# CG简介

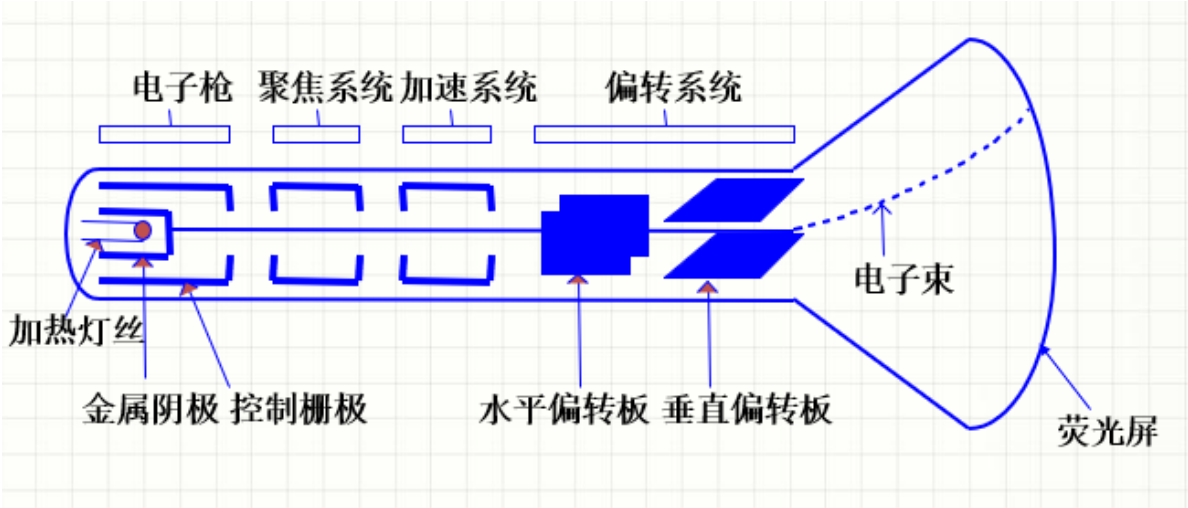
计算机图形学分三部分：

1. 建模：创建3D几何对象的模型，即三维物体模型的计算机表示，主要是形状表述与定义
2. 渲染：生成3D场景的2D表示（图像）
3. 动画：描述对象的运动

# 图形系统硬件

## 视频显示设备

### 刷新式CRT




工作原理：

高速的电子束由电子枪发出，经过聚焦系统、加速系统和偏转系统就会到达荧光屏的特定位置。由于荧光物质在高速电子的轰击下会发生电子跃迁，即电子吸收到能量从低能态变为高能态，由于高能态很不稳定，在很短的时间内荧光物质的电子会从高能态重新回到低能态，这时将发出荧光，屏幕上产生亮点。

荧光屏：

1. 荧光物质特性：颜色、余辉
2. 两种状态：受激态、基本态
3. **余辉**：定义为荧光质从发光开始到亮度衰减到最大亮度的10%的这段时间，通常为10~60ms
    - 长余辉荧光层适合显示复杂的静态图形，而短余辉荧光层适合动画应用

光点：电子束打在显示器的荧光屏，显示器能够显示的最小发光点→硬件最小图像点

物理直径：光点亮度下降到最大亮度的60%处的直径

## 光栅扫描&随机扫描

| 0 | 随机扫描显示器 | 光栅扫描显示器 |
| --- | --- | --- |
| 电子束扫描方式 | 电子束只扫向图形所在的屏幕位置，一次一条线<br>随机系统的刷新速率依赖于显示的线数<br>30~60HZ；100,000条<br>当显示的线条很少时，则延迟每个刷新周期，避免刷新速率超过每秒60帧，否则线条刷新过快，烧坏荧光层 | 电子束自顶向下全屏扫描<br>逐行扫描/隔行扫描<br>刷新率、闪烁 |
| 图形定义 | 图形定义为一组画线命令，存储在被称为刷新显示文件的内存区域 | 图形定义为一组图元信息，存储在被称作帧缓冲区的内存区域<br>Pixel (Picture element)像素点<br>帧缓冲区容量=显示分辨率x色深 |
| 适用范围 | 适于画线应用 | 适于显示包含细微阴影和色彩模式的真实场景 |

