Quake3 场景管理技术研究报告

作者：cywater2000
来自：http://blog.csdn.net/cywater2000
http://blog.gameres.com/show.asp?BlogID=1976&column=0

版本信息：

v1.00	2005-12-31	**完成基本框架**
v1.02	2006-01-04	**修改**
v1.05	2006-01-06	**增加了第四部分**
v1.06	2006-01-11	**大概完成**
v1.07	2006-02-24	**全部完成**
v1.07b	2006-03-28	**Release 版**

一．前言：

经过一个月左右(2005.11.21-12.28)对 Quake3 源代码的研究，本人终于对其相关的场景管理技术有了一定的认识与心得，于是决定写成报告作为总结，同时也作为将来的回顾资料。

本来当初并没有打算发布出来，而只是自己看，所以写得比较随意，语句都没怎么组织，图也比较简陋，还请见谅。同时希望这份报告对你有用。

在读这份报告前，读者应该具备以下几项前提条件：
(1) 学过计算机图形学，空间解析几何和线性代数
(2) 使用 D3D 或 OpenGL 写过 3D 图形程序
(3) 对室内场景管理有一定了解（BSP,Portal,PVS...）
(4) 知道 ID，JC 和 Quake(^_^)

说明：我研究的是 Q3 的场景管理技术，所以只把重点放在了 q3map 这个程序的源代码上，而且我也认为 q3map 是 Q3 的核心技术之一，花一个月的时间细心研究是值得的。在这里感谢 John Carmack 及相关的 ID 程序员们，谢谢你们能把 Q3 开源了。

二．技术规范：

1．总览

Q3 是用 C 写的(图形 API 是 OpenGL)，第一次接触的时候相当郁闷：全局变量满天飞，指针操作处处有...而且注释太少。另外程序大量使用了各种技巧，有些简直是“莫名其妙”...(比如传说中的那个 sqrt 常数 = =!)

Q3 的场景管理用的是 BSP+Portal+PVS。注意这里的 Portal 技术只是帮助生成 PVS，之后就不再使用了。

Q3 使用右手笛卡尔坐标系：即 Z 轴垂直向上，Y 指向屏幕里。

2．规范及风格

源代码中带_r 后缀的函数是递归函数，带_t 后缀的类型为 typedef 类型(通常为结构体)，带 c_前缀的变量为计数器。

Q3 将所有的平面都保存在了一个平面表中：plane_t mapplanes[MAX_MAP_PLANES]以方便查询，并且对于每一个新生成(即原表中没有的)的平面，若索引为 index，则其反平面(normal 和 d 的符号取反，即背面)的索引为 index+1。这样做的好处是对于任意平面，如果其索引为 i 则其反平面为 i^1。

在 Q3 中，要渲染的场景多边形面(dsurface_t，mapDrawSurface_t)一共有四种类型：
(1)由 brush side 产生的普通面(MST_PLANAR)，所有顶点共面
(2)曲面(MST_PATCH)，由二次 Bezier 曲面构成，保存着控制点(至少 9 个)

$$Q(u,v) = \sum_{i=0}^{2}\sum_{j=0}^{2} P_{i,j} B_i(u) B_j(v)$$

(3)由 md3 mesh model 产生的三角形 list 面(MST_TRIANGLE_SOUP)，构成一个完整的 mesh
(4)Billborads\Sprite(MST_FLARE)，通常是中心点或者 4 个边界点。

Q3 中使用了一种称为 shader 脚本的技术。要注意这里的 shader 并不是指现在流行的着色器，而是一种描述平面特性的脚本：Shaders are short text scripts that define the properties of a surface as it appears and functions in a game world (or compatible editing tool). By convention, the documents that contain these scripts usually has the same name as the texture set which contains the textures being modified (e.g; base, hell, castle, etc,). Several specific script documents have also been created to handle special cases, like liquids, sky and special effects.

例如：

A *Quake III Arena* shader file consists of a series of surface attribute and rendering instructions formatted within braces (“{“ and “}”). Below you can see a simple example of syntax and format for a single process, including the Q3MAP keywords or “Surface Parameters”, which follow the first bracket and a single bracketed “stage”:

```
textures/liquids/lava
{
    deformVertexes wave sin 0 3 0 0.1
    tessSize 64
    {
    map textures/common/lava.tga
    }
}
```

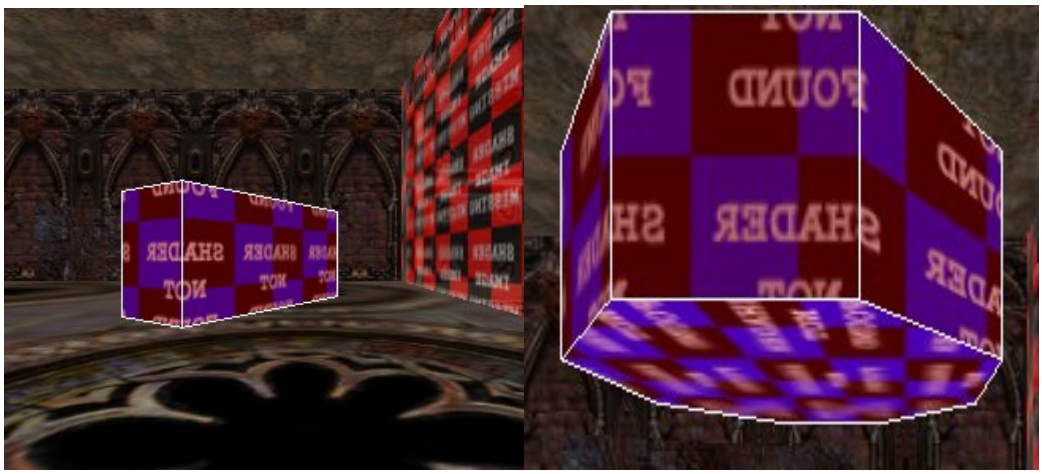
具体说明与用法请参考《Quake III Arena Shader Manual》。

三．Quake3 的场景管理技术

1．Brush

在介绍如何生成 BSP 树之前，这里先介绍 Q3 的一个关键词：Brush。可以这么说：Q3 的场景管理就是建立在 Brush 之上的。

什么是 Brush? 如果用过 Q3 关卡编辑器 Q3Radiant，你就会很清楚：就像名字一样，它是一种表示场景几何体的刷子：既定义了几何体，同时又确定了边界。



Brush 的默认形状是 6 面的长方体，用户可以自己选择其它形状：如三角体，(底面)5 面体等。同时还可以自由的旋转，缩放等。另外，用户还可以通过 CSG(constructive solid geometry)的 Subtract, Merge, Hollow 对多个 Brush 进行操作，以达到如多区域连接等场景构成。

这里就不具体介绍怎么使用了，大家可以参考 Q3Radiant 的手册和网上的教程。

初次接触 Brush 可能都有这样一种想法：为什么要使用 Brush？感觉使用 Brush 会很受限制？

一旦你用会了 Q3Radiant，同时看了 q3map 的相关代码，你就会恍然大悟：

正如前面所说，brush 既定义了几何体，又确定了边界。Brush 能定义几乎你想要的任意几何体（可以接合曲面 Curve/Patch 使用），同时在定义几何体时又确定了边界：brush 的每一个面(brush side)上的多边形(顶点)都在同一个平面上，由各个面闭合成为一个实体。brush 本身就是一个天然的包围体。

因此在生成 BSP 树与碰撞检测时就方便多了。

通常 brush 在 map 文件中都是用如下方式保存的：

例如下面的(q3map 称其为 old stytle brush，但实际上 Q3Radiant 用的也是这种)：

```
// brush 0
{
  ( 218 215 72 ) ( 233 218 72 ) ( 233 218 48 ) sfx/flame5 10 20 90 0.600000 0.500000 134217728 0 5  //一个 brush side 面
...  //其他 brush side 面
}
```

这里前面 3 个坐标是定义此 brush side 平面的顶点数据，
而后面的则是与 shader 纹理相关的：

```
        Shader Name:      sfx/flame5
        h shift(水平平移):        10
        v shift(垂直平移):        20
        rotation:                 90
        h Scale/Stretch: 0.600000
        v Scale/Stretch: 0.500000
    Surface flag:     134217728
        unkonwn:                  0
        value:            5
```

当然，对于更复杂的几何体，可以插入 md3 mesh model。

2 q3map 的运行机制：

场景生成一般是调用三次 q3map 程序：

(1).q3map mapname　　//生成 BSP(.bsp)和 portals(.prt)
(2).q3map -vis mapname　　//生成 PVS
(3).q3map -light mapname　　//生成 lightmap

另外 q3map 为多核心/多 CPU 使用了多线程机制，充分利用了系统资源。

接下来我将接合具体源代码讲解相关的技术。
注意这里的源代码是名为 quake3-1.32b-.source.zip ，大小为 5.45M 的压缩包。(国内可在 http://resource.gameres.com/ 获得)

3. BSP 树

在 ProcessWorldModel 这个函数里，就是对表示世界场景的几何体进行 BSP 相关的处理。世界场景的几何体用 entities[0]表示。
注意曲面不参与生成 BSP 树。叶节点不一定是凸的(有硬件 Z 缓冲帮助)。只有 brush 才参与生成 bsp。
下面给出简易过程：

3.1 从 brush 中产生 bspface：MakeStructuralBspFaceList

注意 Q3 的 BSP 树并不是通过场景的多边形(通常是三角形)分割生成，而是通过 brush 的各个面(brush side)生成。显然这样做要方便得多。注意这些面是外向的，即法线指向实体的外面。
注意 detail brush 不生成 bspface。所谓 detail brush，即《Binary Space Partioning Trees and Polygon Removal in Real Time 3D Rendering》(注：之后提到的**经典算法**即这本书上的算法)中所提到的 static mesh，也就是比较复杂的几何体，如球等。通常它们的三角面太多且大都不在同一个平面，所以分割时会生成更多的叶节点，因此生成 BSP 时不考虑它们，而是在生成完 BSP 树之后将它们加入到相应的叶节点中。
注意虽然 brush 的每个面在 map 文件中是以三个点保存的，但读到内存中时已经确定边

界了：CreateBrushWindings。

在这里介绍一下程序中最常见的一个结构：

```
typedef struct
{
    int     numpoints;
    vec3_t  p[4];       // variable sized
} winding_t;
```

这个结构用来表示一个任意多边形(必须是凸的，且顶点共面)的“缠绕点”，顺时针排列。将它们依次连接起来就构成了此多边形。

因此先将 brush 各个面扩展为 4 个点表示的矩形(因为地图有最大尺寸)，然后用其余各面对此面进行裁剪，显然最后留下的就是此 brush side 的具体形状。

