第三次第六周小班讨论（个人资料）

高级树结构剖析

Spatial data partitioning trees—
空间数据分割树

湖南大学
HUNAN UNIVERSITY

前言 PREFACE

空间数据结构是指空间数据的编排方式和组织关系。空间数据编码是指空间数据结构的具体实现，是将图形数据、影像数据、统计数据等资料按一定的数据结构转换为适合计算机存储和处理的形式。不同数据源采用不同的数据结构处理，内容相差极大，计算机处理数据的效率很大程度取决于数据结构。

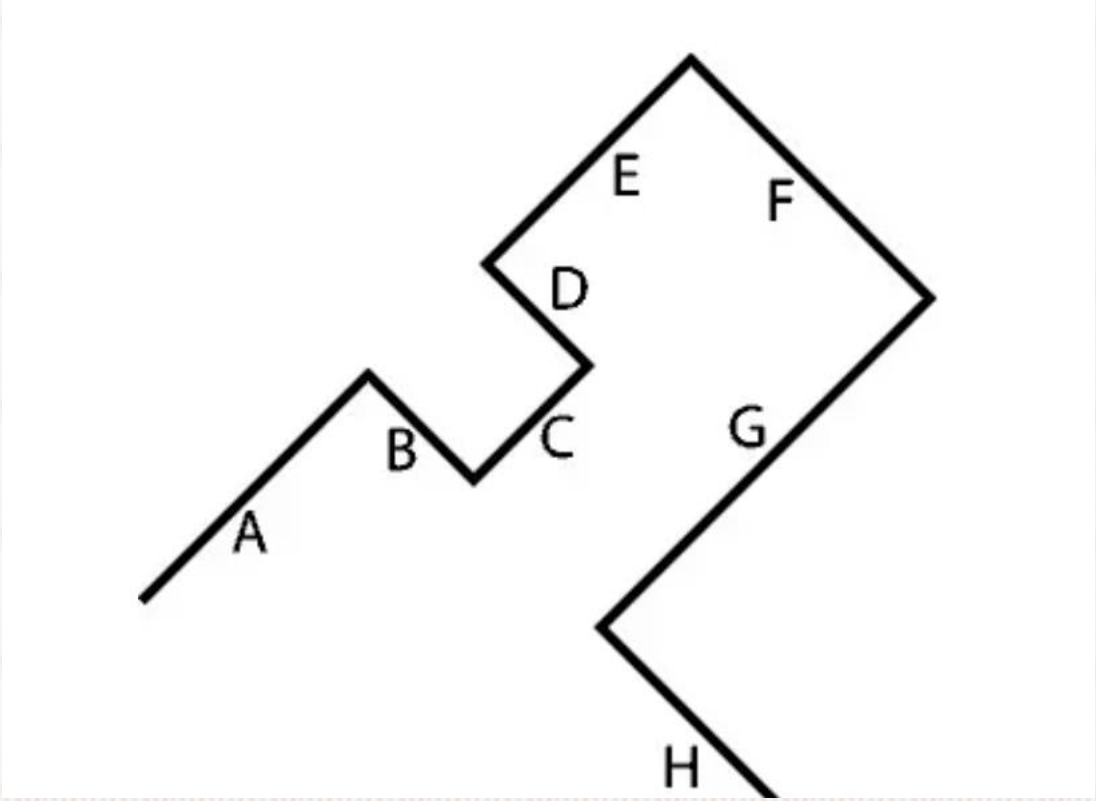
空间数据结构是带有结构的空间数据单元的集合。这些数据单元是数据的基本单位，一个数据单元可以由几个数据项组成，数据单元之间存在某种联系叫做结构。所以，研究空间数据结构，是指研究空间目标间的相关关系，包括几何和非几何的关系。数据结构是数据模型的表述，数据结构往往通过一系列的图表和矩阵，以及计算机码的数据记录来说明。