## 图形信息储存

像素的强度值存储在帧缓冲区(Frame Buffer)的内存区域中

帧缓存区容量计算：

1. 色深：像素点信息的长度，单位为bit(s)
    - eg: 彩色光栅系统支持显示16色，像素点需要4bits代表不同颜色，例如0001代表蓝色，此时色深为4bits
2. 帧缓冲区容量=显示分辨率x色深

## 彩色CRT

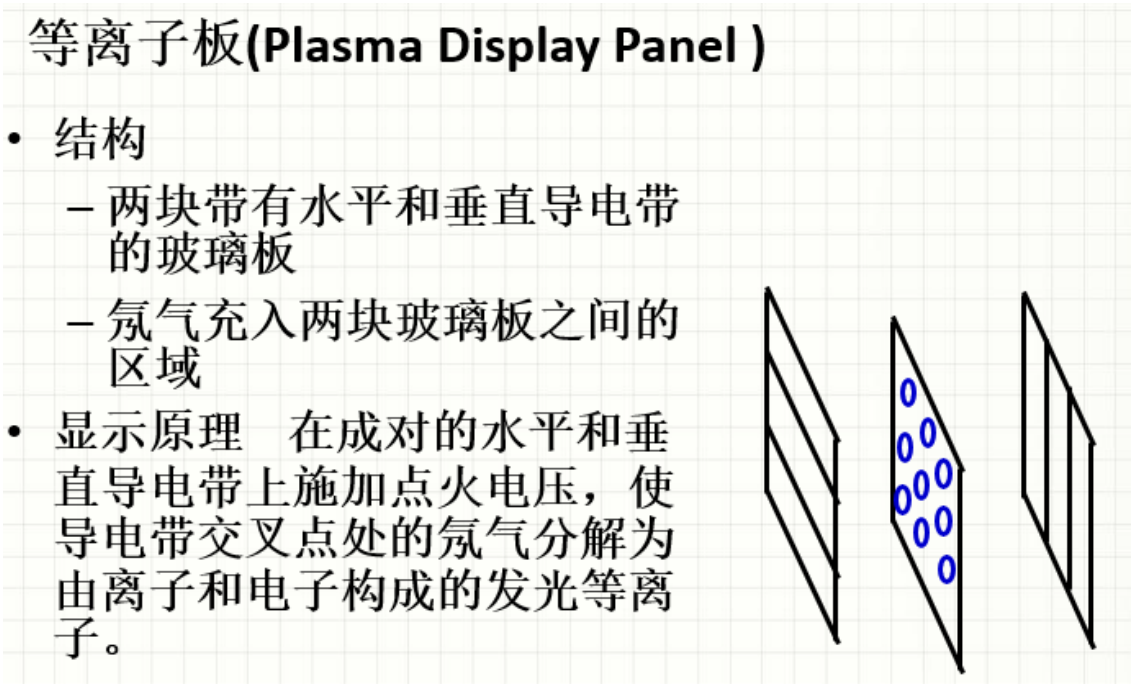
利用能发射不同颜色光的荧光层的组合来显示彩色图形

CRT产生彩色的基本技术

- 电子穿透法→用于随机扫描显示器
- 荫罩法 →用于光栅扫描显示器

## 平板显示器

1. 发射型显示器：将电能转化为光能
    - LED
    - 等离子板




- 薄膜光电显示器
- OLED

2. 非发射型显示器：利用光学效应将光源转化为图形
   - LCD：通过阻塞或传递来自周围的或内部光源的偏振光形成图案

# 图形软件类型

分为两类：

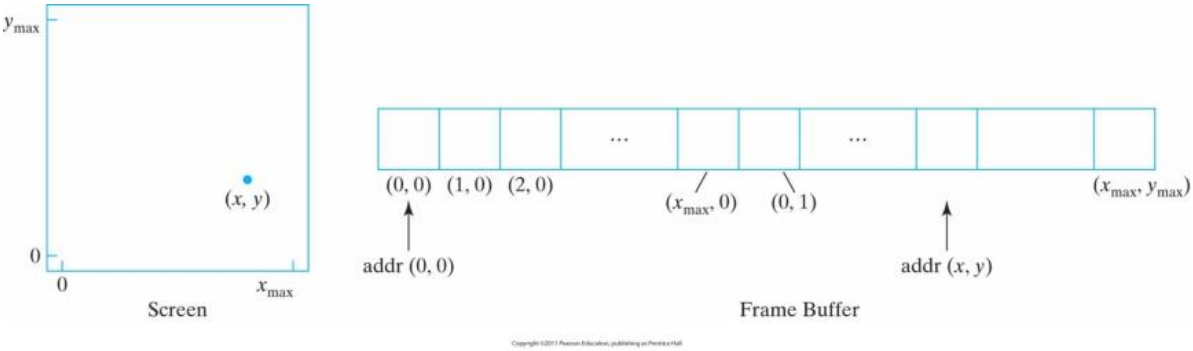
1. 通用编程软件包：为C, FORTRAN一类的高级语言编程提供一组图形函数。基本功能包括生成基本图形、属性设置、视图选择以及图形变换功能等。
2. 专用应用软件包：为非程序员设计，用户可生成图形而无需考虑图形生成原理。 例如 CAD

# 基本图元

## 点绘制

根据坐标(x,y)，对帧缓冲区对应存储单元内容进行操作以画点

### 点的储存




$addr(x,y) = addr(0,0) + (y \times (X_{max} + 1) + x) * \text{color depth}$

# 直线绘制

## DDA

算法流程：

> 1. 选取变化速度较快的轴为采样轴（以取x轴为例）
> 2. 根据直线的斜率式 $y = mx + b$，进行如下循环操作：
