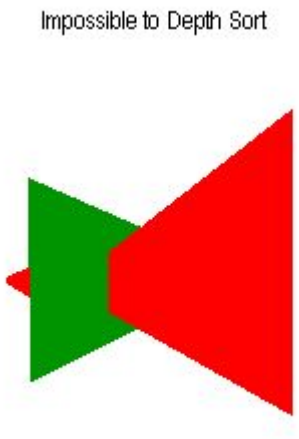
图形渲染引擎要解决的基本问题：

以正确的显示顺序来渲染多边形，并能够处理半透明物体的渲染问题，具体来讲远方的物体需要先绘制，近处的物体需要后绘制，这样才能模拟现实中近处物体遮挡远处物体的现象，并且半透明物体需要在不透明物体之后绘制，因为半透明物体的实质是进行了 alpha blending。

当我们使用一些高层次的图形渲染软件时，仅仅只是提交了多边形网格，并设置了渲染状态、相机等。软件在后台却做了很多处理，首先进行坐标系的变换，从模型坐标系变换到世界坐标系，再变换到相机坐标系，然后进行了背面多边形的剔除以及相机视景体的裁剪，之后它会区分透明和不透明物体，先对不透明多边形基于相机深度进行排序由远到近的渲染多边形网格（或者无须排序而是直接使用一个 Z-Buffer，一般来说有硬件加速），再对半透明物体进行融合计算，因为这样的计算会随着相机移动在每帧中都进行，所以会带来性能。

此外多边形排序无法解决下面情形



利用 BSP 进行图形渲染：

BSP 解决了哪些问题，经典 BSP 树解决了多边形按照正确顺序渲染的问题，并实现了背面剔除（依赖递归遍历二叉树实现实现多边形有序）；Solid Node BSP 树是对经典 BSP 树的改进，增加了碰撞检测功能；Quake3 中使用的是 Solid Leaf BSP 树，因为它将要渲染的多边形都存放在了叶节点中，它用了跟经典 BSP 树完全不同的渲染策略，将场景预先划分为一些凸体区域，并且为这些凸体之间的可见性（PVS：Potential Visibility Set）做了预先计算，这样就不必渲染整个场景了，而是仅仅渲染那些可见的凸体区域的多边形而已，当然不像经典 BSP 树可以实现多边形的有序遍历，所以它会依赖 Z-Buffer，并且利用它可以高效的进行视景体剔除和背面剔除，同时还可以进行实时的碰撞检测。

经典 BSP 树的构造

从多边形列表中选取一个多边形作为分割面，将所有其他多边形分为 front list 和 back list，如果有多边形跟分割面相交的，将其分为两个多边形放在相应 list 中，而用作分割面的多边形以及在分割面上的其他多边形则放在当前节点中，递归处理 front list 和 back list，并将返回值作为当前节点的左右子节点，直到所有多边形都成为了分割面。

经典 BSP 树构建时如何选取合适的分割面

1. 分割面相交的多边形数量尽量少
2. 位于分割面两边的多边形数量尽量一致，即构造的二叉树尽量平衡

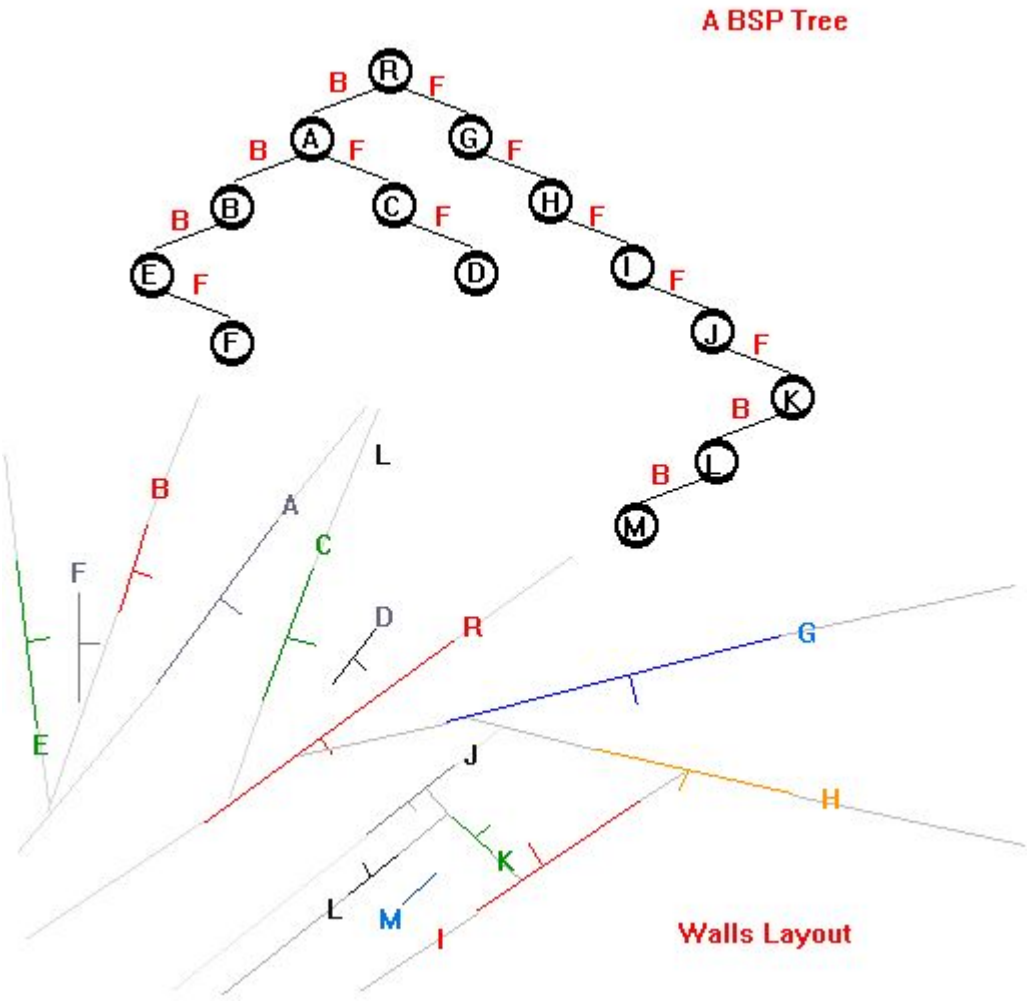
分割面打分公式

```
Score = abs(frontfaces-backfaces)*w1+splits*w2
```