3.2 生成 BSP 树：BuildFaceTree_r

Q3 生成 BSP 树的过程与经典算法基本是一致的，细节有些不一样：

基本框架：

```
void BuildFaceTree_r( node_t *node, bspface_t *list ) {
    i = CountFaceList( list );
    splitPlaneNum = SelectSplitPlaneNum( node, list );//**选择分割平面

    // if we don't have any more faces, this is a leaf
    if ( splitPlaneNum == -1 ) {
        node->planenum = PLANENUM_LEAF;
        return;
    }
    // partition the list
    node->planenum = splitPlaneNum;//**记录分割平面

    plane = &mapplanes[ splitPlaneNum ];

    for ( split = list ; split ; split = next ) {
        next = split->next;

        //**如果face所在的平面与分割平面是同一个
        if ( split->planenum == node->planenum ) {
            FreeBspFace( split );
            continue;
        }

        side = WindingOnPlaneSide( split->w, plane->normal, plane->dist );

        if ( side == SIDE_CROSS )
        {
```

```
            //**注意这里是对任意多边形进行分割
            ClipWindingEpsilon( split->w, plane->normal, plane->dist, CLIP_EPSILON * 2,
                                                    &frontWinding, &backWinding );

            if ( frontWinding ) {
                newFace = AllocBspFace();
                newFace->w = frontWinding;
                newFace->next = childLists[0];
                newFace->planenum = split->planenum;//**子face肯定与父face是同一个平面的
                newFace->priority = split->priority;
                newFace->hint = split->hint;

                childLists[0] = newFace;
            }
            if ( backWinding ) {
                newFace = AllocBspFace();
                newFace->w = backWinding;
                newFace->next = childLists[1];
                newFace->planenum = split->planenum;//**子face肯定与父face是同一个平面的
                newFace->priority = split->priority;
                newFace->hint = split->hint;

                childLists[1] = newFace;
            }
            FreeBspFace( split );//**删除父face
        }
        else if ( side == SIDE_FRONT ) {
            split->next = childLists[0];
            childLists[0] = split;
        }
        else if ( side == SIDE_BACK ) {
            split->next = childLists[1];
            childLists[1] = split;
        }
    }

    for ( i = 0 ; i < 2 ; i++ ) {
        node->children[i] = AllocNode();
        node->children[i]->parent = node;
        VectorCopy( node->mins, node->children[i]->mins );
        VectorCopy( node->maxs, node->children[i]->maxs );
    }
    //**对于用轴平行平面分割后的处理
    for ( i = 0 ; i < 3 ; i++ ) {
```

```
            if ( plane->normal[i] == 1 ) {
                node->children[0]->mins[i] = plane->dist; //**front
                node->children[1]->maxs[i] = plane->dist;//**back
                break;
            }
        }

        // recursively process children
        for ( i = 0 ; i < 2 ; i++ ) {
            BuildFaceTree_r ( node->children[i], childLists[i]);
        }
}
```

3.2.1 选择分割平面 SelectSplitPlaneNum

首先检查此节点(场景)的大小。如果太大，强制将其用轴平行平面分割。（这里只考虑 XY 轴，Z 轴是高，逻辑上可以不用考虑）

然后开始循环检查各个 bspface，每次选择一个并对其余 bspface 进行分割，从而找出最佳分割平面。

注意 Q3 检查只循环一次，并没用像书上那样“有多次且阈值有变化”。

Q3 用到的评测参数有如下几个：分割数 splits，共面数 facing，前面数 front 与后面数 back。注意 Q3 用平面分割的是任意多边形，分割后产生的也是任意多边形，因此分割后最多产生两部分。

```
value =  5*facing - 5*splits; // - abs(front-back); 分枝树比平衡度更重要?
if ( plane->type < 3 ) {
    value+=5;     // axial is better //**对于轴平行平面，优先考虑!
}
value += split->priority;// prioritize hints higher //**本版本的 q3map hint 并没有加上优先级
if ( value > bestValue ) {
    bestValue = value;
    bestSplit = split;
}
```

值得学习的是对于轴平行平面与特殊平面的优待(比如 hint brush)。

注：所谓 hint brush 通常是用来帮助分割 bsp 树的，一般由有经验的关卡设计师手动放置。Hint Brushes are structural, trans, and nonsolid. This means they are totally invisible in a compiled Quake3 level, but their Face-Planes are used to split the BSP like other structural Brush Face. A large Axial Hint brush face will be an ideal candidate to be used for a BSP Split. Adding Hints with Axial faces (perpendicular to the 128-unit grid lines) aligned with other Axial Planes from structural Brush faces, minimises the number of extra split-Planes and Leaf Nodes.

3.2.2 分割 bspface: ClipWindingEpsilon

分割任意多边形的算法核心是：对每个点按顺序检查，如果当前点与下一个点在平面的两侧，则分割之产生新点。

3.2.3 结果

注意对于共面的bspface，Q3是这样处理的：

```
//**如果face所在的平面与分割平面是同一个
if ( split->planenum == node->planenum ) {
//**删除它，这样以后不再用它们分割了(一个平面只能用一次)
    FreeBspFace( split );
    continue;
}
```

当 BSP 生成完之后，每一个叶节点会有一个属性标志：如果 node->opaque 为 qtrue，表示此叶节点是 solid，即此叶节点表示的空间是一个封闭体，视点不可能出现在这里，比如某柱子的里面。通常在 solid leaf 中的 entity 或多边形面是无效的。

3.3 生成 Portal： MakeTreePortals (tree);

Q3 生成 Portal 的方法与经典算法类似，同时也和 GameDev 上的文章:《Automatic Portal Generation》类似。

3.3.1 MakeHeadnodePortals

首先对树的根结点生成 6 个特殊的 Portal：6 个轴平行平面，用来确定界限。

3.3.2 MakeTreePortals_r

Q3 分割 portal 只用分割平面，不用场景中的多边形，所以分割到叶节点为止。到此时，每个 protal 连接着两个叶节点.一个叶节点共享着多个 portal。注意 portal 的连接信息是按前后（front/back side）看的：即此叶节点在此 portal 的前面还是后面。portal_t::nodes[0]表示的是此 portal 的前面叶节点，portal_t::nodes[1] 表示是此 portal 的后面叶节点。而 portal_t::next[0]表示 the next portal of the front node，portal_t::next[1]表示 the next portal of the back node;以此构成链表。

因此如果要访问某个结点的所有相关portal，则使用下面的方法(非常有技巧性)：

```
portal_t *p;
for (p = node->portals ; p ; p = p->next[s])
{
    s = (p->nodes[1] == node);//**is back node? 完全利用C语言的逻辑特点
    //do something
}
```

每个结点的 portal 只可能由两种方式产生：1.通过自己的分割平面 2.祖先结点传下来的 Portal。

整个算法的框架是：

```
void MakeTreePortals_r (node_t *node)
{
    CalcNodeBounds (node);//**通过node的portal计算边界

    if (node->planenum == PLANENUM_LEAF) //**分割到叶节点为止
        return;

    MakeNodePortal (node); //**产生portal
    SplitNodePortals (node);//**分割portal，并传递给children

    //**递归处理children
    MakeTreePortals_r (node->children[0]);
    MakeTreePortals_r (node->children[1]);
}
```

3.3.2.1 MakeNodePortal

每个结点自己产生的 Portal 最初由分割平面产生，并且要被祖先结点裁剪：BaseWindingForNode。

然后这个新产生的 portal 还要被此结点的其他 portal(祖先结点传下来的)裁剪。

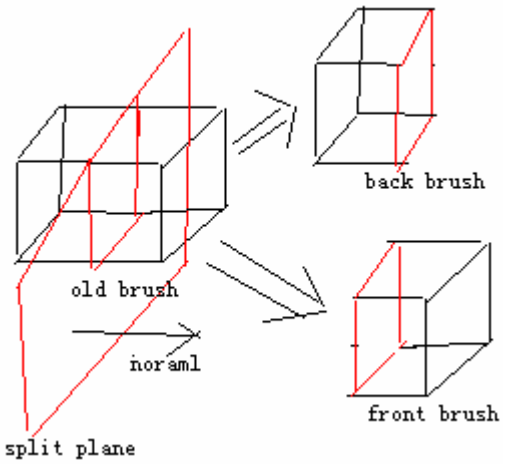
最后，让此 portal 连接上左右子树。

3.3.2.2 SplitNodePortals

现在用当前结点的分割平面对当前结点的所有 portal 进行分割,并按分割后的部分在平面的哪一侧作为依据向下传递。注意由于每个 portal 之前已经连接了两个结点，所以在被当前结点的分割平面分割之后，此 portal(或分割后的两部分)将重新建立连接信息：一端继续连接原来的另一个 node，一端连接着此 node 的某子结点。

3.4 分割 brush：FilterStructuralBrushesIntoTree

将所有的 brush（除 detail brush 之外）用 BSP 树进行分割，并加到相应的叶节点中。这一步是为了以后方便碰撞检测而做的。



注意 SplitBrush 这个函数的核心思想：对 brush 的分割也是最多分割成两部分，注意由于 brush 是立体的，所以分割之后要多产生一个“横截面”来构成闭合区间，这个“横截

面”是分割平面被 brush 的各个 brush side 分割产生的。也就是说先用 brush 的各个面去分割“结点的分割平面”，产生一个“横截面”；然后再用结点的分割平面去分割 brush 的各个面，产生两个子 brush。每个子 brush 加上“横截面”构成新的 brush。

3.5 检测此 BSP 树是否是闭合的：FloodEntities

FloodEntities 函数实际上是将地图中的 Entity 加到 BSP 树的各个相应叶节点中，而所谓的 Entity 指的是:“game-related map information, including information about the map name, weapons, health, armor, triggers, spawn points, lights, and .md3 models to be placed in the map.”

加入的依据是根据此实体的坐标位置在分割平面的哪一侧来遍历二叉树。

如果所有的 Entity 都能加入进去，则说明此 BSP 树是合格的，或者说场景是合格的。

3.6 ClipSidesIntoTree

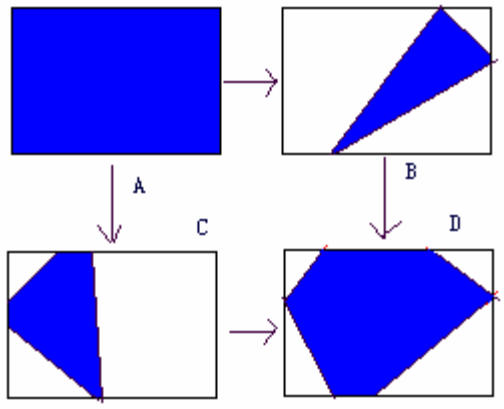
用最初产生的 bsp 树对每个 brush 的 brushside(具体来说是 winding)进行分割。

这样做的目的是为了将 brushside 中的 winding 缩减为最小集(只要可见的)，并用它来够成真正的（可见）“场景多边形面”。这些最小集保存在 side->visibleHull 中。

3.6.1 ClipSideIntoTree_r

分割部分是很简单的，这里不用提了。重点说明一下 AddWindingToConvexHull 这个函数：即将最小集如何加到 side->visibleHull 中去。

如下图所示：



最初的一个 brush side A 是四边形，经过分割到最后，产生了两个子 brush sdie B 和 C，然后将它们组合成一个 ConvexHull。很显然，windingB + windingC <= newwinding，因为要保证是凸的，就可能要扩充面积。最后这个 winding 构成的多边形就是能看到的多边形。

而对于其他一些 brush side，如果完全不可见或者在 BSP 树之外，则 ConvexHull 为 NULL。在之后产生场景多边形和重新生成 BSP 树时，就不考虑它们了。