>    - $x+ = 1$ 或 $x- = 1$
>    - $y+ = m$ 或 $y- = m$
>    - 绘制 $(x, y)$，注意四舍五入
> 3. 直到采样轴达到终点

评价：

- 没有使用乘法
- 除法运算、浮点数操作和取整操作任然耗时
- 较长线段会产生误差积累现象

## Bresenham

算法流程：

> 1. 已知目前点为 $(x_k, y_k)$ 绘制下一个点 $(x_{k+1}, y_{k+1})$
> 2. 此时的决策函数 $P_k = \begin{cases} P_{k-1} + 2\Delta y & P_{k-1} < 0 \\ P_{k-1} + 2\Delta y - 2\Delta x & P_{k-1} \geq 0 \end{cases}$
> 3. 若 $P_k < 0$，则选择 $y_{k+1} = y_k$
> 4. 若 $P_k < 0$，则选择 $y_{k+1} = y_k + 1$
>
> 注意：
>
> - 此算法适用于直线斜率$m \in (0, 1)$
> - $P_0 = 2\Delta y - \Delta x$

## 中点画线算法

算法流程：

> 1. 已知目前点为 $(x_k, y_k)$ 绘制下一个点 $(x_{k+1}, y_{k+1})$
> 2. 令 $F(x, y) = (mx + b) - y$：
>    - 若 $F(x_k + 1, y_k + 0.5) = 0$，下一个点选 $(x_k + 1, y_k)$ 或 $(x_k + 1, y_k + 1)$
>    - 若 $F(x_k + 1, y_k + 0.5) > 0$，下一个点选 $(x_k + 1, y_k)$
>    - 若 $F(x_k + 1, y_k + 0.5) < 0$，下一个点选 $(x_k + 1, y_k + 1)$