构造最优 BSP 树是一个艰难的任务（因为当前分割面产生新的被分割多边形会对后续产生影响），所以一般来说我们构造的 BSP 树只是较优的，仅仅根据上面的打分公式来选取局部最优分割面。

经典 BSP 树

注：下图中左节点表示 back，右节点表示 front



经典 BSP 树构建算法伪代码

```
Node BuildBSP(Polygon[] polygons, Node root):
    // 选取局部最高打分多边形作为分割面多边形
    Splitter = PickupPolygon(polygons)//选取并移除多边形
    Root.splitter = Splitter
    Polygon[] frontlist
    Polygon[] backlist
```

```
    Polygon frontpart, backpart
    Foreach polygon in polygons:
        Switch(WhichSide(polygon, splitter):
            Case Front:
                frontlist.Add(polygon)
            Case Back:
                backlist.Add(polygon)
            Case Span:
                SplitPolygon(polygon, splitter, frontpart, backpart)
                frontlist.Add(frontpart)
                backlist.Add(backpart)
            Case OnPlane:
                If(IsSameFacing(splitter, polygon)):
                    root.sameFacing.Add(polygon)
                Else
                    root.oppositeFacing.Add(polygon)
    Root.front = BuildBSP(frontlist, new Node())
    Root.back = BuildBSP(backlist, new Node())
    Return root
```

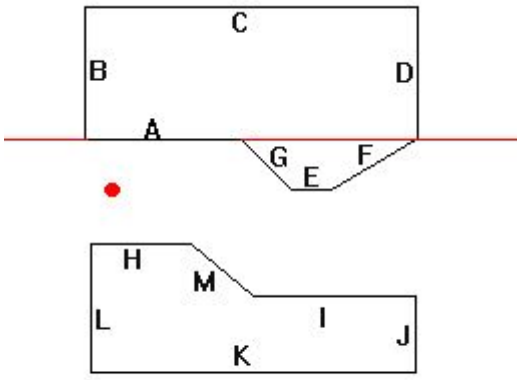
经典 BSP 树渲染伪代码

```
RenderBSP(Node current, Camera camera):
    if(current == null):
        Return
    Side = WhichSide(camera.position, current.splitter)
    If(Side == Front):
        // 相机在分割面的前面，则先渲染分割面后面的多边形（距相机远）
        If(current.back != null):
            RenderBSP(current.back, camera)
        // 再渲染在分割面上的多边形
        DrawPolygon(current.splitter)
        DrawPolygons(current.sameFacing)
        // 最后渲染在分割面前面的多边形（距相机近）
        If(current.front != null):
            RenderBSP(current.front, camera)
    Else If(Side == Back):
        // 相机在分割面的后面，则先渲染分割面前面的多边形（距相机远）
        If(current.front != null):
            RenderBSP(current.front, camera)
        //DrawPolygon(current.splitter)注意：背面剔除，无须渲染它
        // 在渲染在分割面上的多边形
        DrawPolygons(current.oppositeFacing)
        // 最后渲染在分割面后面的多边形
        If(current.back != null):
```

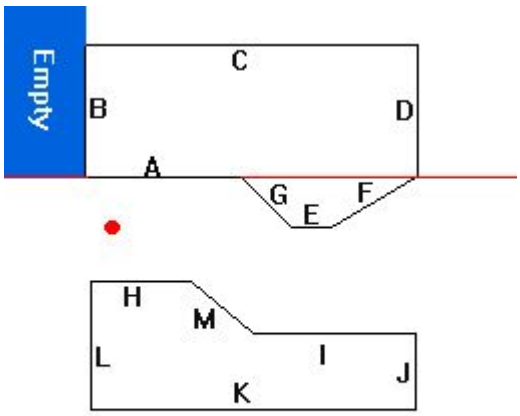
RenderBSP(current.back, camera)

Solid Node BSP 树的构造
同经典 BSP 树的不同点
1. 多边形跟作为分割面的多边形在同一平面上时，将该多边形传入 front list
2. 为节点增加 IsLeaf 和 IsSolid 分别用来标识叶节点和叶节点是否为 solid

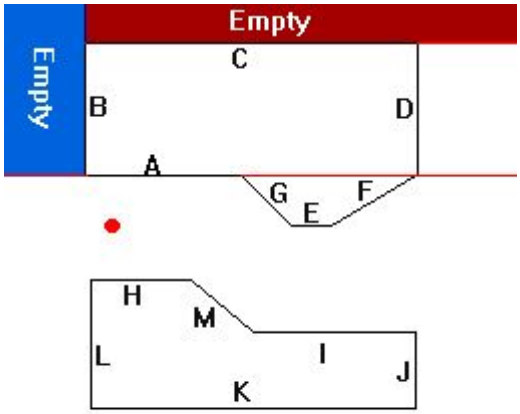
Solid Node BSP 树构建样例
红点是 Player 位置，围起来的可以看成是墙或者柱子之类的东西，它们的里面就是所谓的 solid space，别的地方被称为 empty space