3.6.2 构成场景多边形：DrawSurfaceForSide

此时暂时没有将多边形拆分成三角形 list。注意一下纹理坐标转换算法：

```
dv->st[0] = s->vecs[0][3] + DotProduct( s->vecs[0],dv->xyz );
dv->st[1] = s->vecs[1][3] + DotProduct( s->vecs[1],dv->xyz );
```

```
dv->st[0] /= si->width;
dv->st[1] /= si->height;
```

直接将三维空间的点转换到纹理坐标。注意到 s->vecs[0]前三个数据是关于 s 轴的旋转与缩放的组合，最后一个数据 s->vecs[0][3]是平移值。s->vecs[1]相对于 t 轴同理。

对于 new stytle brush，原理类似。我将在报告的最后重点说明相关的技术。

以后对此 brush side 的其余点求纹理坐标，先通过三个点找到 XYZ 和 st 的对应关系(3 个方程组解三个未知数)，再用这个变换关系直接作用于顶点的三维坐标。这样就无需对纹理坐标进行插值，对于任意多边形的控制也相当方便。

3.7 重新生成 bspface：MakeVisibleBspFaceList

现在重新产生 bspface，但它们只从含有（可见）“场景多边形面”的 brush side 中产生(visibleHull!=NULL)

这样做有什么好处？很明显：

1.抛弃了那些 visibleHull 为 NULL 的 brushside，这样参与生成 bsp 的面少了

2.由于 winding 变小了，以前被分割的那些 bspface，现在可能只是在 front 或者 back 了。

由于上面两个原因，bsp 简化了，节点少了(比如我测试过某地图第一次生成 200 多 leaf，第二次则只有 60 多个)。

3.7 重新生成 BSP 树

```
FreeTree (tree);
tree = FaceBSP( faces );//**rebulid a better bsp
MakeTreePortals( tree );
FilterStructuralBrushesIntoTree( e, tree );
```

3.8 其他设置

```
NumberClusters //对每个可见 leaf 编号，同时计算 portals 数
FloodAreas;//**对地图进行区域编号，并对是 areaportal 的 brush 填充连接两相邻区域的信息
FilterDetailBrushesIntoTree;//**将 detail brush 加到 bsp tree 相应叶节点中
AddTriangleModels  //**将地图中的 mesh model(md3)变成地图场景中的一部分(变成场景多边形面)
SubdivideDrawSurfs//**对要求 subdivide 的面进行 subdivide(即多边形划分成更小的多边形)
```

3.9 保存 Portal 相关数据

在之前 NumberClusters\ NumberLeafs_r 这个函数中，对 Portal 计数。

对于连接两个非 solid(opaque)叶节点的 portal，称为可通过(passage)的 portal，每个这种合格的 portal 只计数一次。

接下来保存相关数据到 prt 文件中：WritePortalFile。

```
fprintf (pf, "%s\n", PORTALFILE);
fprintf (pf, "%i\n", num_visclusters);//**num of vis leaves(non opaque)
fprintf (pf, "%i\n", num_visportals);//**num of passable portals
fprintf (pf, "%i\n", num_solidfaces);//**num of impassable portals
```

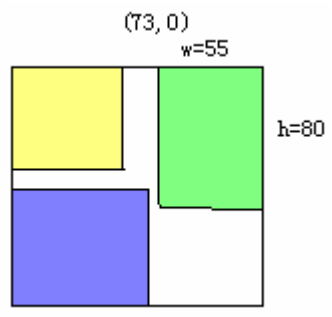
```
WritePortalFile_r(tree->headnode);//**记录能通过的portal(两边的叶节点是nonsolid)，只记录一次（只记录front的portal）
WriteFaceFile_r(tree->headnode);//**记录通不过的 portal(背面用逆序)
```

实际上通不过的 portal 是没有用的。

3.10 分配 lightmap 数据区：AllocateLightmaps

Q3 的 lightmap 大小是 128*128,一个 lightmap 可能被多个渲染面(mapDrawSurface_t)共享，注意这里的“共享”只是宏观共享，即它们去占用此 lightmap 没被使用的部分。例如一个 lightmap 可以被 4 个需要 64*64 大小的 lightmap 数据的渲染面“共享”。

例如：



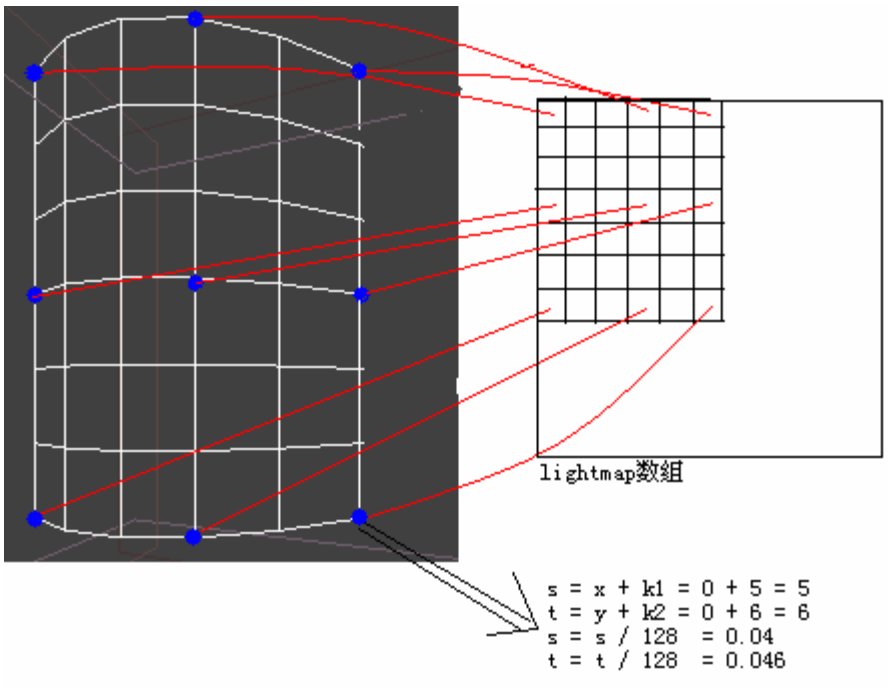
此 lightmap（假设 ID 号为 5）被分成三块，其中绿色的被某渲染面所有：

```
ds->lightmapNum = 5;
ds->lightmapWidth = 55;
ds->lightmapHeight = 80;
ds->lightmapX = 73;
ds->lightmapY = 0;
```

仔细观察发现场景中的 mesh mode 并没有分配 lightmap。原因我想可能是：mesh 本身三角形面数较多，三角形小，所以没有必要细化，只算出顶点的光照即可。

给曲面和普通渲染面分配的方式是不一样的。

对于曲面(AllocateLightmapForPatch)，先对控制点进行细分(只是临时的)：超过指定大小(如两点之间的距离)的要细分(SubdivideMesh),超过“多少 3D 尺寸对应一个 lightmap pixel”的大小 ssize(默认为 16)的也要细分(SubdivideMeshQuads)。细化之后找到原控制点在相应 lightmap 中的位置(即 lightmap 坐标)。例如：



而对于普通渲染面，是直接将三维坐标变换到二维坐标系中。将在后面介绍。

3.11 将 Drawsurfs 加到相应的叶节点中：FilterDrawsurfsIntoTree

即将场景渲染多边形加到相应的叶节点中。

注意 Q3 的 DrawSurface 都是用 bsp 进行逻辑分割，以此确定 DrawSurface 在哪些叶节点中。即：一个 DrawSurface 可能和多个叶节点相关，这些叶节点都保存着指向这个完整 DrawSurface 的索引。

这样做虽然简单省事，但在渲染前必须检测以避免重复渲染。

在递归查找相关叶节点时，曲面和 mesh 都是先用每个三角面查找，然后再用每个顶点查找(更保险)。Billboard 是用中心点查找，而普通多边形是用自己的 Winding(即顶点组)。

4. PVS

计算 PVS 时强烈建议像 Q3 一样使用多线程技术。

Q3 计算 PVS 与经典算法不一样：

1.不设置采样点(sample point) 2.不使用光线跟踪 3.不考虑场景多边形的遮挡。

完全是通过 Portal 与 Portal 之间进行方位比较得出的（有利用到包围球）。

因此 Q3 的 PVS 计算比较“粗”，每个叶节点的 PVS 集比较多，我个人感觉不好。还是应该使用光线跟踪加上多边形的遮挡判断。

另外注意 Q3 用了逐步求精的方式，每一次的测试都是依据上一次的结果，如果某 portal 不满足上一次的要求，则本次将不予考虑。

4.1 加载 Portal 数据：LoadPortals

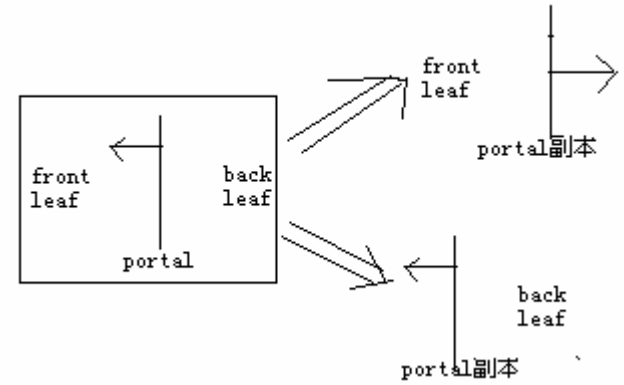
Q3 判断 PVS 数据用到了位判断：即一个 leaf 占一位，一个 portal 占一位。并且为了利用 64 位系统的优势，强制使用了 64bit 为基本单位(因此会有空间的浪费，但可忽略)，这样即使是 32 位的系统，也就是 2 个机器字长，是字长的偶数，访问也比较快。"x+63"保证至少是一个 64 位的，之后"&~63"保证了“去除掉”不到 64 位的低位，">>3"即/8，换算成字节数。

当具体访问时,如果当前项与某 j 项有关(这里表示可见)，则置当前项的 BitBuffer[j>>3]的第(1<<(j&7))位为 1。

```
leafbytes = ((portalclusters+63)&~63)>>3;
leaflongs = leafbytes/sizeof(long);

portalbytes = ((numportals*2+63)&~63)>>3;
portallongs = portalbytes/sizeof(long);
```

由于文件中的 passage portal 只保存的 front leaf 的 portal，所以读到内存中还要把背向的重新加上(面向 back leaf)。注意这两个 portal 实际上是同一个 portal，因此 ID 号分配是一样的，但是为了方便计算，构造了两个稍有不同的副本。

总体要求是：顶点连接顺序是内向的(即从 leaf 看去是顺时针排列)，但 portal 所在平面外向(即法线外指，朝向另一边的 leaf)。这些要求起的作用将在计算 PVS 时展现出来。




```
p->num = i+1;//**portal id
p->hint = hint;
p->winding = w; //**顶点要求连接顺序是内向
//**portal外向normal pointing into neighbor
VectorSubtract (vec3_origin, plane.normal, p->plane.normal);//** -normal
p->plane.dist = -plane.dist;//** -d
p->leaf = leafnums[1];//**neighbor
SetPortalSphere (p);//**包围球
p++;

// create backwards portal
//**for back leaf
l = &leafs[leafnums[1]];
if (l->numportals == MAX_PORTALS_ON_LEAF)
    Error ("Leaf with too many portals");
```