目录

湖南大学
HUNAN UNIVERSITY

第一部分

概述



空间数据分割树

空间数据分割树（Binary Space Partioning Tree，BSP Tree）是一种用于处理几何体数据的数据结构。它在计算机图形学、游戏开发和其他领域中有广泛的应用。

- BSP 树是一棵二叉树，也被称为**二维空间分割树**。
- 它可以被视为 **kd 树** 的推广应用。
- 与 kd 树类似，BSP 树也是二叉树，但是现在可以任意选择分割平面的方向和位置。

空间数据分割树的分类

- **KD树：**

kd 树（k-dimensional tree）是一个包含空间信息的二项树数据结构，它是用来计算 kNN 的一个非常常用的工具。如果特征的维度是 D，样本的数量是 N，那么一般来讲 kd 树算法的复杂度是 O(DlogN)，相比于穷算的 O(DN)省去了非常多的计算量。

- **BSP树：**

Binary space partitioning(BSP)是一种使用超平面递归划分空间到凸集的一种方法。使用该方法划分空间可以得到表示空间中对象的一个树型数据结构。这个树形数据结构被我们叫做BSP树。

湖南大学
HUNAN UNIVERSITY

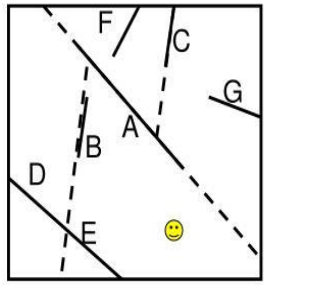
第二部分

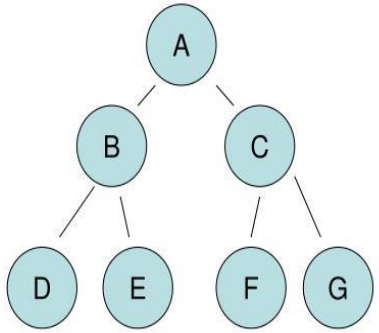
物理结构的定义

标题二 物理实现

副标题

Binary Space Partitioning Trees





- Determine which side of the plane the camera lies on by a point/plane comparison
- The polygon set on the far side of the plane is is beyond the near side set
- From the far side set again do the comparison and determine the far side of its splitting plane
- Back to front order is for this example is: F - G- A - D - B- E
- This does not tell which polygon is closest but which polygon occludes (hides) another polygon.

双亲表示法：

•原理：在这种表示法中，每个节点都包含一个指向其父节点的指针。根节点的父节点指针通常设置为特殊值（例如-1）。
•代码实现：使用结构体定义节点类型，其中包括数据域和父节点指针。整个树的数据信息存储在一个数组中

孩子表示法：

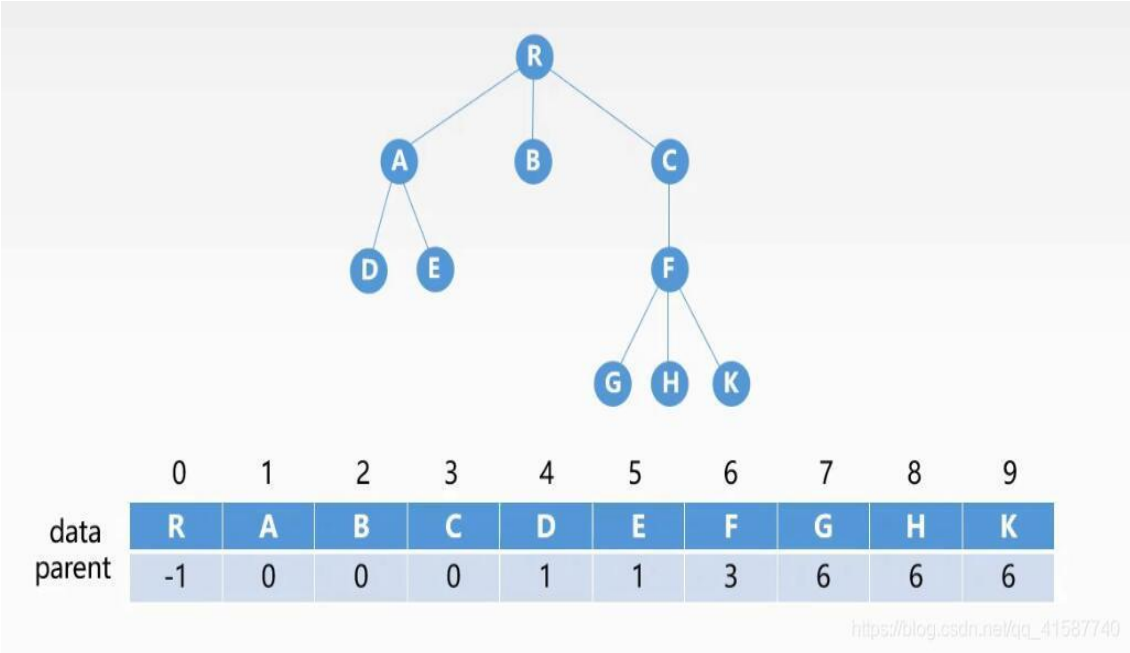
•原理：每个节点包含一个指向其孩子节点的指针。孩子节点之间使用单链表连接。
•代码实现：定义单链表节点类型，其中包括孩子节点的下标和指向下一个孩子节点的指针。整个树的数据信息存储在一个数组中