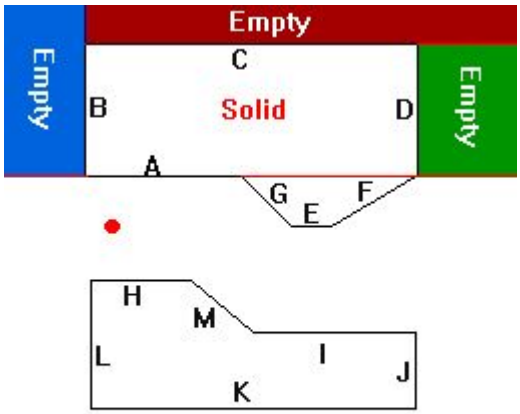
A 作为分割面



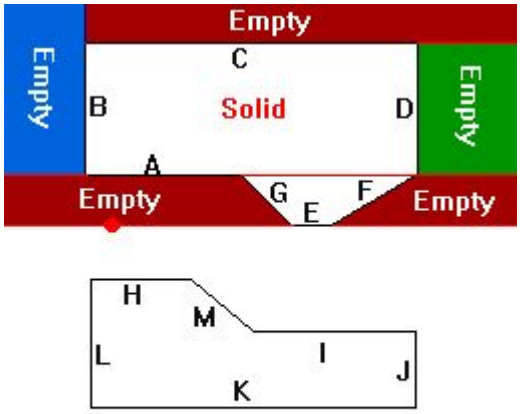
开始处理 A 的 back list，B 作为分割面



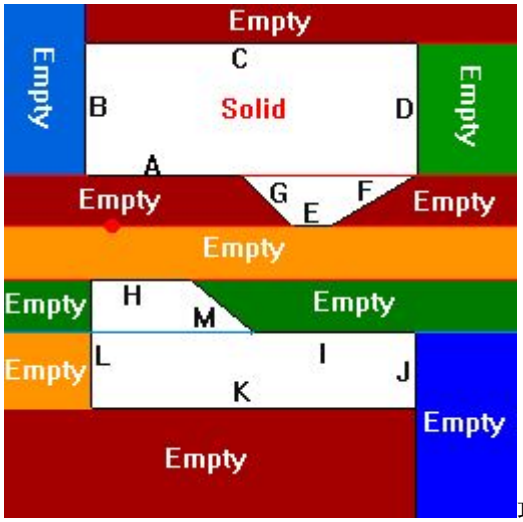
C 作为分割面



D 作为分割面，分割面包围而成的凸体



开始处理 A 的 front list，依次 E,F,G 又构造了凸体 AEFG



最终效果图

Solid Node BSP 树碰撞检测

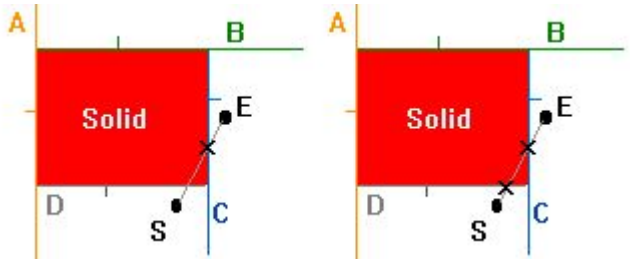
如果我们想要测试，Player 的位置是否走进了墙内，不使用 BSP 树时，我们要跟场景内所有的多边形进行测试，但现在呢？

我们只要从 BSP 根节点，进行简单的点在分割面哪边的测试，最终 Player 位置会落在一个叶节点中的，然后我们只需要测试当前叶节点是否是 solid 就可以知道用户是否走到了墙内

但其实真正的碰撞检测不是这样做的，上面的做法仅仅测试了一个点在场景的 empty 还是 solid 中而已，真实的碰撞检测样例是使用位移起点和终点来做的（准确来讲是一段预测的运动轨迹），并且会附带移动物体的包围盒，但跟确定 Player 位置一样也要对 BSP 树进行类似的搜索

线段的碰撞检测

在进行真正的碰撞处理之前，先让我们来处理简单的情形，检测起点到终点位置是否有障碍物会发生碰撞，而先不去考虑移动物体包围盒的问题



从 S 移动到 E

线段碰撞检测伪代码

```
Boolean CanMoveTo(Vector3 start, Vector3 end, Node current):
    If(current.IsLeaf == true):
        Return !current.IsSolid
    sideS = GetSide(start, current.splitter)
    sideE = GetSide(end, current.splitter)
    Intersection = GetIntersection(start, end, current)
    If(sideS == Front && sideE == Back):
```

```
    Return CanMoveTo(start, Intersection, current.front) &&
canMoveTo(Intersection, end, current.back)
Else If(sideS == back && sideE == front):
    Return CanMoveTo(end, Intersection, current.front) &&
CanMoveTo(Intersection, start, current.back)
Else If(sideS == OnPlane && sideE == OnPlane):
    Return CanMoveTo(start, end, current.front)
Else If(sideS == Front || sideE == Front):
    Return CanMoveTo(start, end, current.front)
Else
    Return CanMoveTo(start, end, current.back)
```

该函数只是返回了从 start 到 end 是否能够行走，跟我们在 Quake3 中使用的 BSP 碰撞算法还是有差别的，因为我们不仅仅希望可以知道是否碰撞了，还希望知道在哪个点发生的碰撞。