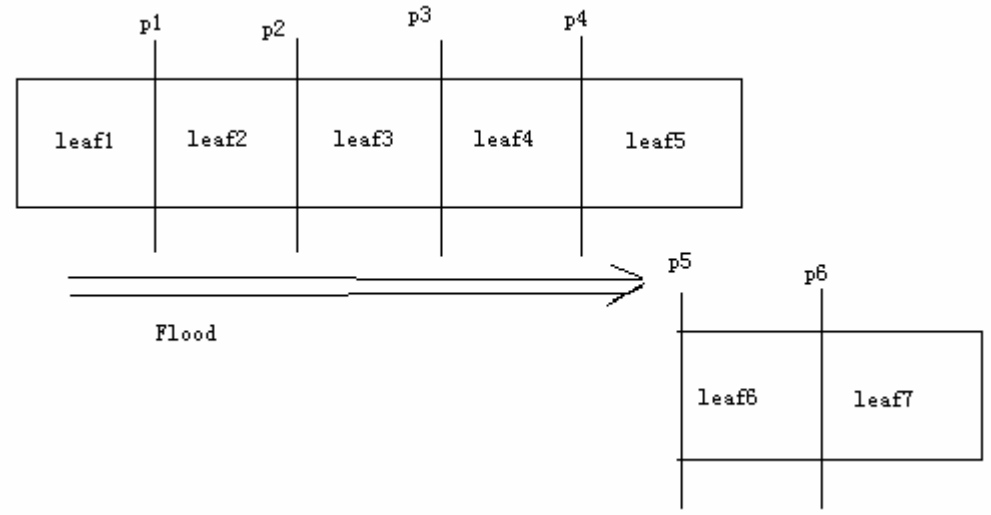
```
l->portals[l->numportals] = p;
l->numportals++;

p->num = i+1;//**注意正和背id号是一样的
p->hint = hint;
p->winding = NewWinding(w->numpoints);
p->winding->numpoints = w->numpoints;
//**顶点要求连接顺序是内向
for (j=0 ; j<w->numpoints ; j++)
{
    VectorCopy (w->points[w->numpoints-1-j], p->winding->points[j]);//**逆向copy
}
//**portal对背向leaf来说，本身就是外向
p->plane = plane;
p->leaf = leafnums[0];
SetPortalSphere (p);
p++;
```

4.2 预处理：BasePortalVis

通过此函数对所有 portal 进行两次预处理：一次是正面选择，一次是 flood 选择。

所谓正面选择，是由于之前在加载 Portal 数据时已经将所有 portal 都设成“外向”了，因此只有当两个 portal 互相正对时(即互相在对方前面，凸集)才有可能互见。此预处理填充 vportal_t::portalfront 域。

所谓 Flood 选择(SimpleFlood)：Flood 这里可以翻译成“渗透，注满”，意思是像水一样渐渐蔓延到相邻的区域。这是第二次预处理：用当前 portal 渗透地检测相邻叶节点，“相邻叶节点”的相邻叶节点... 此预处理填充 vportal_t:: portalflood 域。



如上图所示，在 p1 的 flood 中，虽然 p5,p6 在它的前面，但由于没有路径可以到达，所以 flood 不通过。

```
void SimpleFlood (vportal_t *srcportal, int leafnum)
```

```
{
    int      i;
    leaf_t   *leaf;
    vportal_t*p;
    int      pnum;

    //**当前srcportal正对着当前leaf
    leaf = &leafs[leafnum];//**neighbor leaf

    for (i=0 ; i<leaf->numportals ; i++)
    {
        p = leaf->portals[i];
        if (p->removed)
            continue;

        pnum = p - portals;//**get portal index
    if ( ! (srcportal->portalfront[pnum>>3] & (1<<(pnum&7)) ) ) //**没正对则一点机会都没有
            continue;

        if (srcportal->portalflood[pnum>>3] & (1<<(pnum&7)) )//**已经设置了
            continue;

        srcportal->portalflood[pnum>>3] |= (1<<(pnum&7));

        SimpleFlood (srcportal, p->leaf);//**继续渗透此portal的neighbor leaf
    }
}
```

4.3 按可能可见数 nummightsee 对 portal 们进行排序：SortPortals;

通过两次预处理，每个 Portal 的 pvs 数可以通过如下方式得到：

```
//**计算当前leaf可能会看见多少其他leaf
p->nummightsee = CountBits (p->portalflood, numportals*2);
```

然后使用快速排序法进行排序，从少到多排。(Sorts the portals from the least complex, so the later ones can reuse the earlier information.)

4.4 主算法：CalcPassagePortalVis

4.4.1 CreatePassages

CreatePassages 创建 passage 通道，这些通道由当前的 portal(source)与它相邻叶节点中的其他 portal(target)构成，然后对其它可能通过此通道看见的 portal 进行 bit 设置。

这里有一个重要的函数：AddSeperators：

```
int AddSeperators (winding_t *source, winding_t *pass, qboolean flipclip, plane_t *seperators,
```

```
                    int maxseperators);
/*
flipclip = false 函数返回时：source在平面后，pass在平面前
flipclip = true  函数返回时：source在平面前，pass在平面后
这些构成passage的平面是由source的一条边(si,sj)和source与pass构成的一条边(si,pk)组成的：即
source取2个点，pass取1个点。source的每条边只能在平面中用一次，也就是说，最多得到source边数个
平面
*/
```

这个函数的作用是：通过上面的方法创建通道平面，每个平面必须保证满足上面的要求，即将 source 和 target 区分在它两侧。

它在 CreatePassages 是这样用的：

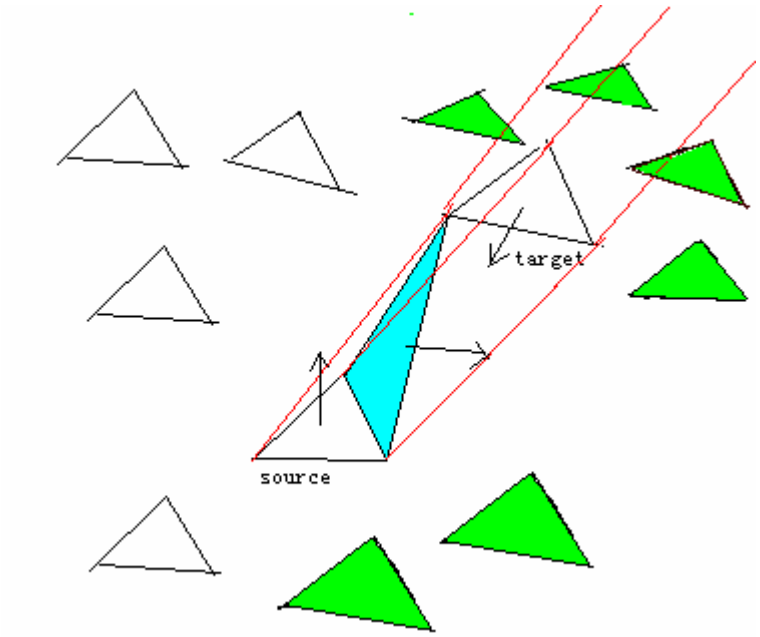
```
numseperators = AddSeperators(portal->winding, target->winding, qfalse, seperators,
                              MAX_SEPERATORS*2);

numseperators += AddSeperators(target->winding, portal->winding, qtrue,
                               &seperators[numseperators], MAX_SEPERATORS*2-numseperators);
```

由上可知，经过两次调用，source portal 在这些平面的后面，target 在这些平面的前面。

接下来用这些通道判断其余所有 portalflood 中的 portal(在之前通过 flood 处理后可能被 source portal 和 target 同时看到的 portal)在它的哪一侧，如果和 target 在同一侧(前面)，则是可能可见的。

如何判断是同一侧？只有对所有的 seperators 平面满足要求时(portal 有点在它们前面)，即被裁减之后还有剩余点在这些平面的前面，则说明空间位置与 target 类似，所以有可能看见，然后设置 bit 位：passage->cansee[j >> 3] |= (1<<(j&7));

图例说明：



为了简化，我们用三角形的 portal 作例子，同时假设只有一个合格的 seperator 平面(用

蓝色表示)。显然所有绿色的 portal 在本次通过了测试（均有点在 seperator 平面的前面）。(注：红色线构成的区间是真正的可视区间)。

4.4.2 PassagePortalFlow

PassagePortalFlow 使用的技术类似于 CreatePassages，区别在于：它更细，更严格，只对两个 portal 之间观察方向的其他 portal 进行考察。即以上图为例，利用了“红线”。

这里讲讲 RecursivePassagePortalFlow 这个函数：它是 pvs 计算最核心的函数。

Q3 计算 pvs 没用光线跟踪，并且也没有考虑场景多边形可能产生的阻挡。完全只用 portal。

整个算法是基于以下原理：

1.每个 portal 只连接 2 个 leaf

2.如果两个 leaf 之间能互相看见，要么它们之间有 portal 直接连接，要么它们之间有间接 portal 连接

所谓间接 portal，指的是它们和其他叶节点之间的 portal，以及这些叶节点再和其他叶节点之间的 portal...

即：Leaf A can see B = PortalAB

= PortalA1 + Portal12 + Porta23 + . . . + Portalmn + PortalnB

(+号是连接的意思)

而判断是否有间接 portal 的方法类似于第三次的预处理：CreatePassages

用当前 portal(base portal)与上一级(因为是递归处理)已经判断可以看见的 portal 组成 passage:即产生多个划分平面，然后用这些平面对要考察的 portal(这个 portal 是上一级 portal 相邻叶节点的 portal)进行裁剪，如果裁剪之后还有点通过，则说明它在 base portal 通向上一级 portal 的方向上，说明能够看到它。

这里的函数：ClipToSeperators 的作用是：用“逐渐裁剪”之后 base portal 的 winding(每次取 2 个点)与上一级 portal 通过的 winding(每次取一个点)构成一些平面(CreatePassage 相同的算法)，再用这些平面去裁剪当前 portal 的 winding。

注意到 base portal 的 winding 每次都可能会被当前考察的 portal 裁剪(VisChopWinding)，同样当前考察的 portal 也会被 base portal 裁剪，这样得到的通道会更精确。

5. Lightmap

Q3 并没有使用辐射度计算光照，而是使用了简单的光线跟踪(也没有考虑反射与折射)。我在网上查到的原因之一是：“Quake 1 and 3 used a much simpler lighting model. The artists decided that they could get the look they were after, much more easily with the simpler lighting model. So although the quake 2 lighting is more physically accurate, it just wasn't as good for making levels”。