# 圆形绘制

## 中点画圆算法

算法流程：

> 1. 从 $(x_0, y_0) = (0, r)$ 画起
> 2. 初始决策参数为 $P_0 = \frac{5}{4} - r$，若 $r$ 为整数则可提前取整
> 3. $P_{k+1} = \begin{cases} P_k + 2x_{k+1} + 1 & P_k < 0 \\ P_k + 2x_{k+1} - 2y_{k+1} - 1 & P_k \geq 0 \end{cases}$
> 4. 若 $P_k < 0$，则下一个点为 $(x_{k+1}, y_{k+1}) = (x_k + 1, y_k)$
> 5. 若 $P_k \geq 0$，则下一个点为 $(x_{k+1}, y_{k+1}) = (x_k + 1, y_k - 1)$
> 6. 循环直到$x \geq y$
>
> 注意：
>
> - 计算顺序为：$k \to P_k \to (x_{k+1}, y_{k+1}) \to 2x_{k+1} \to 2y_{k+1}$

## Bresenham

算法流程：

> 1. $P_0 = 3 - 2r$
> 2. $P_{k+1} = \begin{cases} P_k + 4x_k + 6 & P_k < 0 \\ P_k + 4(x_k - y_k) + 10 & P_k \geq 0 \end{cases}$
> 3. 从 $(x_0, y_0) = (0, r)$ 画起
> 4. 若 $P_k < 0$，则下一个点为 $(x_{k+1}, y_{k+1}) = (x_k + 1, y_k)$
> 5. 若 $P_k \geq 0$，则下一个点为 $(x_{k+1}, y_{k+1}) = (x_k + 1, y_k - 1)$
> 6. 循环直到$x \geq y$

## 总结

Bresenham算法以决策参数的增量计算为基础，仅包括简单的整数处理

中点方法更易应用于其他圆锥曲线，沿任何圆锥曲线所确定的像素位置，其误差限制在像素分段的1/2以内

# 图元属性

## 存储模式

- 直接存储
  - 优点：
    - 特别灵活，可同时使用颜色范围内的任一颜色
    - 硬件简单（仅仅在概念上）
  - 缺点：
    - 帧缓冲区存储容量需求大
    - 视频操作速度慢（例如24位色光栅系统，改变一个像素需要操作3个字节）
- 颜色查找表
  - 优点：
    - 使用彩色表可提供合理的、同时显示的颜色数量，而无需大容量帧缓冲器。 256或512种不同颜色足以显示单个图像
    - 表项可随时改变，容易在场景设计、图形使用时试验不同颜色组合的效果，而无需改变对图形数据结构的属性设置
    - 视频操作速度快，因为每个像素在帧缓冲区中占有更少的数据位
    - 有些图形系统提供两种能力，方便用户选择
  - 缺点：
    - 需要高速 RAM-DAC
    - 能同时使用颜色数有限

## 灰度

当RGB函数中指定相同的红、绿、蓝分量时，产生的色彩是某种程度的灰色

## 走样

图形数字化过程中，由于低频采样而造成的图形畸变

解决方法：

1. 提高硬件分辨率
2. 反走样技术：
   - **过取样（后滤波）**：将屏幕看成由比实际更细的网格所覆盖，从而增加取样频率，然后根据这种更细的网格，使用取样点来确定屏幕中每个像素的亮度等级
     - 单像素直线：先用画线算法在高分辨率下画，然后转换到低分辨率
     - 有限宽度直线：计算高分辨率下在直线内的小方格数，再转换至低分辨率
     - 子像素的加权掩模：将像素分块后为每个像素赋予不同的权重，最后求和来确定亮度。一般靠中心位置的子像素亮度高。
   - **区域取样（前滤波）**：通过计算待显示的每个像素在对象上的覆盖区域来确定像素亮度
     - 有限宽直线：计算覆盖区占像素格比例来确定像素亮度
     - 过滤技术：它是一种连续的加权方案，是更精确的反走样方法。类似于应用加权像素掩码，用一个连续的加权曲面(或过滤函数)覆盖像素，而不是一个离散的象素掩模。常用过滤函数：正方形、圆锥和高斯函数。应用过滤函数的方法是将像素曲面集成来得到加权的平均亮度。
3. **像素移相技术**：将电子束移动像素点直径的1/4，1/2或3/4，使描绘的点更接近直线的真实路径。