孩子兄弟表示法：

•原理：每个节点包含指向其第一个孩子节点和下一个兄弟节点的指针。
•代码实现：使用结构体定义节点类型，其中包括数据域、指向第一个孩子节点的指针和指向下一个兄弟节点的指针

双亲表示法

具 体 表 示



	0	1	2	3	4	5	6	7	8	9
data	R	A	B	C	D	E	F	G	H	K
parent	-1	0	0	0	1	1	3	6	6	6

R为头节点，所以parent=-1；

ABC的双亲节点数组下标为0，所以parent=0；

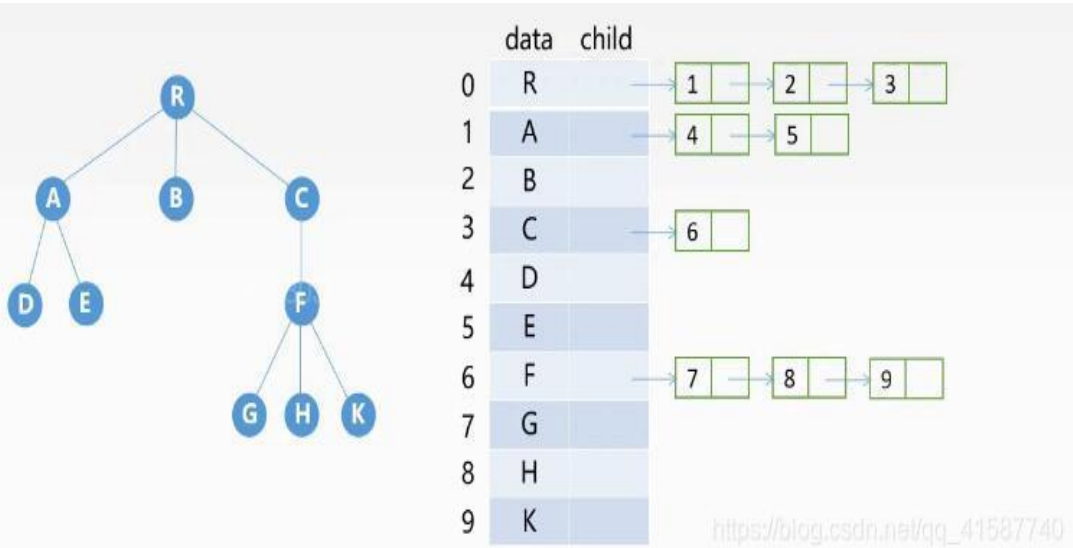
DE的双亲节点数组下标为1，所以parent=1；

代码实现

```
1  typedef struct{
2      int data;        //数据域
3      int parent;      //伪指针
4  }PTNode;             //节点类型
5
6  typedef  struct{
7      PTNode nodes[MAX_TREE_SIZE];        //节点数据信息
8      int n;           // 节点个数
9  }PTree;              //树的类型
10
```