Leaf BSP 树的构建

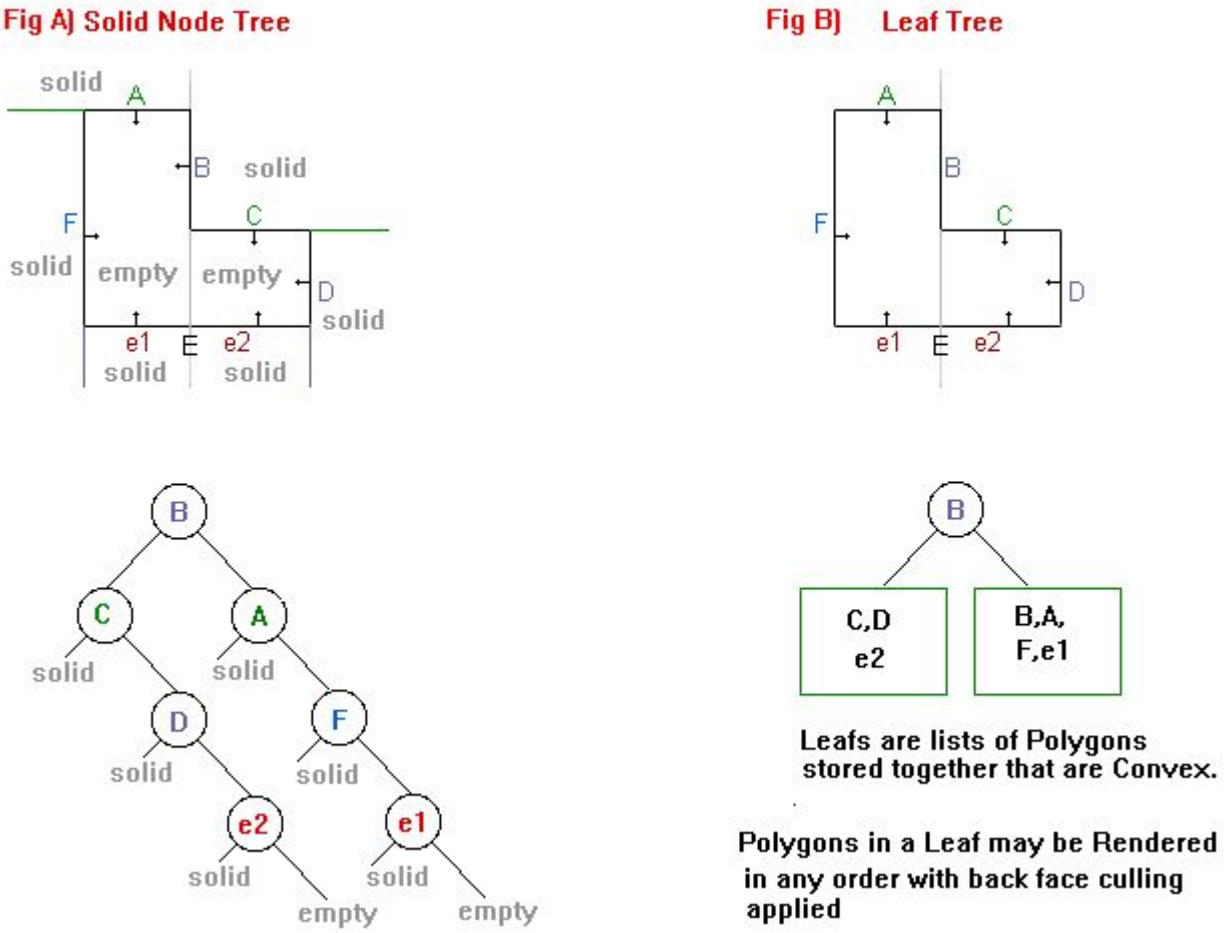
从多边形列表中选取一个多边形，以该多边形所在平面作为分割面，和经典 BSP 树不同，Leaf BSP 树仅仅会在节点中存放该多边形所在的平面而不是该多边形自身，并且会将该多边形传入到 front list 进行递归调用（也就是说该多边形在后续的步骤中可能会被分割为新的多边形）；同时被选为分割面的多边形会对其进行标记，因为 Leaf BSP 树中是不允许被选为分割面的多边形在之后又被选为多边形的，并且这种属性会被继承，即新产生的分割多边形会被标记为被分割多边形相同的属性，也即如果一个多边形被选为分割面，则后续对该多边形的分割产生的新的多边形也是不可以被选为分割面的；依照上述规则依次对多边形列表进行递归调用，检测到当前的多边形列表中所有的多边形都在彼此的前面，也即构成了一个凸体空间（不管是 front list 或 back list，也不管多边形有没有选为分割面），此时创建叶节点，Leaf BSP 树的构建完成。

渲染 Leaf BSP 树，只需要根据相机位置搜索 Leaf BSP 树，直到到达相应的叶节点（凸体区域），然后先渲染父节点的另一个叶节点，再渲染当前的叶节点（渲染时可以进行背面剔除）。

Leaf BSP 树的缺陷是，没有 solid，empty 标识，无法进行碰撞检测，可以看到 Leaf BSP 树和经典 BSP 树的最大区别在于，Leaf BSP 树将所有要渲染的多边形都放在了叶节点中，而经典 BSP 树则将所有要渲染的多边形作为普通节点的分割面多边形存放。Leaf BSP 将多边形放在叶节点的好处在于，将场景从多边形层次提升到了叶节点定义的凸体区域层次，如果对 Leaf BSP 进行改进，使其具有叶节点相互之间是否可见（可视）的信息，那么可以根据相机位置大量的剔除凸体区域。