4. **直线段亮度差的校正**：由于斜线单位长度的亮度低于水平线和垂直线，光栅系统中的斜线比水平和垂直线暗。所以可以根据直线斜率调整亮度来校正。

# 图形填充

## 多边形识别

### 叉积法

绕多边形的周长计算相邻边向量的叉乘，如果各叉积Z分量均同号，则为凸多边形；如果有正、有负，则为凹多边形

### 延长线法

延长每边，观察：如果顶点分布在延长线两侧，则为凹多边形；否则为凸多边形

## 分割多边形

算法流程：

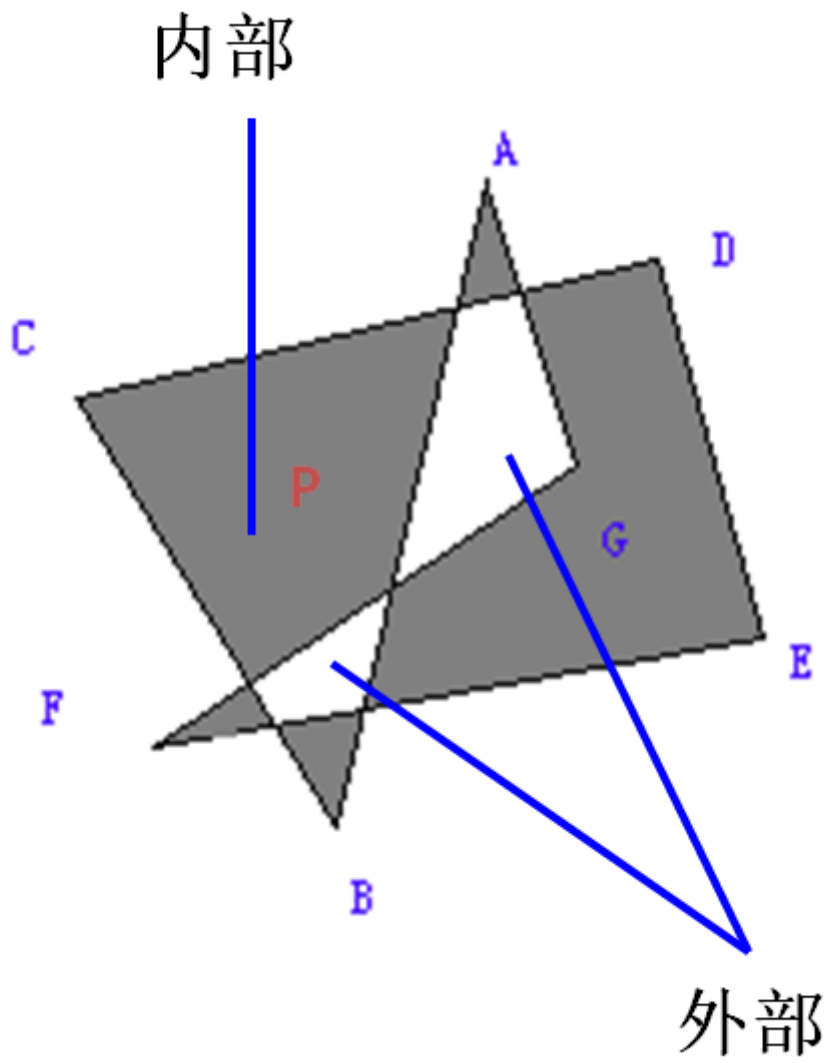
> 1. 按逆时针方向计算多边形的边向量的叉积
> 2. 并记录叉积结果Z分量的符号
> 3. 如果Z分量变为负值，则多边形为凹多边形，可以沿叉乘向量对中的第一条边的延长线将多边形分解开