孩子表示法

具 体 表 示



代码实现

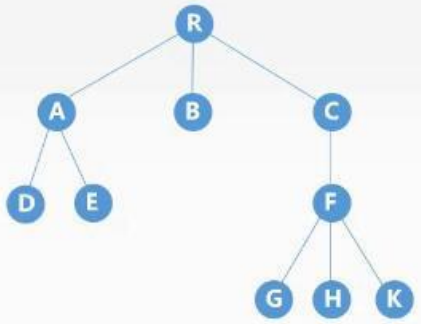
```
typedef struct CNode{
    int child;                  //孩子节点的下标
    struct CNode *next;         //下一个孩子节点的指针
}CNode;                         //单链表节点类型

typedef struct{
    int data;                   //节点数据
    struct CNode *firstchild;   //孩子节点的第一个节点，即单链表的头节点
}PNode;                         //树的节点类型

typedef struct{
    PNode nodes[MAX_TREE_SIZE]; //所有节点数据
    int n;                      //节点个数
}CTree;                         //树的类型
```

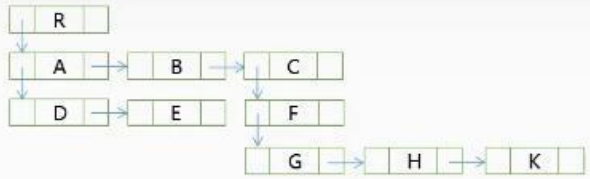
孩子兄弟表示法

具 体 表 示

以二叉链表作为树的存储结构，又称二叉树表示法。

左孩子右兄弟





代码实现

```
typedef struct CSNode{
    int data;
    struct CSNode *fristchild,*nextsibling;
}CSNode,*CSTree;
```

湖南大学
HUNAN UNIVERSITY

第三部分

基本操作

BSP树的生成算法

列出所有墙面

选择其中一堵墙为分割墙，将空间分割为P1和P2两个部分。

将完全在分割墙前面的所有墙放入一个列表中，称为“前面”。

将所有完全位于分隔墙后面的墙放入另一个名为“back”的列表中。

如果分隔墙与任何墙壁相交，则将该墙分成两部分，将部分墙分离器壁的前部插入前列表，并在后面的列表中的其他部分。

将分割墙放置在bsp树中。第一堵墙只是成为树的根。此后，如果在场景中墙位于父墙的前面，则墙将放置在其父墙右侧的 bsp 树中。如果在场景中墙位于父墙的后面，则墙将放置在父墙左侧的 bsp 树中。

代码实现

Init

```
void BspTree<T>::InitBspTree(std::vector<Face<T>> &_faces,Point<T> min_point,Point<T> max_point,int max_depth)
{
    maxDepth=max_depth;
    root=new BspTreeNode<T>;
    root->depth=1;
    root->faces=_faces;
    root->lChild=nullptr;
    root->rChild=nullptr;

    root->maxPoint=max_point;
    root->minPoint=min_point;

    splitSpace(root,AXIS_X,1);
}
```

Delete

```
void BspTree<T>::deleteTree(BspTreeNode<T> *root)
{
        if(root)
        {
                deleteTree(root->lChild);
                deleteTree(root->rChild);
                delete root;
                root=nullptr;
        }
}
```

