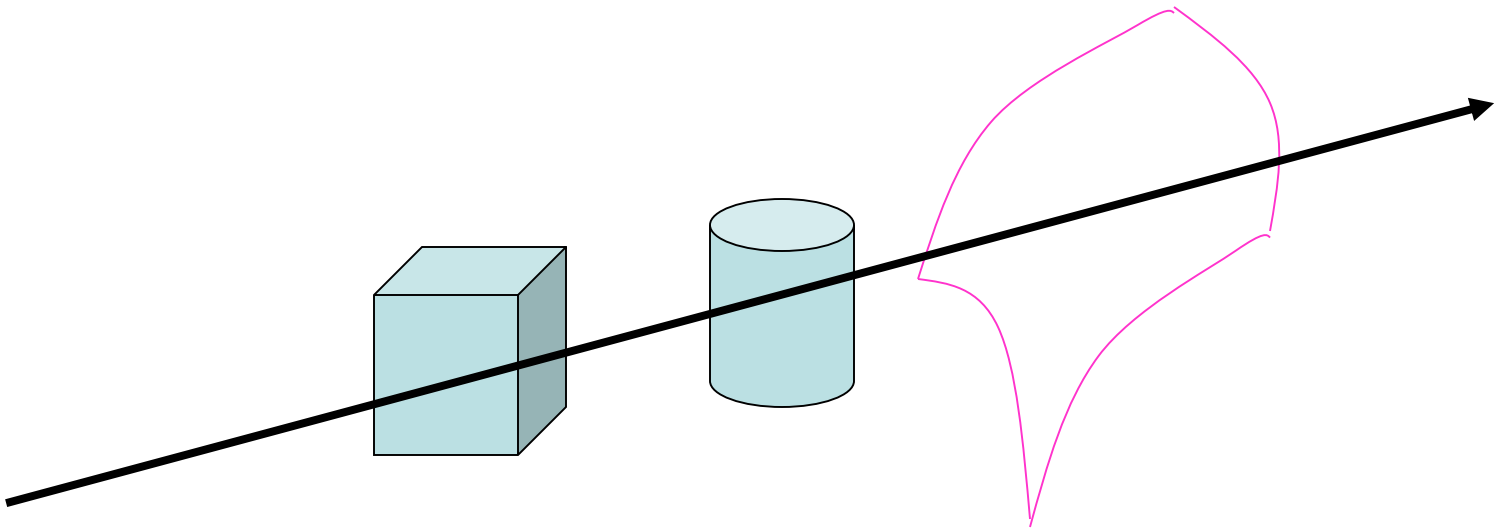


光线投射算法-算法描述

```
for ( v= 0;v<vmax;v++)
    for (u= 0; u<umax; u++)
  {  形成通过像素(u,v)的投影线;
      for (场景中每一个多边形)
          将投影线与多边形求交;
       if  (有交点)
          以最近交点所属多边形的颜色显示像素(u,v)
       else
          以背景色显示像素(u,v);
     }
```

光线投射算法

- 基本问题
  - 光线与物体表面的求交



消隐的考虑

- 选择不同的消隐算法
  - 消隐问题有不同的算法，有些算法要求速度快，有些要求图形的真实度高。
  - 例如，快速消隐算法可用于实时模拟如飞行模拟等；

- 具有高度真实感图形的消隐算法可用于计算机动画等领域，
  - 所生成的图形一般具有连续色调，并能产生阴影、透明、表面纹理及反射、折射等视觉效果。
  - 不过这类算法比较慢。产生一幅图可能需要几分钟甚至几小时。

- 所以，在进行消隐算法的设计时，应在计算速度和图形细节之间进行权衡，任何一种算法都不能兼顾两者。

消隐的考虑

- 消隐算法的实现空间：消隐算法可以在物体空间或图像空间中实现。
  - 物体空间算法是在定义**物体的坐标系**中实现的，而图像空间算法是在对象显示的**屏幕坐标系**中实现的。
  - 物体空间算法以尽可能高的精度完成几何计算，所以可以把图像放大许多倍而**不致损害其准确性**，但是图像空间算法只能以与显示屏的分辨率相适应的精度来完成计算，所以其图像的**放大效果较差**。
  - 这两类算法的性能特性也是不同的。物体空间算法所需的计算时间**随场量中物体的个数而增加**，而图像空间的计算时间则**随图像中可见部分的复杂程度而增加。**