## 内外测试

### 奇偶规则

算法流程：

> 1. 对于每一个区块，取其内任意一个点，做任意一条射线
> 2. 计算穿过射线的多边形边数
> 3. 若穿过的边数为奇数为内部区域，否则为外部区域

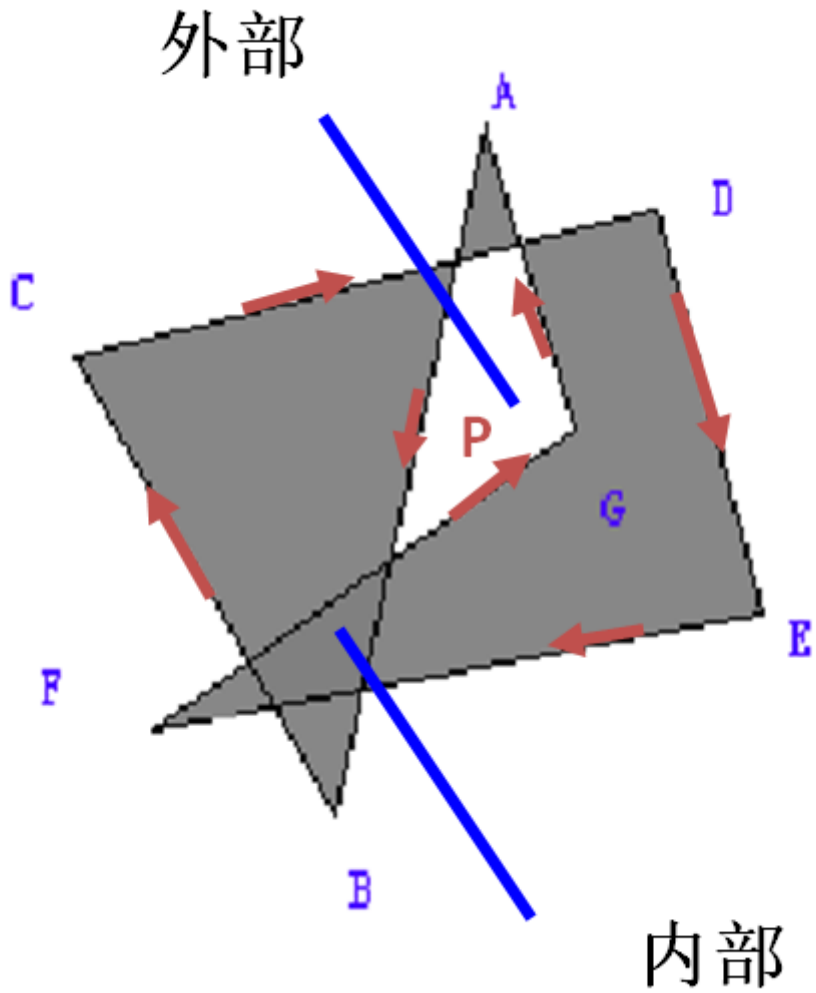



## 非零环绕数规则

适用于有规定方向的多边形

算法流程：

> 1. 环绕数初始为零
> 2. 对于每一个区块，取其内任意一个点，做任意一条射线
> 3. 射线从有向边的**右侧穿至左侧**，环绕数加1
> 4. 射线从有向边的**左侧穿至右侧**，环绕数减1
> 5. 非零为内部区域




# 区域填充算法

## 边的存储

### 有序边表

按顶点输入顺序依次形成边

存储到每条边最小Y值所对应的扫描线位置（水平边除外）

相同最小y值的边按较低顶点x值升序排列（类似处理哈希冲突）

最小储存单元存储信息包含：$y_{终点}$，$x_{起点}$，$\frac{1}{m}$

### 活化边表

把与当前扫描线相交的边称为活化边

并把它们按与扫描线交点x坐标递增的顺序存放在一个链表中,形成活化边表

实则一个单链表，是有序边表里抽出来的一条

## 扫描线法

扫描线自底向上扫描，计算扫描线与多边形边界的交点确定填充区间，再用要求的颜色显示这些区间的像素，即完成填充工作。一条扫描线的填充过程分为求交、排序、配对和填色四个步骤

**注意**：对于单调临边，我们采取将较低边的 $y$ 缩小 1

## 边界填充算法

连通判定

- 4连通
  - 有时不能完整填充
- 8连通
  - 有时会产生越界

算法流程：

> 指定内部点入栈；当栈非空时重复执行如下操作：
>
> 1. 获得当前栈顶像素的颜色
> 2. 若像素颜色既不是边界色也不是填充色，则置成填充色；依次将右、左、上、下四邻接点压栈
> 3. 若当前栈顶像素已是边界点或已填充，则弹栈