代码实现 CutFace

```
void BspTree<T>::cutFace(const Face<T> &face,Axis axis,const T &splitter
{
        T p[3]; //记录某个方向的点分量
        switch(axis)
        {
        case AXIS_X:
                p[0]=face.point[0].x;
                p[1]=face.point[1].x;
                p[2]=face.point[2].x;
                break;
        case AXIS_Y:
                p[0]=face.point[0].y;
                p[1]=face.point[1].y;
                p[2]=face.point[2].y;
                break;
        case AXIS_Z:
                p[0]=face.point[0].z;
                p[1]=face.point[1].z;
                p[2]=face.point[2].z;
                break;
        }
        //对分隔到两边的顶点计数
        for(int i=0;i<3;++i)
        {
                if(p[i]<splitter)
                        leftCount++;
                else if(p[i]>splitter)
                        rightCount++;
                else if(fabs(p[i]-splitter)<1e-6)
                        bothCount++;
        }
}
```

SplitSpace

```
void BspTree<T>::splitSpace(BspTreeNode<T> *node,Axis axis,int depth)
{
        //在当前层节点做分割处理，为下层子节点数据做填充

        if(!node)
                return;
        node->axis=axis;
        node->depth=depth;
        if(depth==maxDepth)
                return;
        if(node->faces.size()<2) //上层节点拥有的面片数太少就不再划分
                return;

        node->lChild=new BspTreeNode<T>;
        node->rChild=new BspTreeNode<T>;
        //先给子节点包围盒赋值
        node->lChild->maxPoint=node->maxPoint;
        node->lChild->minPoint=node->minPoint;
        node->rChild->maxPoint=node->maxPoint;
        node->rChild->minPoint=node->minPoint;

        T xLen=node->maxPoint.x-node->minPoint.x;
        T yLen=node->maxPoint.y-node->minPoint.y;
        T zLen=node->maxPoint.z-node->minPoint.z;
        //设置该节点的划分点
        Axis mAxis=AXIS_X;
        if(yLen>xLen&&yLen>zLen)
                mAxis=AXIS_Y;
        if(zLen>xLen&&zLen>yLen)
                mAxis=AXIS Z:
```

```
for(int i=0;i<node->faces.size();++i)
{
    // 重置计数
    int leftCount=0;
    int rightCount=0;
    int bothCount=0;

    //对每个面做切分到子节点
    cutFace(node->faces[i],mAxis,node->splitter,leftCount,rightCount,bothCount
    if(leftCount||bothCount)
        node->lChild->faces.push_back(node->faces[i]);
    if(rightCount||bothCount)
        node->rChild->faces.push_back(node->faces[i]);
}
```