Solid Node BSP 树和 Leaf BSP 树的区别样例

注：下图中左节点表示 back，右节点表示 front



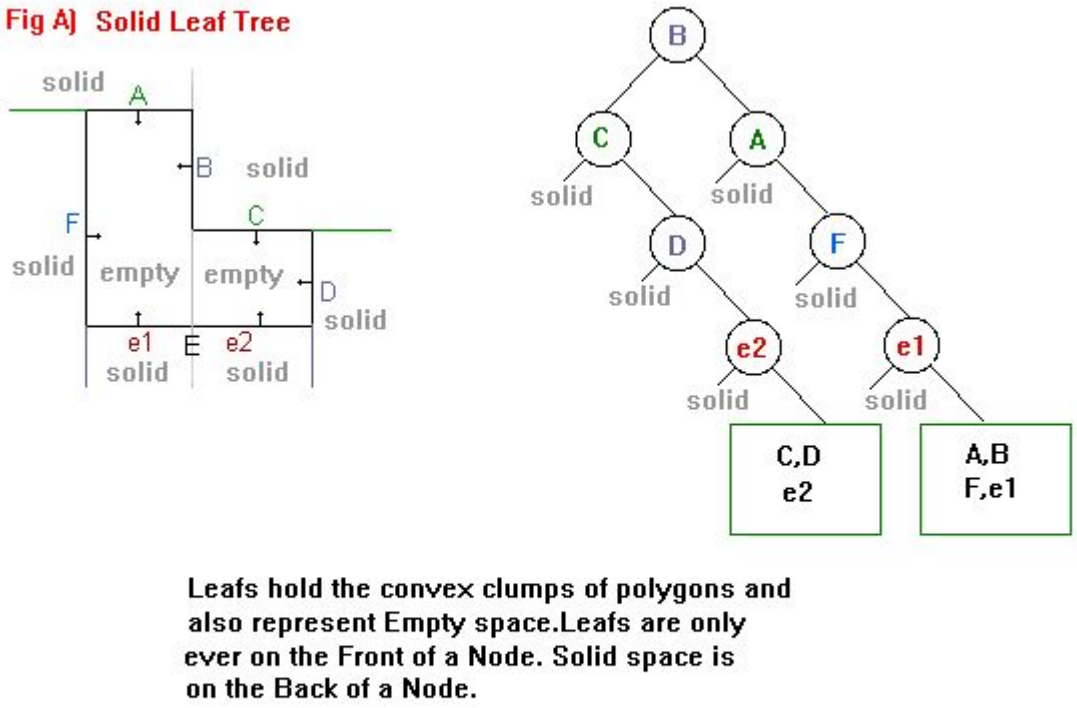
Solid Leaf BSP 树

Solid Node Leaf 中将多边形储存在普通节点中，并且提供了 solid、empty 机制进行碰撞检测，而 Leaf BSP 树则将所有的待渲染多边形存放在了叶节点中，可是没有提供碰撞检测功能，Solid Leaf BSP 树要做的事情就是将 Solid Node BSP 树和 Leaf BSP 树的优点结合起来。

构建 Solid Leaf BSP 树的过程，从多边形列表中选取一个多边形，以该多边形所在平面作为分割面，将分割面存放在节点中并予以标记（保证后面不会再次被选为分割面多边形），根据分割面将多边形列表分为 front list 和 back list，如果多边形在分割面上，则根据多边形和分割面朝向是否一致分别将其放在 front list 和 back list 中，并递归的处理 front list 和 back list。对 front list 的处理，直到 front list 中的所有多边形都已被选为分割面时（凸体出现），创建一个叶节点，对应 empty space，存放 front list 中的所有多边形，将该叶节点作为父节点的 front 子节点。对 back list 的处理，直到 back list 中没有任何多边形后，将父节点的 back 节点标记为-1，对应 solid space。（注：当所有的多边形作为分割面后说明多边形在彼此的前面）

Solid Leaf BSP 树构建图解

注：下图中左节点表示 back，右节点表示 front



Leafs hold the convex clumps of polygons and also represent Empty space.Leafs are only ever on the Front of a Node. Solid space is on the Back of a Node.

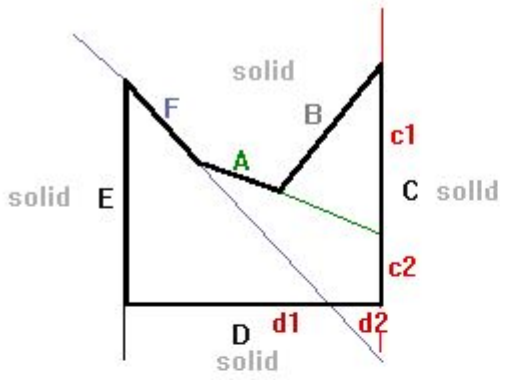
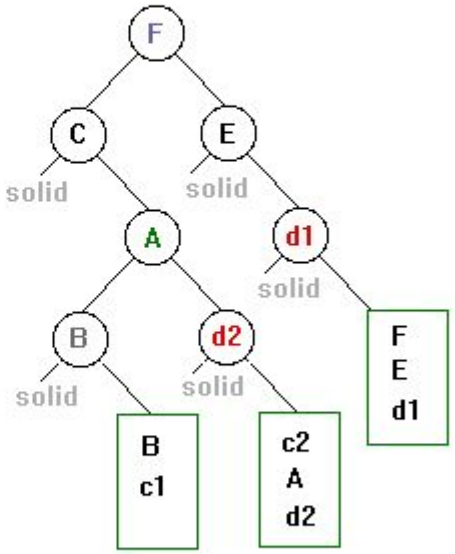


Fig D)

