《计算几何》课程实验——

利用 Incremental 算法
求取三维凸包

计研 9

刘喜明　**994952**

许云杰　**994951**

曹　祥　**994960**

2000.01.19

【算法思想】

Incremental 算法的基本思想比较容易理解，就是先求取少数点的凸包再逐步地加入新的点，最后求出整个输入点集的凸包。该算法既适用于二维也适用于三维，只不过在算法具体实现上有所不同。

这里对本实验所采用的算法做一个简单的描述：首先在输入点集 P 中选取四个不在同一平面上的三维点，构造四面体作为初始凸包 H；此后每加入一个点 p，依次判断 H 上的每个三角面片对于 p 来说可见与否，如果都不可见，则说明 p 位于 H 内部，不予处理；如果有可见的面片，则先找出可见面片与不可见面片交界的边组成的集合 E，从 H 中删去可见面片，以 E 和 p 构造锥形面，生成若干新的三角面片，添加到 H 中去。

【数据结构】

一、 点、边、面数据结构

```
typedef struct _vertex VERTEX;          //点
typedef VERTEX * P_VERTEX;              //点指针
typedef struct _edge EDGE;              //边
typedef EDGE * P_EDGE;                  //边指针
typedef struct _plane PLANE;            //三角面片
typedef PLANE * P_PLANE;                //面片指针
struct _vertex
{
    double   x, y, z;                   //点的三维坐标值
    BOOL   m_bProcessed;                //是否已被处理过
    P_EDGE     m_Created;               //已创建新边（用于生成锥形面）
};
struct _edge
{
    P_VERTEX  start, end;               //起点和终点
    P_PLANE    left, right;             //相邻面片
    BOOL       m_bDeleted;              //是否将被删除（相邻面片皆为可见时）
};
struct _plane
{
```

```
    P_VERTEX  v[3];          //三个顶点，逆时针方向（按右手螺旋法则）
    P_EDGE    e[3];          //三条边，逆时针方向（按右手螺旋法则）
    BOOL      m_bVisible;    //可见与否
};
```

其中边的数据结构中相邻面片 left，right 的具体确定将遵循右手法则，即让右手掌的方向和边一样（起点指向终点），让掌心朝向凸包的外部，则右拇指所指的方向即为面片 right 的位置所在，而相反的方向为 left 的位置。

二、 点集、边集、面集数据结构

利用动态数组 Carray 实现：

```
typedef CArray<P_VERTEX, P_VERTEX&> VertexArray;      //点集
typedef CArray<P_EDGE, P_EDGE&> EdgeArray;            //边集
typedef CArray<P_PLANE, P_PLANE&> PlaneArray;         面集
```

三、 凸包数据结构

凸包由点表、边表和面表组成：

```
typedef struct _hull3d
{
    VertexArray     vertices;     //点表
    EdgeArray       edges;        //边表
    PlaneArray      planes;       //面表
} HULL3D;
```

【算法实现】

一、空间四点的有向体积

算法中需要判断某一个三角面片相对于某个点的可见性，这里利用空间计算四面体体积的行列式法得到一个有正负两种可能性的有向体积，并借此来判断面片对于点的可见性，计算公式如下：

设$dx_i = v_i \to x - v_3 \to x,$

$dy_i = v_i \to y - v_3 \to y,$

$dz_i = v_i \to z - v_3 \to z \ldots\ldots\ldots\ldots\ldots\ldots i = 0,1,2$

$V(v_0, v_1, v_2, v_3)$为四面体的有向体积,则

$$Volume(v_0,v_1,v_2,v_3)=6*V(v_0,v_1,v_2,v_3)=\begin{vmatrix} dx_0 & dy_0 & dz_0 \\ dx_1 & dy_1 & dz_1 \\ dx_2 & dy_2 & dz_2 \end{vmatrix}$$

$$=dx_0(dy_1*dz_2-dz_1*dy_2)-dy_0(dx_1*dz_2-dz_1*dx_2)+dz_0(dx_1*dy_2-dy_1*dx_2)$$

其中$v_0 \sim v_3$的先后次序直接影响到结果的正负，算法中一般令$v_0 \sim v_2$分别为某面片的三个顶点（仍按照右手螺旋逆时针方向）。

二、 锥形面的生成

算法中需要用到生成锥形体的生成算法，输入为空间若干条有向边（无序）、一个孤立点和一个现有凸包（可见面片尚未被删去），输出为新的凸包（包含锥形面中的三角面片），算法简单描述如下：

```
MakeConePlane( EdgeArray & edges,   P_VERTEX v,   HULL3D & hull )
{
    将孤立点添加到凸包中;
    for ( edges 中的每一条边 edge )
    {
        if ( edge->start->m_Created 已被创建 )
            直接获得已创建的边指针 edge1
        else
        {
            创建新边 edge1
            edge->start->m_Created = edge1;
            添加 edge1 到凸包中
        }
        if ( edge->end->m_Created 已被创建 )
            直接获得已创建的边指针 edge2
        else
        {
            创建新边 edge2
            edge->end->m_Created = edge2;
            添加 edge2 到凸包中
        }
    if ( edge->left 对于点 v 可见 )
```