特点：堆栈空间需求量非常大

## 泛洪式填充算法

同边界填充算法

但是判定改成了：将是原填充色的像素改为指定填充色

# 字符

# 字符字模构成方式

## 点阵式

位图字体

采用逐位映射的方式得到字符的字模编码

特点：

- 易于定义显示
- 空间需求大
- 变化困难

## 矢量式

轮廓字体

将字符笔画分解为线段，以线段端点坐标为字符字模的编码

编码顺序：

- 自然笔画
- 连续笔画

特点：

- 空间需求小
- 时间开销大
- 无极缩放

# 几何变换

## 坐标储存

对于点 $P(x,y)$

**标准化设备坐标**为：

$$P=\begin{pmatrix}x\\y\end{pmatrix}$$

**齐次坐标**为：

$$P=\begin{pmatrix}x\times h\\y\times h\\h\end{pmatrix}$$

$h$ 一般等于1

## 2D几何变换

### 平移变换

$$P'=T(t_x,t_y)\times P$$

$$\Longrightarrow\begin{pmatrix}x'\\y'\\1\end{pmatrix}=\begin{pmatrix}1&0&t_x\\0&1&t_y\\0&0&1\end{pmatrix}\times\begin{pmatrix}x\\y\\1\end{pmatrix}$$

### 旋转变换

$$P' = R(\theta) \times P$$

$$\Longrightarrow \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

逆时针为正

## 缩放变换

$$P' = S(S_x, S_y) \times P$$

$$\Longrightarrow \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

## 复合变换

$$P' = M \times P$$
$$M = M_n \times M_{n-1} \times \cdots \times M_2 \times M_1$$