Polygon C gets split even though it has been used as a split plane



Solid Leaf BSP 树渲染

Solid Leaf BSP 采用了完全不同于经典 BSP 树的渲染策略，不再依赖遍历二叉树来使多边形有序渲染，而是利用在编译 Solid Leaf BSP 文件是生成的 PVS，PVS 简单可以理解为每个叶节点会记录从自己的区域看出去其它所有的叶节点是否可以被看到（注意这里和相机朝向没关系，在这里只要从任何位置可以看到别的叶节点就是可见的），也就是说整个 Solid Leaf BSP 树的 PVS 可以看成是一个 NumberOfLeaf*NumberOfLeaf 大小的位数组，用来记录所有叶节点之间的相互可见性，要注意的是 PVS 数据是在编译 BSP 文件时生成的，也就是说运行时只需要载入 PVS 进行快速的位数组测试即可。（其实文件中的 PVS 数据使用了某种压缩策略，并不是一个 NumberOfLeaf*NumberOfLeaf 大小的位数组）

Solid Leaf BSP 采用了和经典 BSP 树完全不同的渲染策略，根据相机位置搜索 Solid Leaf BSP 树，直到到达相应的叶节点（相机一般不会在 solid 中的吧），然后通过 PVS 来检测所有的叶节点，记录其中可见的叶节点，最后渲染所有可见的叶节点中的多边形列表，因为这些叶节点是相机深度无序的，所以需要将图形引擎的 Z-Buffer 打开。

在 Quake3 中的渲染在做了 PVS 测试后，还可以对叶节点进行视景体剔除以及背面剔除，所以可以看到通过使用 Solid Leaf BSP 树，要渲染的多边形呈对数级别下降。

Solid Leaf BSP 渲染伪代码

```
RenderBSP(Node root, Camera camera):
    If(root.IsSolid):
        // 在 solid space 中
        Return
    If(root.IsLeaf):
        // 在 empty space 中，渲染所有可见 leaf
        Foreach(lf in allleafs):
            // 获取可见 leaf，这里只为说明 pvs 作用
            // 真正的 pvs 测试不是这样的
            If(pvs[root.index][lf.index] == 1)
                Foreach(polygon in lf.polygons):
                    DrawPolygon(polygon)  // 开启了 Z-Buffer
    Side = WhichSide(root.plane, camera.position)
    If(side == front):
        RenderBSP(root.front, camera)
    Else
        RenderBSP(root.back, camera)
```

Solid Leaf BSP 简单碰撞伪代码

```
Boolean CanMoveTo(Node current, Vector3 start, Vector3 end):
    plane = planes[current.PlaneIndex]
    sideS = WhichSide(start, plane)
    sideE = WhichSide(end, plane)
    intersection = new Vector3()
    // 起点和终点都在分割面的前面
    If(sideS == Front && sideE == Front):
        If(current.IsLeaf == false):
            // 当前节点不是叶节点，所以继续递归
            Return CanMoveTo(current.front, start, end)
        Else
            // 当前是叶节点，则移动线段位于 empty space 内
            Return true
    // 起点在分割面前，终点在分割面后
    Else If(sideS == Front && sideE == Back):
        // 如果终点已经在 solid space 空间内，说明不能移动到这里
        If(current.back == -1):
            return false
        GetIntersection(start, end, plane, intersection)
        // 分别求子线段在各自子树中是否可到达
```

```
    If(current.IsLeaf == false):
        Return CanMoveTo(current.front, start, intersection) &&
               CanMoveTo(current.back, intersection, end)
    Else
        Return true &&
               CanMoveTo(current.back, intersection, end)
// 起点在分割面后，终点在分割面前
Else If(sideS == Back && sideE == Front):
    // 如果起点已经在 solid space 空间内，说明不能移动到这里
    If(current.back == -1):
        Return false
    GetIntersection(start, end, plane, intersection)
    // 分别求子线段在各自子树中是否可到达
    If(current.IsLeaf == false):
        Return CanMoveTo(current.front, end, intersection) &&
               CanMoveTo(current.back, intersection, start)
    Else
        Return true &&
               CanMoveTo(current.back, intersection, start)
// 如果起点和终点在分割面的同一侧，则递归计算
If(sideS == Front || sideE == Front):
    If(current.IsLeaf == false):
        Returen CanMoveTo(current.front, start, end)
    Else If(current.back == -1):
        Return false
    Else
        Return CanMoveTo(current.back, start, end)
Return true
```

Quake3 BSP 文件要点

Enitties 中以字符串的格式，包含了游戏场景相关的一些信息，比如环境光的大小，高亮点的位置，Player 模型在场景中的放置位置等信息

Node 中的 Bounding Box 包围了整个左右子树的多边形，仅仅是在 BSP 文件编译期间用于 PVS 计算的，在游戏运行时并没有任何用处

Leaf 中的 Bounding Box 包围了叶节点中的多边形，在运行时可以用于视景体的剔除。此外 Leaf 中包含了要绘制的三角形列表（faces）以及进行碰撞检测的 Brush。

Texture 中除了记录使用的图片名字外，还有 Surface flags 和 Content flags，其中 Surface flags 用于表明纹理图片的性质参数，Content flags 用于表明 Brush 所包围的内容