加上我自己的总结如下：

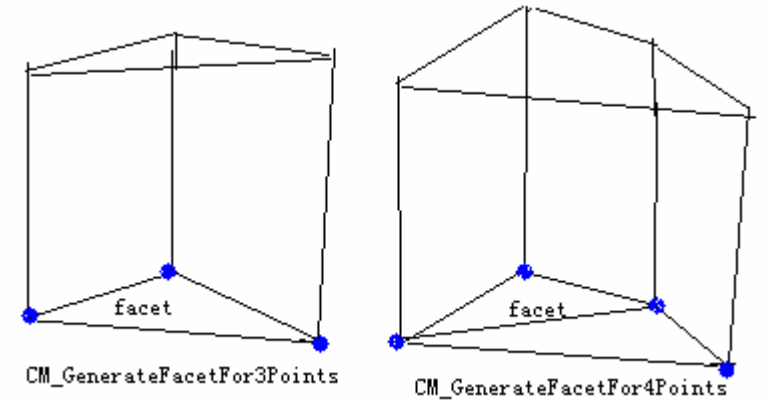
(1) 辐射度算法太复杂(要切割成小平面，能量传递，复杂的光线跟踪)，太费时间

(2) 辐射度算法很难使用多线程控制
(3) 辐射度算法很难控制达到自己想要的光照效果，而美工可以方便自由地使用图像工具做到。

Q3 的光源有以下几种类型：

```
typedef enum
{
    emit_point,     //**point light
    emit_area,      //**surface light，由场景多边形产生
    emit_spotlight, //**spot light
    emit_sun       //**direction light
} emittype_t;
```

5.1 组合小平面：InitSurfacesForTesting



除了 billborad 不计算光照外，其余三种面都要先组合成小平面(Facet)简化计算光照。Facet 是三角形或者由两个共面三角形组成的四边形。注意 Facet 的每条边都建立一个边界平面（用法线 cross product 每边，产生的平面法线外指），显然由所有边界平面就可以构成一个半闭区间(头没有盖上)。这些边界平面是用来判断点是否在 Facet 上的，后述。

对于普通面与 md3 mesh model 面，在 FacetsForTriangleSurface 直接生成。
对于曲面，要像在分配 lightmap 时那样先细化控制点之后由这些控制点组合成 Facet。

这里要注意 TextureMatrixFromPoints，通过三个顶点的纹理坐标还原三维顶点到纹理坐标的映射关系。由于只有三个方程，所以只能找到三个未知数，因此平移值取为 0。Q3 用矩阵解方程组，即对方程组的增广矩阵 AB 进行初等变换，简化为阶梯形矩阵之后求出。

5.2 由能发光的 Surface 创建光源：CreateSurfaceLights

由于 Q3 有一些特殊的 shader 能让平面发光(比如日光灯，火焰)，所以要先从这些平面中产生光源。

对于 autosprite 为 qtrue 的 shader，即 billboard，看成点光源(emit_point)。它的初始光强为：`dl->photons = ls->value * pointScale; //**光(量)子，即总强度`

其余看成 emit_area,即平面光。对于由曲面或 md3 model 产生的面，每个 Facet 是一个光面；对于一个普通平面则是一个光面。注意如果有些光面要求建双面（ls->twoSided），

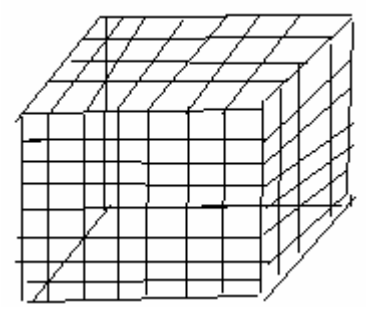
则还要建一个背面照射身后。如果这些光面超过了指定的最大尺寸 lightSubdivide，则要细分成更小的发光平面（SubdivideAreaLight）。

```
intensity = value * area * areaScale;//**总强度由光源本身及面积决定
```

5.3 LightWorld

5.3.1 SetupGrid

创建空间光照格子。可以将场景空间想像成是由一个一个的立方体组成的。



Lightvols

The lightvols lump stores a uniform grid of lighting information used to illuminate non-map objects. There are a total of length / sizeof(lightvol) records in the lump, where length is the size of the lump itself, as specified in the lump directory.

```
Lightvols make up a 3D grid whose dimensions are:
nx = floor(models[0].maxs[0] / 64) - ceil(models[0].mins[0] / 64) + 1
ny = floor(models[0].maxs[1] / 64) - ceil(models[0].mins[1] / 64) + 1
nz = floor(models[0].maxs[2] / 128) - ceil(models[0].mins[2] / 128) + 1

ubyte[3] ambient  Ambient color component. RGB.
ubyte[3] directional  Directional color component. RGB.
ubyte[2] dir  Direction to light. 0=phi, 1=theta.

vec3_t    gridSize = { 64, 64, 128 };
*/
for ( i = 0 ; i < 3 ; i++ ) {
    gridMins[i] = gridSize[i] * ceil( dmodels[0].mins[i] / gridSize[i] );
    maxs[i] = gridSize[i] * floor( dmodels[0].maxs[i] / gridSize[i] );
    gridBounds[i] = (maxs[i] - gridMins[i])/gridSize[i] + 1;
}
numGridPoints = gridBounds[0] * gridBounds[1] * gridBounds[2];
```

不过目前我还是没想明白为什么要设空间格子：后面它们将保存光照值，但为什么要保存呢？lightmap 存的是场景的光照值，grid 存的光照值有什么用？难道是将它们用在场景中活动的 md3 model 上？现在的游戏中，活动的 mesh model 光照通常应该是动态计算的吧？难道是因为当时硬件太慢，所以区域中的 diffuse color 全部先计算了？也不对啊，这些光照与法线是相关的，不同的 mesh 法线也不一样啊？Q3 中是这么说的“Grid samples are foe quickly determining the lighting of dynamically placed entities in the world”看

来的确是这样。不过现在这个技术可以不使用了，除平面光之外的其他光源(point, spot, direction)都可以直接交给硬件动态实现了。

5.3.2 CreateEntityLights

从 Entity 的 light 中创建光源(即那些手动添加的 light)。这些 light 是“真正”的光源。

5.3.3 TraceGrid

用 TraceGrid 先计算每个格子得到的**环境光**值(由所有的光共同作用的效果),即 Lightvols。

在 LightContributionToPoint 中求的是环境光照值。注意平面光的原点 origin 是平均求出的重心。

注意 Q3 的光线跟踪是先判断此光线经过了哪些叶节点（r = TraceLine_r(0, start, stop, tw)）。判断经过哪些叶节点的方法是：将起点与终点与当前结点的分割平面进行判断，如果在同一侧，则继续判断那一侧的子结点；如果在不同侧，则求交点(用离平面距离的比值求)。然后两段射线再各自放到相应子结点中继续判断。当遇到叶节点时：

```
if (node & (1<<31)) { //**is leaf
    //**如果遇到solid叶节点，则肯定碰撞了
    if (node & ( 1 << 30 ) ) { //**solid
        VectorCopy (start, tw->trace->hit);
        tw->trace->passSolid = qtrue;
        return qtrue;
    }
    else { //**如果是可见叶节点，则暂时没有碰撞，还要通过其它方法判断(是否与多边形相交)
        // save the node off for more exact testing
        if ( tw->numOpenLeafs == MAX_MAP_LEAFS ) {
            return qfalse;
        }
        //**记录所有经过的可见叶节点
        tw->openLeafNumbers[ tw->numOpenLeafs ] = node & ~(3 << 30);//**get leaf num
        tw->numOpenLeafs++;

        return qfalse;
    }
}
```

如果与可见叶节点相交，则要判断是否与叶节点中的多边形相交：TraceAgainstSurface。先用包围球排除不必要的面，剩下的用 TraceAgainstFacet 对每个面的小 Facet 进行判断：

对于射线的起点与终点分别与 Facet 所在的平面进行判断求位移，如果分别在 Facet 的两侧，并且如果当前交点在射线长度中的比值 f 比当前的 hitFraction 小，则说明是最近的交点，则求交点(比值为 0 表示原始射线的起点，比值为 1 表示原始射线的终点，这是光线跟踪中很常用的方法)。然后让交点与 Facet 的所有边界平面测试看是否在 Facet 上：注意 Q3 没用射线与三角形求交算法判断交点。因为 Facet 既可能是三角形，又可能是四边形。

如果用三角形求交算法，则要将四边形重新拆成两个三角形。所以在创建 facet 时，构造了包围平面，用包围平面判断更快更方便！只要交点在所有包围平面构成的空间内部即可(由于包围平面全部外指，所以只要求全在它们的背面)。

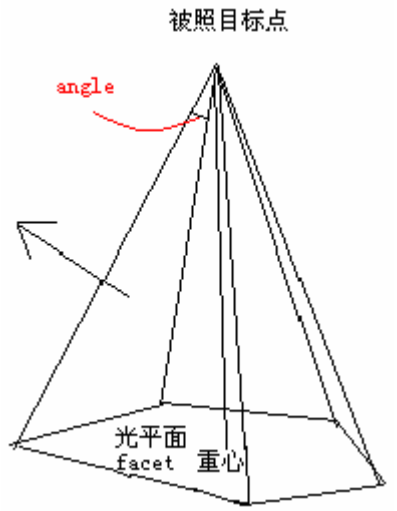
注意Facet的属性可能是带透明效果的。

```
// if this is a transparent surface, calculate filter value
if ( shader->surfaceFlags & SURF_ALPHASHADOW ) {
    SetFacetFilter( tr, shader, facet, point ); //**对于透明的，可以通过，但要修改Filter值
}
else {
    // completely opaque
    VectorClear( tr->trace->filter );
    tr->trace->hitFraction = f;
}
```

Q3对于透明的面(带了SURF_ALPHASHADOW属性，即透明面也要产生阴影)，用了简化处理。在SetFacetFilter中，从纹理中取得相应的alpha值：

```
    // alpha test makes this a binary option
    b = b < 128 ? 0 : 255;//**这里简化了，只有两种情况
    //**这样要么是0,要么不变
    tr->trace->filter[0] = tr->trace->filter[0] * (255-b) / 255;
    tr->trace->filter[1] = tr->trace->filter[1] * (255-b) / 255;
    r->trace->filter[2] = tr->trace->filter[2] * (255-b) / 255;
```

也就是说，对于透明的，可以通过，但要修改 Filter 值。



对于平面光，要求形状因子：PointToPolygonFormFactor。 算法思想是：点离平面越远，则构成的每个三角形(目标点与光面上的 winding 点构成)的夹角 angle 越小，三角形法线与“目标点到光面重心的连线”的夹角越大，则点积 cos 值 facing 越小。而 total += facing * angle，显然点离平面越远，求得的因子绝对值越小。换句话说，因子与目标点与光面构成的锥体体积有关。

对每个光源都计算一次环境光，然后累加起来：

```
    VectorSubtract( light->origin, origin, dir );
```

```
        VectorNormalize( dir, dir );
        VectorCopy( add, contributions[numCon].color ); //**此光贡献的color
        VectorCopy( dir, contributions[numCon].dir );//**此光的方向
        numCon++;
        addSize = VectorLength( add );//**亮度
        //**累加光照方向。显然亮度越大，此光的方向所占的比重就越大
        VectorMA( summedDir, addSize, dir, summedDir );
```