湖南大学
HUNAN UNIVERSITY

第四部分

应用

•自动生成室内portal

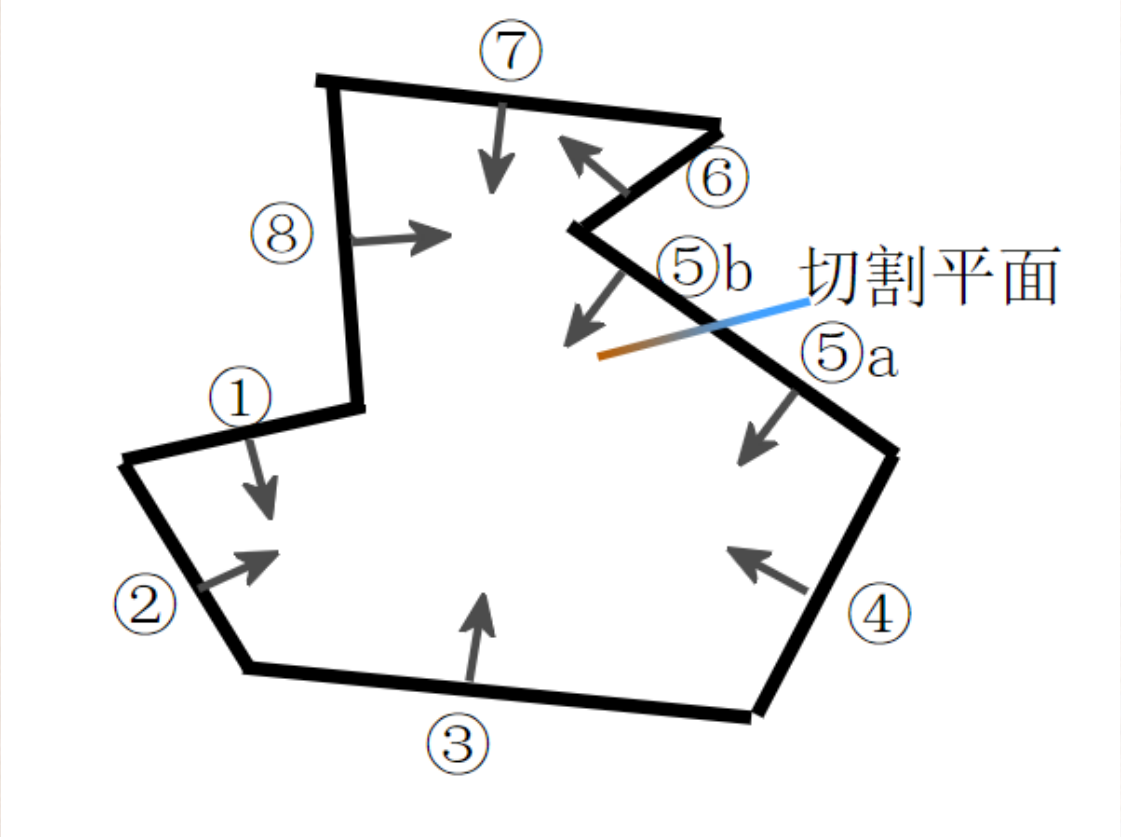
大型室内场景游戏引擎基本离不开portal系统：
1.portal系统可在运行时进行额外的视野剔除，过滤掉很多被遮挡的物体渲染，有效地优化室内渲染。
2.portal系统还可以离线构造PVS(潜在可见集)，计算出在某个划分区域潜在可以看到哪些其他区域，将这些数据存储成一个潜在可见集；在运行时根据该集合实时加载潜在可看到的区域。

但是对于关卡编辑师来说，对每个房间/大厅/走廊/门...手动放置每个portal无疑是极大的工作量。于是有一种利用BSP树自动生成portal的做法，大致做法是：
1.首先，将室内需要渲染的墙体/门/柱子等室内较大物体所代表的边缘作为需要处理的平面，然后基于这些平面构造BSP树。