Name	Value	Notes
SURF_LIGHT	0x1	value will hold the light strength
SURF_SKY2D	0x2	don't draw, indicates we should skylight + draw 2d sky but not draw the 3D skybox
SURF_SKY	0x4	don't draw, but add to skybox
SURF_WARP	0x8	turbulent water warp
SURF_TRANS	0x10	
SURF_NOPORTAL	0x20	the surface can not have a portal placed on it
SURF_TRIGGER	0x40	FIXME: This is an xbox hack to work around elimination of trigger surfaces, which breaks occluders
SURF_NODRAW	0x80	don't bother referencing the texture
SURF_HINT	0x100	make a primary bsp splitter
SURF_SKIP	0x200	completely ignore, allowing non-closed brushes
SURF_NOLIGHT	0x400	Don't calculate light
SURF_BUMPLIGHT	0x800	calculate three lightmaps for the surface for bumpmapping
SURF_NOSHADOWS	0x1000	Don't receive shadows
SURF_NODECALS	0x2000	Don't receive decals
SURF_NOCHOP	0x4000	Don't subdivide patches on this surface
SURF_HITBOX	0x8000	surface is part of a hitbox

Name	Value	Notes
CONTENTS_EMPTY	0	No contents
CONTENTS_SOLID	0x1	an eye is never valid in a solid
CONTENTS_WINDOW	0x2	translucent, but not watery (glass)
CONTENTS_AUX	0x4	
CONTENTS_GRATE	0x8	alpha-tested "grate" textures. Bullets/sight pass through, but solids don't
CONTENTS_SLIME	0x10	
CONTENTS_WATER	0x20	
CONTENTS_MIST	0x40	
CONTENTS_OPAQUE	0x80	block AI line of sight
CONTENTS_TESTFOGVOLUME	0x100	things that cannot be seen through (may be non-solid though)
CONTENTS_UNUSED	0x200	unused
CONTENTS_UNUSED6	0x400	unused
CONTENTS_TEAM1	0x800	per team contents used to differentiate collisions between players and objects on different teams
CONTENTS_TEAM2	0x1000	
CONTENTS_IGNORE_NODRAW_OPAQUE	0x2000	ignore CONTENTS_OPAQUE on surfaces that have SURF_NODRAW
CONTENTS_MOVEABLE	0x4000	hits entities which are MOVETYPE_PUSH (doors, plats, etc.)
CONTENTS_AREAPORTAL	0x8000	remaining contents are non-visible, and don't eat brushes

Quake3 的碰撞检测
碰撞检测的基本单位 Brush。
Brush 由若干 Brushside 和 texture 组成的，而 Brushside 由 plane 和 texture 组成，所有的 plane 切割而成的空间就是 brush 空间，是一个凸体，其中 Brush.texture 定义了 brush 包围的内容性质，而 Brushside.texture 则定义了包围面性质
输入信息
InputStart 要进行碰撞检测的起点
InputEnd 要进行碰撞检测的终点
输出信息
Startsolid 起点是否在 solid 中
Allsolid 起点和终点是否都在 solid 中
OutputFraction 最近碰撞点距离起点的百分比
OutputEnd 可以移动到的终止点
先来看如何对线段进行碰撞检测

```
Trace():
    // 初始化变量
    Startsolid = false
    Allsolid = false
    Fraction = 1.0f
    // 搜索 BSP 树跟叶节点中的 Brush 做碰撞计算
    CheckNode(0, 0.0f, 1.0f, InputStart, InputEnd)
    If(Fraction == 1.0f):
        // 说明没有发生碰撞
        OutputEnd = InputEnd
    Else
        // 发生了碰撞，计算可移动到的终止点
        OutputEnd = InputStart +  (InputEnd - InputStart)*Fraction


CheckNode(NodeIndex, StartFraction, EndFraction, Start, End):
    // 当前节点是叶节点，进行 Brush 检测
    If(NodeIndex < 0):
        Leaf = BSP.Leafs[-1-NodeIndex]
        Foreach(get Brush from Leaf):
            If(Brush.NumberOfBrushside > 0 &&
(BSP.Textures[Brush.TextureIndex].ContentFlags & PLAYER_SOLIID)):
                // 表示 Brush 区域可以进行碰撞检测
                CheckBrush(Brush)
    // 当前不是叶节点，继续搜索 BSP 树
    Node = BSP.Nodes[NodeIndex]
    Plane = BSP.Planes[Node.PlaneIndex]
    StartDistance = DotProduct(Start, Plane.Normal) - Plane.Distance
    EndDistance = DotProduct(End, Plane.Normal) - Plane.Distance

    If(StartDistance >= 0 && EndDistance >= 0):
        // 如果 start 和 end 都在分割面前面，则递归搜索左子树
        CheckNode(Node.children[0], StartFraction, EndFraction, Start,
    End)
    Else If(StartDistance < 0 && EndDistance < 0):
        // 如果 start 和 end 都是分割面后面，则递归搜索右子树
        CheckNode(Node.children[1], StartFraction, EndFraction, Start,
    End)
    Else
        // 如果 start 和 end 分别在分割面的两边
        Int Side
        Float fraction1, fraction2, fraction
        Vector3 intersection
```

```
    If(StartDistance < EndDistance):
        // start 在分割面后面
        Side = 1
        fraction1 = (StartDistance + EPSILON)/(StartDistance -
    EndDistance)
        fraction2 = (StartDistance + EPSILON)/(StartDistance -
    EndDistance)
    Else If(StartDistance > EndDistance):
        // start 在分割面前面
        Side = 0
        fraction1 = (StartDistance + EPSILON)/(StartDistance -
    EndDistance)
        fraction2 = (StartDistance - EPSILON)/(StartDistance -
    EndDistance)
    Else
        Side = 0
        fraction1 = 1.0f
        fraction2 = 0.0f
    // 数据修正
    If(fraction1 < 0.0f) fraction1 = 0.0f
    Else If(fraction1 > 1.0f) fraction1 = 1.0f
    If(fraction2 < 0.0f) fraction2 = 0.0f
    Else If(fraction2 > 1.0f) fraction2 = 1.0f
    // 计算交点数据
    fraction = StartFraction + (EndFraction-StartFraction)*fraction1
    intersection = Start + (End-Start)*fraction
    // start 和 intersection 遍历子树
    CheckNode(Node.children[Side], StartFraction, fraction, Start,
intersection)
    // 计算交点数据
    fraction = StartFraction + (EndFraction-StartFraction)*fraction2
    intersection = Start + (End-Start)*fraction
    // intersection 和 end 遍历子树
    CheckNode(Node.children[1-Side], fraction, EndFraction,
intersection, End)



CheckBrush(Brush brush):
    // 注意 Brush 只是由平面切割而成的凸体
    startout = false
    endout = false
    fraction1 = -1.0f
    fraction2 = 1.0f
```

```
Foreach(get brush's plane from brush):
    StartDistance = DotProduct(InputStart, plane.normal) -
plane.distance
    EndDistance = DotProduct(InputEnd, plane.normal) -
plane.distance

// 标识起点、终点是否在 Brush 凸体之外
If(StartDistance > 0):
    startout = true
If(EndDistance > 0):
    endout = true

// 对线段和 Brush 凸体进行碰撞计算
If(StartDistance > 0 && EndDistance > 0):
    // 起点和终点在 Brush 凸体之外，即线段 InputStart 到
    // InputEnd 没有和该 Brush 发生碰撞
    Return
Else If(StartDistance < 0 && EndDistance < 0)
    // 起点和终点均在凸体某个分割面之后，线段仍可能会
    // 跟别的 Brush 面相交
    Continue
Else
    If(StartDistance > EndDistance):
        temp = (StartDistance - ESPILON)/(StartDistance -
    EndDistance)
        If(temp > fraction1): fraction1 = temp
    Else
        temp = (StartDistance + ESPILON)/(StartDistance -
    EndDistance)
        If(temp < fraction2): fraction2 = temp

// 输出起点终点是否在 solid 信息
If(startout == false):
    Startsolid = true  // 起点在 solid 中
    If(endout == false):
        Allsolid = true  // 起点和终点都在 solid 中

    // 计算可以移动距离因子 0.0-1.0
    If(fraction1 < fraction2):
        If(fraction1 > -1.0f && fraction1 < OutputFraction):
            If(fraction1 < 0.0f): fraction1 = 0.0f
            OutputFraction = fraction1
```