最后计算总体环境光与方向光：

```
    // now that we have identified the primary light direction,
    // go back and separate all the light into directed and ambient
    VectorNormalize( summedDir, summedDir ); //**求得光照最主要的方向
    VectorCopy( ambientColor, color ); //**得到最初设置的环境光
    VectorClear( directedColor );
    for ( i = 0 ; i < numCon ; i++ ) {
        float    d;
        //**注意方向都是点到光
    d = DotProduct( contributions[i].dir, summedDir );//**求每个方向光与主方向之间的夹角
        if ( d < 0 ) {
            d = 0;
        }
        //**将此方向光的光照投影到主方向上
        VectorMA( directedColor, d, contributions[i].color, directedColor );
        // the ambient light will be at 1/4 the value of directed light
        d = 0.25 * ( 1.0 - d ); //**经验公式?
        VectorMA( color, d, contributions[i].color, color );//**累加环境光
    }
    // now do some fudging to keep the ambient from being too low
        VectorMA( color, 0.25, directedColor, color );
```

最后将数据保存到 grid 中：

```
ColorToBytes( color, gridData + num*8 );//**byte 0,1,2 ambient color
ColorToBytes( directedColor, gridData + num*8 + 3 ); //**byte 3,4,5 directed color
VectorNormalize( summedDir, summedDir );
NormalToLatLong( summedDir, gridData + num*8 + 6);//**two byte encoded normals 6,7 summed dir
```

5.3.4 计算 lightmap：TraceLtm

这里再重复一下：Q3 对于 md3 mesh，只简单地计算每个点的光照。Md3 mesh model 不参加 lightmap 的运算：因为 Q3 的 lightmap 不用辐射度算法，而用普通的光线跟踪，并只是对顶点之间的光照进行重新细化计算(即对两个顶点之间再分出更多的顶点)由于 mesh 本身三角形数多，三角形小，所以没有必要细化，只算出顶点的光照即可。

首先是顶点光照：

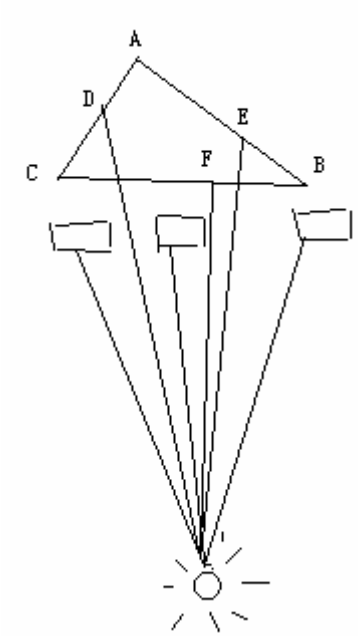
在 VertexLighting 中调用 LightingAtSample 对顶点进行光照计算。每个顶点的光照值是所有光源光照值累加的结果。具体就不讲了，步骤与前面类似，只是公式换成了计算

diffuse 的：I = Id * Kd * N.L。注意不同类型的光计算有不同，特别是对于 spot light，类似于 D3D 中的计算方法。

注意在计算顶点光照时，阴影的计算是很简单的，要么不产生(光线全通过)，要么产生(光线不通过)。

接下来就是 lightmap 的计算。

其实 lightmap 的计算简言之，即在顶点之间插值产生新的顶点，然后计算这些新顶点的光照值。原因如下图：



很明显，如果只计算此三角形的三个顶点值(ABC)，则光源照射不到它们。但插值细分出更多点的话(DEF)，则可以照亮。当然，这只是一种情况。通常大多数情况是：顶点插值(相当于 Phong Shading)产生的光照过渡更加自然，而颜色插值(Gouraud Shading)产生的光照相比要差一些。

当然，如果三角形本身很小的话差别会很小。

注意 Q3 中可以使用类似“抗锯齿”(Antialiasing)的方法对 lightmap 的结果进行“超采样”。即：计算的时候分配更大的 lightmap（一般是 2 倍大）,计算结果用大 lightmap 的周围点取平均值。

那如何对顶点进行插值呢？其实并不是真正在两个顶点之间插值，而是通过 lightmap 坐标找到对应的三维顶点坐标(对于曲面要先细分出控制点，利用控制点找)。

另外，对于被遮挡的采样点，通过周围的点的平均值计算它的光照点(类似于超采样)。

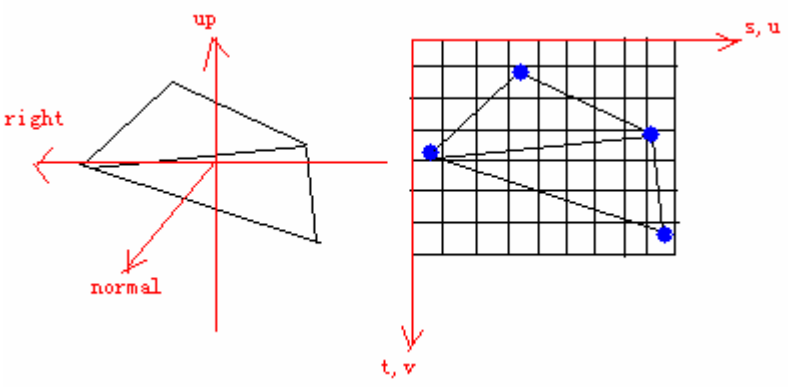
四．Quake3 的其它技术与技巧

1．三维顶点到纹理坐标的映射

Q3 中每一个顶点并不存在纹理坐标这样的东西(指 winding 中的顶点，转换为渲染面时还是有纹理坐标的，废话...)，它为每一个 face 都建立纹理坐标系和 lightmap 坐标系，如果需要获得顶点对应的纹理坐标和 lightmap 坐标，只需要将顶点坐标通过公式(矩阵)变换

到纹理空间就可以获得了。这样做的好处是避免了插值计算，毕竟 Q3 中的多边形是任意的。

如下图所示：



Q3 对于纹理坐标变换有两种方法。

1.1 一种是老方法(BPRIMIT_OLDBRUSHES)：

先找一个与当前平面最接近的轴平行平面。然后将此轴平行平面上的坐标基当作纹理坐标的轴。

```
/*下面一共六个平面，每个平面三个vect3数据。从左向右：第一个是法线，后两个是平面上的轴
注意：q3的纹理坐标系是对称的。/
vec3_t    baseaxis[18] =
{
{0,0,1}, {1,0,0}, {0,-1,0},          // floor
{0,0,-1}, {1,0,0}, {0,-1,0},         // ceiling
{1,0,0}, {0,1,0}, {0,0,-1},          // west wall
{-1,0,0}, {0,1,0}, {0,0,-1},         // east wall
{0,1,0}, {1,0,0}, {0,0,-1},          // south wall
{0,-1,0}, {1,0,0}, {0,0,-1}          // north wall
};
void TextureAxisFromPlane(plane_t *pln, vec3_t xv, vec3_t yv)
{
    int      bestaxis;
    vec_t    dot,best;
    int      i;
    best = 0;
    bestaxis = 0;

    //**选两个法线之间夹角最小的那个平面，这样纹理平面与多边形平面夹角也更小，尽量重合
    for (i=0 ; i<6 ; i++)
    {
        dot = DotProduct (pln->normal, baseaxis[i*3]);
        if (dot > best)
        {
            best = dot;
            bestaxis = i;
```

```
        }
    }
    VectorCopy (baseaxis[bestaxis*3+1], xv);
    VectorCopy (baseaxis[bestaxis*3+2], yv);
}
```

然后组合成变换矩阵：

```
==================
QuakeTextureVecs

Creates world-to-texture mapping vecs for crappy quake plane arrangements
==================
void QuakeTextureVecs( plane_t *plane, vec_t shift[2], vec_t rotate, vec_t scale[2],
                       vec_t mappingVecs[2][4] ) {
    vec3_t    vecs[2];
    int       sv, tv;
    vec_t     ang, sinv, cosv;
    vec_t     ns, nt;
    int       i, j;

    //**从平面中找到合适的纹理坐标轴(注意并不是准确的平面轴[BaseWindingForPlane那种方法],
    //**而是从6个轴平面中选，这样要简单一些)
    TextureAxisFromPlane(plane, vecs[0], vecs[1]);

    if (!scale[0])
        scale[0] = 1;
    if (!scale[1])
        scale[1] = 1;

    // rotate axis
    if (rotate == 0)
        { sinv = 0 ; cosv = 1; }
    else if (rotate == 90)
        { sinv = 1 ; cosv = 0; }
    else if (rotate == 180)
        { sinv = 0 ; cosv = -1; }
    else if (rotate == 270)
        { sinv = -1 ; cosv = 0; }
    else
    {
        ang = rotate / 180 * Q_PI;
        sinv = sin(ang);
        cosv = cos(ang);
    }
```

```
    if (vecs[0][0])
        sv = 0;
    else if (vecs[0][1])
        sv = 1;
    else
        sv = 2;

    if (vecs[1][0])
        tv = 0;
    else if (vecs[1][1])
        tv = 1;
    else
        tv = 2;
/* 纹理坐标的变换。先将s,t轴绕s,t平面的法线旋转,注意这里可以将s,t当作x,y轴，则绕z轴转
    [s, t, 0, 0] * [cos,  sin, 0, 0]
                   [-sin, cos, 0, 0]
                   [0,      0, 1, 0]
                   [0,      0, 0, 1]*/
    for (i=0 ; i<2 ; i++) {
        ns = cosv * vecs[i][sv] - sinv * vecs[i][tv];
        nt = sinv * vecs[i][sv] +  cosv * vecs[i][tv];
        vecs[i][sv] = ns;
        vecs[i][tv] = nt;
    }
    //**乘上缩放值
    for (i=0 ; i<2 ; i++)
        for (j=0 ; j<3 ; j++)
            mappingVecs[i][j] = vecs[i][j] / scale[i];

    //**加上平移
    mappingVecs[0][3] = shift[0];
    mappingVecs[1][3] = shift[1];
}
```

映射的时候这样做：

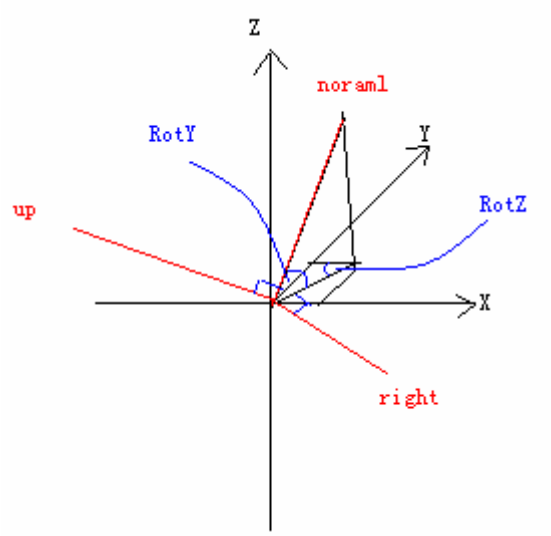
```
    // calculate texture s/t    s->vecs即上面提到的mappingVecs
    dv->st[0] = s->vecs[0][3] + DotProduct( s->vecs[0],dv->xyz );
    dv->st[1] = s->vecs[1][3] + DotProduct( s->vecs[1],dv->xyz );
    dv->st[0] /= si->width;
    dv->st[1] /= si->height;
```