以点表（edge->start, edge->end, v ）、边表(edge, edge2, edge1)创建面片 face;

else

以点表（edge->end, edge->start, v ）、边表(edge, edge1, edge2)创建面片 face;

添加面片 face 到凸包中;

}

三、 主算法

主算法可以大致描述如下：

```
CreateHull ( HULL3D & hull,   VertexArray & vertices )
{
    找到前三个不在一条直线的点 v0, v1, v2；
    寻找下一个点 v3 使得 Volume(v0, v1, v2, v3)  不等于 0；
    if ( Volume(v0, v1, v2, v3) > 0 )
        以点表（v0, v1, v2 ）构造边表 (e0, e1, e2 ) 并生成面片 face；
    else
        以点表（v0, v2, v1 ）构造边表 (e0, e2, e1 ) 并生成面片 face；
    添加 v0, v1, v2, e0, e1, e2, face 到凸包中;
    MakeConePlane( (e0, e1, e2), v3, hull );
    for (  每一个未被处理的点 v )
    {
        for (  凸包 hull 中的每一个面片 face )
            调用 Volume()函数判断 face 对于 v 的可见性；
        for(  凸包 hull 中的每一条边 edge )
            if ( edge->left 和 edge->right 对于 v 的可见性相反 )
                将 edge 加入到边集 edges 中；
            else if ( edge->left 和 edge->right 对于 v 均可见 )
                标志 edge 为“将被删除”;
        MakeConePlane( edges, v, hull );
        删去相对于 v 可见的面片；
    }
    删去凸包 hull 中标志为“将被删除”的边;
}
```

【算法复杂度分析】

算法复杂度只要计算在已有四面体凸包基础上不断添加点的算法复杂度即可，首先不难利用欧拉公式和“所有面皆为三角形”的条件证明，本算法中的凸包中点数 V、边数 E、面数 F 存在关系：F=2V-4, 2E=3V。

一、**时间复杂度**

容易看到，在算法 CreateHull()中的最外层 for 循环中，第一个循环判断可见性复杂度为 O(F)；第二个循环寻找关键边复杂度为 O(E)；最后构造锥形面片复杂度也为 O(E)；加起来复杂度为 O(V)。所以总的复杂度应该为 O(N*H)，其中 N 为输入点集的点个数，H 为最后生成凸包上的点个数。

容易得出，H 最小为 O(4)，最大为 O(N)，所以本算法的复杂度在最坏的情况下仍为 O(N*N)，而最好可能达到 O(N*4)=O(N)；

二、 **空间复杂度**

这个从前面的算法中不难得到：判断可见性复杂度为 O(3*F)；构造锥形面片复杂度为 O(2*E/2)；共为 O(V)。所以加上外层循环仍为 O(N*H)，和时间复杂度一样。

【用户界面】

整个实验演示系统采用 Windows 图形界面，三维点集主要采用文件输入，另外也支持临时添加新的点，还支持对五种正多面体求取凸包。

程序运行方式可以选择快速执行和单步执行：前者直接显示最后的凸包结果；后者将跟踪求取凸包的整个过程，先构造最初的四面体，然后每执行一步新加一个点，直到所有点都被处理过。

输入点集和输出凸包的显示采用 OpenGL 编程技术，支持的功能主要有：可以用鼠标抓取凸包（点集）做旋转；可以选择前景和背景颜色；凸包显示可以选择“显示框架”和“渲染面片”（光照模型）；可以放缩图形；

整个系统的用户界面如下页图所示。

【小结与展望】

通过本次实验,对 incremental 算法有了清楚认识,通过编程实践,对点,线,面的及其相互的联系的表示在程序上的实现也有了深刻的认识,更重要的是,通过编程,对 incremental 算法有了从理论联系实际的认识.

本系统编程过程中曾经对原有算法做了一点改进，主要是对起始凸包的构造做改进，基本思想是利用正多面体与对点的“包容性”，假设以正多面体外接球的球心为原点，那么正多面体的每一个顶点将对应一个矢量，如果在求取凸包之前，先遍历输入三维点集中的所有点，找到每个矢量上的最远点，然后以这些最远点构筑初始凸包，这样以来，可以想象输入点集将有大部分的点落入凸包内部，这样可以使得算法的复杂度在一定程度上得到改进。

本系统尝试过在正八面体的思想上实现上述改进，结果在大量随机点的例子中有明显改观，但如果存在若干点同为某一方向矢量（对应于正八面体的某个顶点）上的最原点，则程序有可能会出错。