2.将BSP树节点相连着的左节点视为一个儿子，右节点视为一个邻居。

3.所有相连的父子节点所代表的平面组成了一个凸多边形房间。

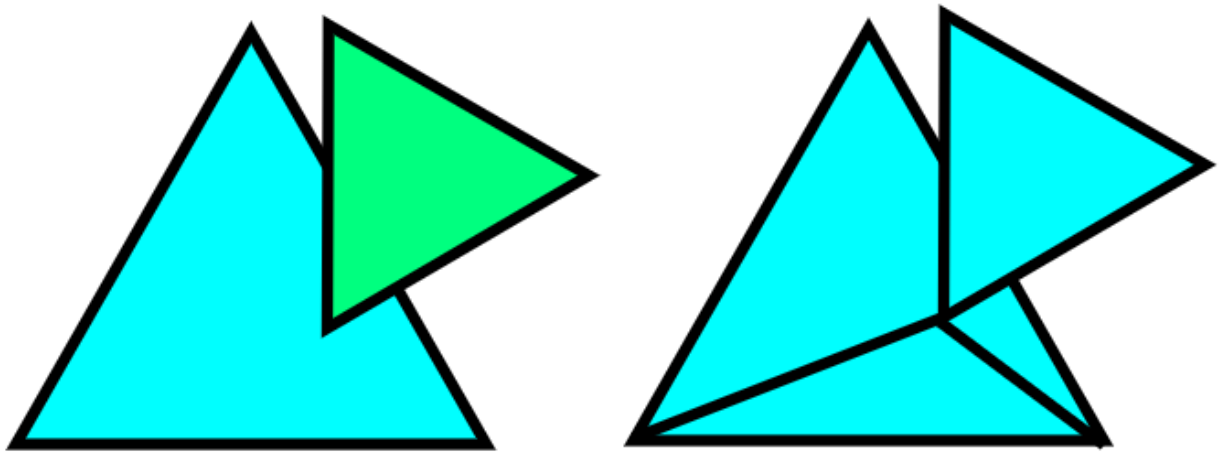
4.计算每个相邻的房间之间相衔接的点，称为portal点。



•自动生成导航网格

导航网格（Nav Mesh）是一种表示凸多边形的节点，目前主流游戏的游戏AI寻路中最常用的节点种类。通过用导航网格，AI寻路所要搜索的节点数量大大减少且变得灵活。

因此我们可以预先计算可移动地形和静态障碍，得到一个不规则的大地图形状（可能有凹处也可能有中空处），然后以该形状的所有边来构造一棵BSP树来分割得到若干个凸多边形房间，而这些分割出来的每个凸多边形都是一个导航网格。

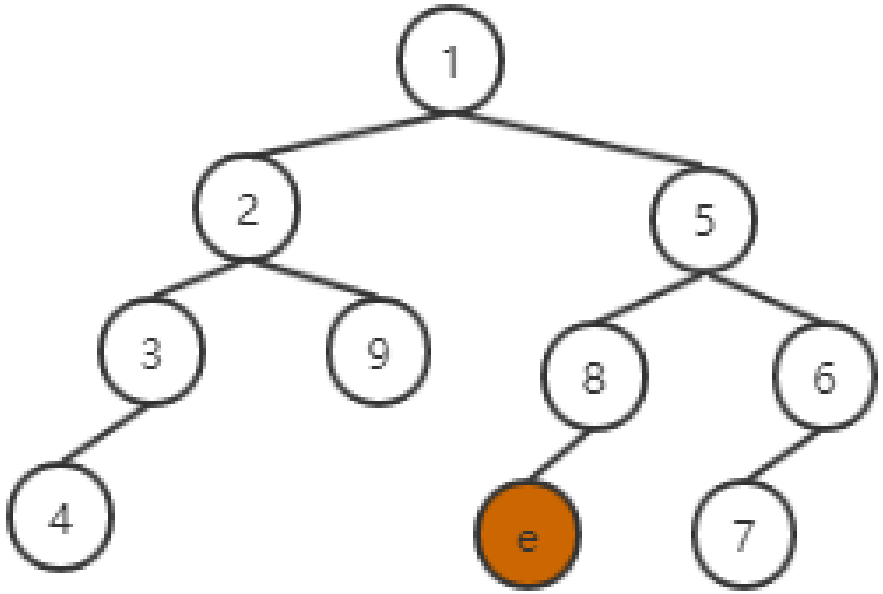


•渲染顺序优化

在至少二十多年前，老旧硬件是没有深度缓存，程序员使用BSP树从远到近渲染（从远处到摄像机位置）三角形图元，避免较远的三角形覆盖到较近的三角形上，从而到达正确的三角形图元渲染顺序，这也就是古老的画家算法。

首先根据摄像机的位置，遍历BSP树找到并记录其位置相对应的叶节点,称之eyeNode，它将会是顺序遍历渲染的一个重要的中止条件。
由于eyeNode往往是在一些平面的前面，另一些平面的后面，所以为了达到正确的从近到远的顺序，需要两次不同方向的遍历。

从远处到eyeNode处的遍历顺序：
第一次遍历，左中右顺序，从根节点开始，直到eyeNode停止；
第二次遍历，右中左顺序，从根节点开始，直到eyeNode停止。



该BSP树节点代表的数据应该是一个三角形（渲染的基本图元），因为恰好三角形也是个平面形状，因此该BSP树节点代表的平面也就是其数据本身。

谢 谢 观 看

thank you for watching

参考文献

- 《游戏编程精粹5(Game Programming Gems 5)》Kim.Pallister [2007-9]
- 《游戏编程精粹6(Game Programming Gems 6)》Michael Dickheiser [2007-11]
- BSP技术详解1 - Dreams - 博客园