1.2 还有一种新方法，是直接求平面的坐标基(Axis Base)。

这种方法最准确且简单易懂，目前我知道两种方法。一种是求 camera 坐标系类似的叉乘法方法(左手系：up * normal = right, normal * right = up，通常 up 先取(0,+-1,0)，除非 normal 的 y 分量最大，否则 up 取成其他，如(0,0,+-1)))。但总感觉这个方法不是太

好，因为需要事先找一个辅助的 up 轴。

另一种方法是 Q3 使用的。（这里感谢 GameRes 上的网友 jack33 对我的帮助）




```
// NOTE : ComputeAxisBase here and in editor code must always BE THE SAME !
// WARNING : special case behaviour of atan2(y,x) <-> atan(y/x) might not be the same everywhere when x == 0
// rotation by (0,RotY,RotZ) assigns X to normal
void ComputeAxisBase(vec3_t normal,vec3_t texX,vec3_t texY)
{
    vec_t RotY,RotZ;

    // compute the two rotations around Y and Z to rotate X to normal
    //**tan = -z / sqrt(x^2+y^2) 取名叫RotX才对!
    RotY = -atan2(normal[2], sqrt(normal[1]*normal[1] + normal[0]*normal[0]));
    //**tan = y / x
    RotZ = atan2(normal[1], normal[0]);

    // rotate (0,1,0) and (0,0,1) to compute texX and texY
    /*得到righ轴：将(0,1,0)先绕Z轴顺时针旋转(PI/2-RotZ)，即和normal投影线同向，然后再绕Z轴顺时针旋转度PI/2：其实最简单是直接将(1,0,0) 绕Z轴顺时针旋转(PI/2-RotZ)。
    [1,0 0] * [cos sin  0]
              [-sin cos 0] = [cos, sin, 0]
              [0    0   1]
    又顺时针是负值(右手系)：x = cos(-(PI/2-RotZ))= cos((PI/2-RotZ)) = sin(RotZ)
                            y = sin(-(PI/2-RotZ))= -sin((PI/2-RotZ)) = -cos(RotZ)
    但是这个right轴相对于可见平面来说，与纹理s轴反向。所以要取反*/
    texX[0]= -sin(RotZ);
    texX[1]= cos(RotZ);
    texX[2]= 0;
    // the texY vector is along -Z ( T texture coorinates axis )
```

```
/*得到up轴：这里是直接使用球面坐标系转直角坐标系的算法(球的公式)
其中RotZ是方位角，即球半径在XOY平面的投影线与X轴的夹角。
这里的天顶角＝PI/2-(PI/2-RotY)= RotY。同理，因为up实际上与t轴是反向的，所以也要的即反*/
texY[0]= -sin(RotY)*cos(RotZ);
texY[1]= -sin(RotY)*sin(RotZ);
texY[2]= -cos(RotY);
}
```

总结：可以考虑将 Q3 的方法与叉乘法结合。用 Q3 的方法求 right(这样就不用考虑选择辅助 up 轴)，然后直接用 right 与 noraml 叉乘得 up(这样就不用算那么多三角函数)。

以后这样变换三维顶点到纹理坐标：

```
// calculate texture s/t from brush primitive texture matrix
//**下面两个点积是求顶点坐标在轴上的投影，也就是说在轴平面上的坐标
x = DotProduct( dv->xyz, texX );
y = DotProduct( dv->xyz, texY );
//**然后按轴平面的变动跟着变动一次
dv->st[0]=s->texMat[0][0]*x+s->texMat[0][1]*y+s->texMat[0][2];
dv->st[1]=s->texMat[1][0]*x+s->texMat[1][1]*y+s->texMat[1][2];
```

注意上面 texMat[0][0]，texMat[0][1]，texMat[1][0]和 texMat [1][1]是关于纹理轴的缩放与旋转。texMat[0][2]与 texMat[1][2]是平移。
texMat 正好与下面的矩阵行列互换(转置)

$$[x,y] * \begin{bmatrix} sx, & 0, \\ 0, & sy, \\ 0, & 0 \end{bmatrix} * \begin{bmatrix} cos, & sin \\ -sin, & cos \\ 0, & 0 \end{bmatrix} + \begin{bmatrix} 0, & 0 \\ 0, & 0 \\ tx, & ty \end{bmatrix} = [x,y] * \begin{bmatrix} sx*cos, & sx*sin \\ -sy*sin, & sy*cos \\ tx, & ty \end{bmatrix}$$

即可以如下代替：

```
u = x*cosf(rotate)*scale[0] + y*(-sinf(rotate))*scale[1] + shift[0];
v = x*sinf(rotate)*scale[0] + y*(cosf(rotate))*scale[1] + shift[1];
```

其实无论哪种方法，它的本意都是先将 3 维顶点数据投影到平面上，然后再用纹理变换矩阵作用一次投影坐标。至于纹理变换(Texture Transform)就不多说了，在学习笔记 2 中有记录。

通常初始效果都不能让人满意(肯定的)，这时候需要在场景编辑器中手动调整纹理变换的各个参数(scale，rotate，shift/translate)。

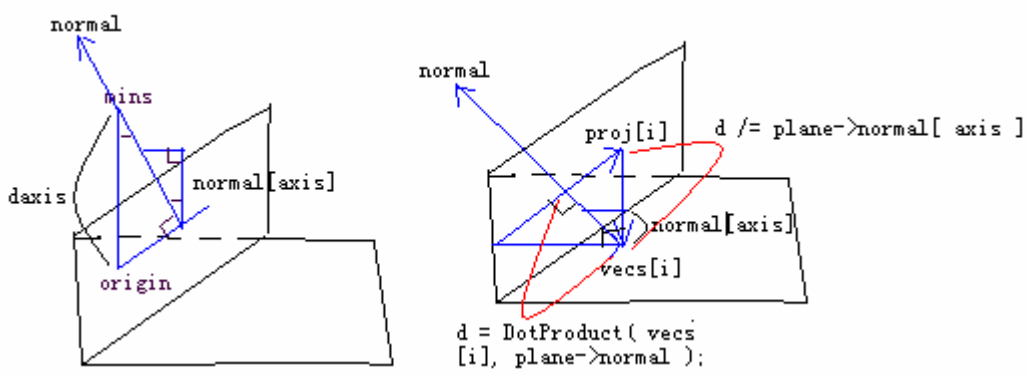
2．三维顶点到 lightmap 坐标的映射

映射到 Lightmap 与映射到纹理类似。

在“3.10 分配 lightmap 数据区：AllocateLightmaps”中已经介绍了如何映射曲面中的控制点到 lightmap 坐标，相对普通平面来说比较容易，也比较清晰：因为曲面的控制点是按行和列排序的，正好可以与二维的纹理坐标系对应上。

但对于普通平面来说，相对要麻烦一些。

（这里要感谢 CSDN 游戏开发讨论区的网友 IO_X 与 happy__888）



这样顶点映射到 lightmap sample：

```
// calculate the world coordinates of the lightmap samples
    // project mins onto plane to get origin
    d = DotProduct( mins, plane->normal ) - plane->dist;
    /*
    将mins点投影到原渲染平面上(多边形面)，注意这里是沿着最大轴(axis)投影
    即：d相当于从mins的位置出发沿着axis方向走，直到碰到plane时已走过的距离
    之所以沿axis方向投影，是因为lightmap平面取的是垂直于axis轴的平面，前面求st值的时候是在那个平面下求的
    cos = plane->normal[ axis ] / 1 = d / x
    x = d / plane->normal[ axis ] = "d相当于从mins的位置出发沿着axis方向走，直到碰到plane时已走过的距离"
    */
    d /= plane->normal[ axis ];
    VectorCopy( mins, origin );
    origin[axis] -= d;

    // project stepped lightmap blocks and subtract to get planevecs
    /*求坐标基(类似于仿射坐标系)。这里实际上是将s,t轴投影到原渲染平面上
    投影的要求是：“mins点与各点”在lm平面上的投影点之间的距离 等于 mins点在原平面上的投影点与各点之间的距离。这样才能保证逆变换是正确的(lm坐标＝＞空间坐标)
    所以和上面一样，也是沿着最大轴(axis)投影。注意这次不是点而是矢量，所以用了矢量运算。
    */
    for ( i = 0 ; i < 2 ; i++ ) {
```

```
        vec3_t    normalized;
        float     len;

        len = VectorNormalize( vecs[i], normalized );
        //**len是1.0/ssize，所除以len就是乘以ssize，即还原原来的尺寸
        VectorScale( normalized, (1.0/len), vecs[i] );
        /*由于vecs[i]与法线的夹角大于90度，所以最后d值为负。变成矢量时要注意，例如Z轴最大的话，则为(0,0,d)，但d是负的，所以向下，所以vecs[i][axis] -= d之后得到的矢量即投影矢量，也即直角三角形的斜边*/
        d = DotProduct( vecs[i], plane->normal );
        d /= plane->normal[ axis ];
        vecs[i][axis] -= d;
        }

VectorCopy( origin, ds->lightmapOrigin );
VectorCopy( vecs[0], ds->lightmapVecs[0] ); //**s
VectorCopy( vecs[1], ds->lightmapVecs[1] ); //**t
VectorCopy( plane->normal, ds->lightmapVecs[2] );//**normal
```

这样 lightmap sample 映射到顶点：

```
for ( k = 0 ; k < 3 ; k++ ) {
/*lightmapOrigin是此平面的原点，而lightmapVecs[0]和lightmapVecs[1]是平面的坐标基。因此lightmapOrigin + i*lightmapVecs[0] + j*lightmapVecs[1] 即lightmap某smaple在世界坐标系中的坐标值。*/
base[k] = lightmapOrigin[k] + normal[k] + i * lightmapVecs[0][k]
                                        + j * lightmapVecs[1][k];
}
```

其实这样做也许更好（不使用中间平面）：

(1)直接计算原平面的坐标基(s,t)

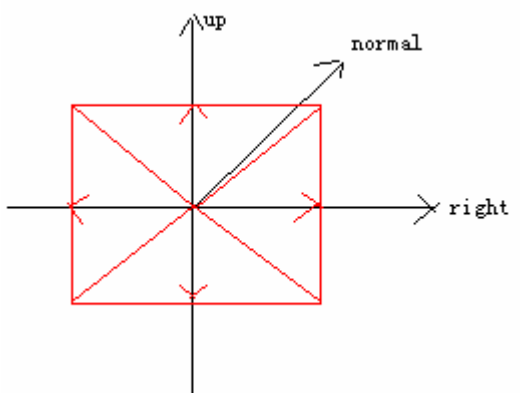
（2）求 mins

(3)求各顶点在 lightmap 坐标系下的坐标

（4）求 mins 在平面上的投影点坐标(原点)

(网上有一篇计算 lightmap 的文章(Light Mapping - Theory and Implementation)，大家可以参考)