上面用来碰撞检测一条线段的基本思想：搜索 BSP 树，根据节点的分割面，将位

于分割面两侧的子线段，分别在左右子节点中递归进行搜索，直到到达叶节点；原始线段跟叶节点中所有 Brush 进行碰撞测试，具体的计算方法是，线段跟每个 Brush 的切割面进行相交计算，求出最近相交点的距离因子

如何对带有包围盒的 Player 运动轨迹进行计算呢？
跟前面线段的BSP 碰撞计算相比,这里仅仅多了输入参数 mins 和 maxs,即 Player 的包围盒，带有包围盒的碰撞计算跟线段的碰撞计算只有略微的差异
输入信息
InputStart 预测的运动起点
InputEnd 预测的运动终点
Mins 玩家包围盒的 mins
Maxs 玩家包围盒的 maxs

```
TraceBox():
    If(Mins[0] == 0.0f && Mins[1] == 0.0f && Mins[2] == 0.0f
    && Maxs[0] == 0.0f && Maxs[1] == 0.0f && Maxs[2] == 0.0f):
        // 退化为线段的碰撞检测
        Trace()
    Else
        // 计算包围盒碰撞计算要用到的一些参数
        BoxExtents[0] = -Mins[0] > Maxs[0] ? -Mins[0]:Maxs[0]
        BoxExtents[1] = -Mins[1] > Maxs[1] ? -Mins[1]:Maxs[1]
        BoxExtents[2] = -Mins[2] > Maxs[2] ? -Mins[2]:Maxs[2]
        Trace()
```

CheckNode 函数的变化
在将运动轨迹线段递归下降到叶节点时要考虑，包围盒产生的影响

```
    // 计算包围盒在分割面法线方向的最远点距原点的距离
    Offset = abs(plane.normal[0])*BoxExtents[0] +
             abs(plane.normal[1])*BoxExtents[1] +
             abs(plane.normal[2])*BoxExtents[2]
    。。。
    If(StartDistance >= Offset && EndDistance >= Offset)
    。。。
    If(StartDistance < -Offset && EndDistance < -Offset)
    。。。
    If(StartDistance < EndDistance):
        // start 在分割面后面
        Side = 1
        fraction1 = (StartDistance - Offset + EPSILON)/(StartDistance -
    EndDistance)
        fraction2 = (StartDistance + Offset + EPSILON)/(StartDistance -
    EndDistance)
    Else If(StartDistance > EndDistance):
        // start 在分割面前面
        Side = 0
```

```
    fraction1 = (StartDistance + Offset + EPSILON)/(StartDistance - EndDistance)
    fraction2 = (StartDistance - Offset - EPSILON)/(StartDistance - EndDistance)
```

CheckBrush 函数的变化

```
    // 求包围盒和 Brush 切割面的最近距离
    Vector3 temp
    Temp[0] = plane.normal[0] > 0 ? Mins[0]:Maxs[0]
    Temp[1] = plane.normal[1] > 0 ? Mins[1]:Maxs[1]
    Temp[2] = plane.normal[2] > 0 ? Mins[2]:Maxs[2]
    StartDistance = Dot(InputStart+Temp, plane.normal)-plane.distance
    EndDistance = Dot(InputEnd+Temp, plane.normal)-plane.distance
```