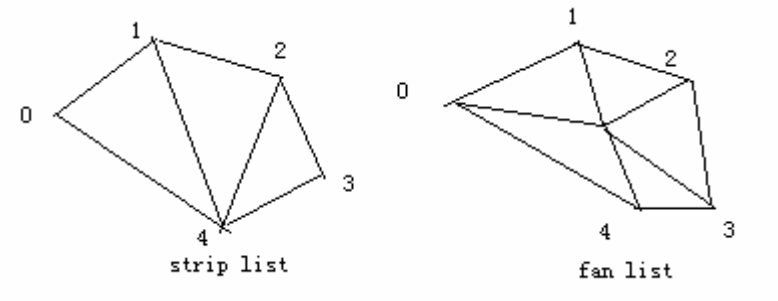
3．为平面创建 Winding：BaseWindingForPlane




```
//**用来在一个平面上创建代表此平面的4个点(例如：可以直观地观察平面)
winding_t *BaseWindingForPlane (vec3_t normal, vec_t dist)
{
    int       i, x;
    vec_t     max, v;
    vec3_t    org, vright, vup;
    winding_t *w;

// find the major axis

    max = -BOGUS_RANGE;
    x = -1;
    for (i=0 ; i<3; i++)
    {
        v = fabs(normal[i]);
        if (v > max)
        {
            x = i;
            max = v;
        }
    }
    if (x==-1)
        Error ("BaseWindingForPlane: no axis found");

    VectorCopy (vec3_origin, vup);
    switch (x)
    {
    case 0:
    case 1:
        vup[2] = 1;//**z
        break;
    case 2:
```

```
            vup[0] = 1;//**x
            break;
        }

        /*实际上算法是很简单的，道理与获得camera的轴look, up, right是一样的*/
        v = DotProduct (vup, normal);
        VectorMA (vup, -v, normal, vup);//** vup = vup + -v*normal
        VectorNormalize (vup, vup);

        VectorScale (normal, dist, org);//**org = normal*dist

        CrossProduct (vup, normal, vright);

        //**沿着up与right方向扩展
        VectorScale (vup, MAX_WORLD_COORD, vup);
        VectorScale (vright, MAX_WORLD_COORD, vright);

// project a really bigaxis aligned box onto the plane
        w = AllocWinding (4);

        //**找到四个点(矩形的四个顶点)
        //**注意还要加上偏移，因为平面距原点是有一定距离的
        //**其实先将它们在过原点的平面上扩展，最后再加上偏移更直观一些
        VectorSubtract (org, vright, w->p[0]);
        VectorAdd (w->p[0], vup, w->p[0]);

        VectorAdd (org, vright, w->p[1]);
        VectorAdd (w->p[1], vup, w->p[1]);

        VectorAdd (org, vright, w->p[2]);
        VectorSubtract (w->p[2], vup, w->p[2]);

        VectorSubtract (org, vright, w->p[3]);
        VectorSubtract (w->p[3], vup, w->p[3]);

        w->numpoints = 4;

        return w;
}
```

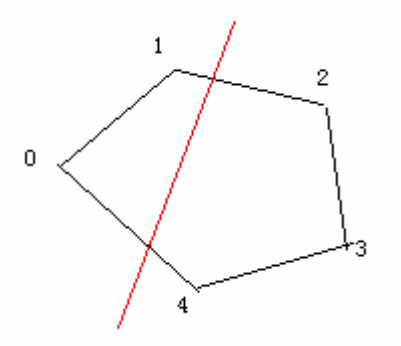
4. 凸多边形拆分成三角形 List： SurfaceAsTristrip




```
================
SurfaceAsTristrip

Try to create indices that make (points-2) triangles in tristrip order
================
#define  MAX_INDICES   1024
static void SurfaceAsTristrip( dsurface_t *ds ) {
    int                 i;
    int                 rotate;
    int                 numIndices;
    int                 ni;
    int                 a, b, c;
    int                 indices[MAX_INDICES];
    // determine the triangle strip order
    numIndices = ( ds->numVerts - 2 ) * 3;//**numIndices = (numVerts-2) * 3
    if ( numIndices > MAX_INDICES ) {
        Error( "MAX_INDICES exceeded for surface" );
    }
    // try all possible orderings of the points looking for a strip order that isn't degenerate
    /*这个算法的意思是，从顶点0开始，把每个顶点当作一次基准点，以此基准点考虑其他点。
    它构成三角形的方法类似于D3D的Triangle Strips，一点一点地增加
    如果此基准点下正好构成了完整的三角形表，则合格。
    否则考虑下一个基准点。而之前构成的三角形无效(被重复填充掉)
    */
    for ( rotate = 0 ; rotate < ds->numVerts ; rotate++ ) {//**取基准点
        for ( ni = 0, i = 0 ; i < ds->numVerts - 2 - i ; i++ ) {//**以此基准点考虑其他点
            //**注意到上面每次的ni = 0

            a = ( ds->numVerts - 1 - i + rotate ) % ds->numVerts;
            b = ( i + rotate ) % ds->numVerts;
            c = ( ds->numVerts - 2 - i + rotate ) % ds->numVerts;
            if ( IsTriangleDegenerate( drawVerts + ds->firstVert, a, b, c ) ) {
                break;
            }
            indices[ni++] = a;
```

```
            indices[ni++] = b;
            indices[ni++] = c;

            if ( i + 1 != ds->numVerts - 1 - i ) {//**strip
                a = ( ds->numVerts - 2 - i + rotate ) % ds->numVerts;
                b = ( i + rotate ) % ds->numVerts;
                c = ( i + 1 + rotate ) % ds->numVerts;

                if ( IsTriangleDegenerate( drawVerts + ds->firstVert, a, b, c ) ) {
                    break;
                }
                indices[ni++] = a;
                indices[ni++] = b;
                indices[ni++] = c;
            }
        }
        if ( ni == numIndices ) { //**恰好完成三角形组合
            break;        // got it done without degenerate triangles
        }
    }

    // if any triangle in the strip is degenerate,
    // render from a centered fan point instead
    if ( ni < numIndices ) {//**没有完成，就重新组合成Triangle Fans
        c_fanSurfaces++;
        SurfaceAsTriFan( ds );
        return;
    }

    // a normal tristrip
    c_stripSurfaces++;

    if ( numDrawIndexes + ni > MAX_MAP_DRAW_INDEXES ) {
        Error( "MAX_MAP_DRAW_INDEXES" );
    }
    ds->firstIndex = numDrawIndexes;
    ds->numIndexes = ni;

    //**copy to tatal index buffer
    memcpy( drawIndexes + numDrawIndexes, indices, ni * sizeof(int) );
    numDrawIndexes += ni;
}
```

5. 分割任意凸多边形：ClipWindingEpsilon




```
void ClipWindingEpsilon (winding_t *in, vec3_t normal, vec_t dist,
                         vec_t epsilon, winding_t **front, winding_t **back)
{
    vec_t    dists[MAX_POINTS_ON_WINDING+4];
    int      sides[MAX_POINTS_ON_WINDING+4];
    int      counts[3];
    static   vec_t    dot;      // VC 4.2 optimizer bug if not static
    int      i, j;
    vec_t    *p1, *p2;//**old set is vec_t
    vec3_t   mid;
    winding_t*f, *b;
    int      maxpts;

    counts[0] = counts[1] = counts[2] = 0;//**记录三种情况下包含的点
// determine sides for each point
    for (i=0 ; i<in->numpoints ; i++)
    {
        dot = DotProduct (in->p[i], normal);
        dot -= dist;
        dists[i] = dot;

        if (dot > epsilon)
            sides[i] = SIDE_FRONT;
        else if (dot < -epsilon)
            sides[i] = SIDE_BACK;
        else
        {
            sides[i] = SIDE_ON;
        }
        counts[sides[i]]++;
    }

    //**保存第一个点的相关值在最后一个点相关值的后面，方便后面循环取得
    sides[i] = sides[0];
```

```
dists[i] = dists[0];

*front = *back = NULL;

if (!counts[0])//**SIDE_FRONT=0
{
    *back = CopyWinding (in);
    return;
}
if (!counts[1])//**SIDE_BACK=1
{
    *front = CopyWinding (in);
    return;
}

//**下面是分割的情况
//**这里提供了一个阈值 也就是说分割后产生的点的个数不能超过 (原有点+4)*2
maxpts = in->numpoints+4;   // cant use counts[0]+2 because of fp grouping errors

*front = f = AllocWinding (maxpts);
*back = b = AllocWinding (maxpts);

for (i=0 ; i<in->numpoints ; i++)
{
    p1 = in->p[i];

    //**渲染q3 bsp时，对于共面的情况既可以当作前面，也能当作后面。
    if (sides[i] == SIDE_ON)
    {
        VectorCopy (p1, f->p[f->numpoints]);
        f->numpoints++;
        VectorCopy (p1, b->p[b->numpoints]);
        b->numpoints++;
        continue;
    }

    if (sides[i] == SIDE_FRONT)
    {
        VectorCopy (p1, f->p[f->numpoints]);
        f->numpoints++;
    }
    if (sides[i] == SIDE_BACK)
    {
        VectorCopy (p1, b->p[b->numpoints]);
```

```
            b->numpoints++;
        }

        //**看下一个点是否和此点“相反”
        if (sides[i+1] == SIDE_ON || sides[i+1] == sides[i])
            continue;

    //**如果“相反”，就要在它们之前产生一个新的分割点
    // generate a split point
        p2 = in->p[(i+1)%in->numpoints];//**%是因为最后一个点的下一个点是第一个点

        //**这里求交没有用射线与平面求交的方法，而是直接用了前面算出来的距离来做比值
        dot = dists[i] / (dists[i]-dists[i+1]);//**这里巧妙地利用了正负关系
        for (j=0 ; j<3 ; j++)
        {    // avoid round off error when possible//**什么错误？
            if (normal[j] == 1)
                mid[j] = dist;
            else if (normal[j] == -1)
                mid[j] = -dist;
            else
                mid[j] = p1[j] + dot*(p2[j]-p1[j]);
        }

        //**交点被分配到两边
        VectorCopy (mid, f->p[f->numpoints]);
        f->numpoints++;
        VectorCopy (mid, b->p[b->numpoints]);
        b->numpoints++;
    }

    //**注意这里的错误处理也很重要
    if (f->numpoints > maxpts || b->numpoints > maxpts)
        Error ("ClipWinding: points exceeded estimate");
    if (f->numpoints > MAX_POINTS_ON_WINDING || b->numpoints > MAX_POINTS_ON_WINDING)
        Error ("ClipWinding: MAX_POINTS_ON_WINDING");
}
```

5．动态分配固定数组(名字不太准确)

```
winding_t *NewWinding (int points)
{
/* typedef struct
    {
        int      numpoints;
        vec3_t   points[MAX_POINTS_ON_FIXED_WINDING];          // variable sized
    } winding_t; */

    winding_t      *w;
    int          size;

    if (points > MAX_POINTS_ON_WINDING)
        Error ("NewWinding: %i points", points);
    /*
    下面这个是一个技巧：
    ((winding_t *)0)->points[points]得到的是winding_t::points[points]的地址(注意points成员是
float[3]型所以points是二维数组)。由于起始地址是0，所以正好是这个结构变量的大小 */
    size = (int)((winding_t *)0)->points[points];
    w = malloc (size);
    memset (w, 0, size);

    return w;
}
```

最后感谢你对这份报告的关注。

cywater2000　2006-3-28