注意操作顺序与矩阵排列顺序的关系

## 其他变换

反射：

1. x轴反射
2. y轴反射
3. 原点反射
4. 45度直线反射

错切：

1. x轴方向错切
2. y轴方向错切

## 几何变换的光栅方法

一些简单的变换可通过将储存像素值的矩形数组从帧缓冲器的一个位置移到帧缓冲器的另一个位置。这样只需很少的算术操作，因此像素变换特别有效。

操纵矩形像素数组的光栅功能通常称为光栅操作，将一块像素从一个位置移到另一位置称为像素的块移动。在二值系统中，这个操作称为位块移动。

# 3D几何变换

## 平移变换

$$P' = T(t_x, t_y, t_z) \times P$$

$$\Longrightarrow \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

### 旋转变换

$$P' = R_x(\theta) \times P$$

$$\Longrightarrow \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

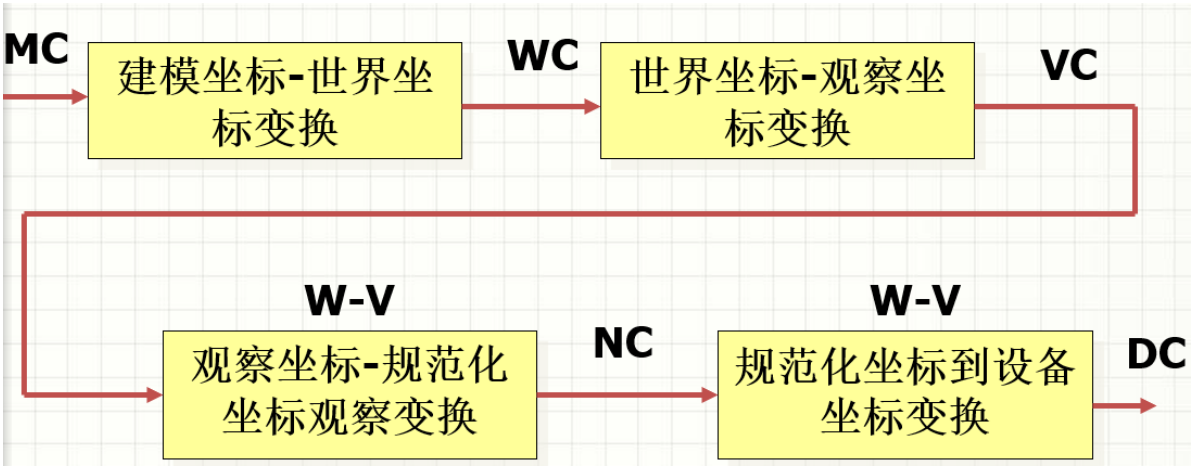
### 缩放变换

$$P' = S(S_x, S_y, S_z) \times P$$

$$\Longrightarrow \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

# 2D窗口裁剪

## 二维观察变换流水线



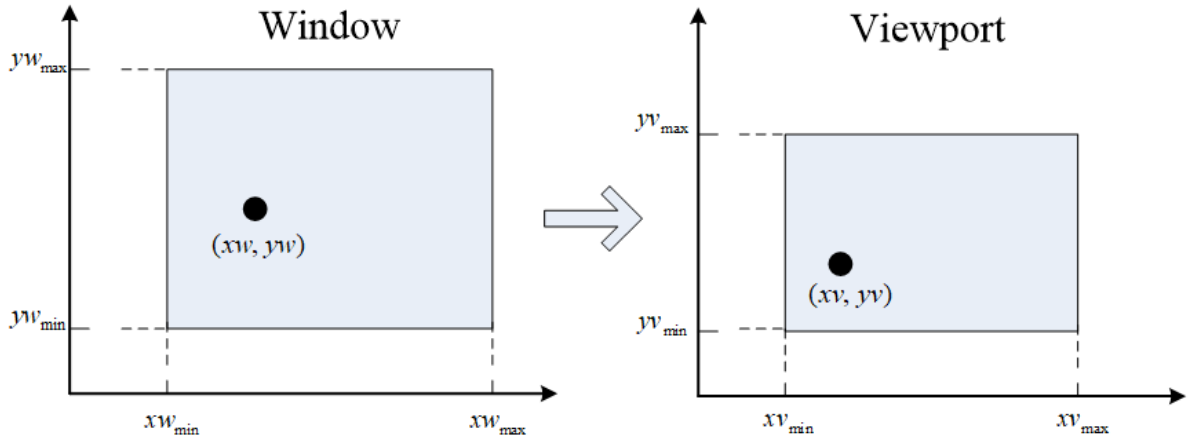

$WC \to DC$：**二维观察变换**，或简称为**窗口到视口的变换**或**窗口变换**

**视区、视口(viewport)**：显示设备上用于窗口映射的坐标区域

**裁剪窗口(clipping window)或窗口(window)**：常规图形系统中，世界坐标系中指定的用于显示的坐标区域

2D场景中，我们用世界坐标系中的向量**V**作为观察坐标系的 $y_{view}$ 轴方向。向量**V**称为二维观察向上向量。

## 窗口→视口变换




保持视口与窗口中的对象具有同样的相对位置，必须满足

$$\frac{xv - xv_{\min}}{xv_{\max} - xv_{\min}} = \frac{xw - xw_{\min}}{xw_{\max} - xw_{\min}}$$

$$\frac{yv - yv_{\min}}{yv_{\max} - yv_{\min}} = \frac{yw - yw_{\min}}{yw_{\max} - yw_{\min}}$$

$$X_v = S_x X_w + t_x$$

$$Y_v = S_y Y_w + t_y$$

缩放系数

$$S_x = (xv_{max} - xv_{min}) / (xw_{max} - xw_{min})$$

$$S_y = (yv_{max} - yv_{min}) / (yw_{max} - yw_{min})$$

平移系数

$$t_x = (xw_{max}\, xv_{min} - xw_{min}\ \ xv_{max}) / (xw_{max} - xw_{min})$$

$$t_y = (yw_{max} yv_{min} - yw_{min} yv_{max}) / (yw_{max} - yw_{min})$$

**规范化**：将裁剪窗口映射至$[-1,1] \times [-1,1]$规范化正方形再映射到屏幕窗口

因为规范化视口是$[0,1] \times [0,1]$，所以可以先规范化一次再显示

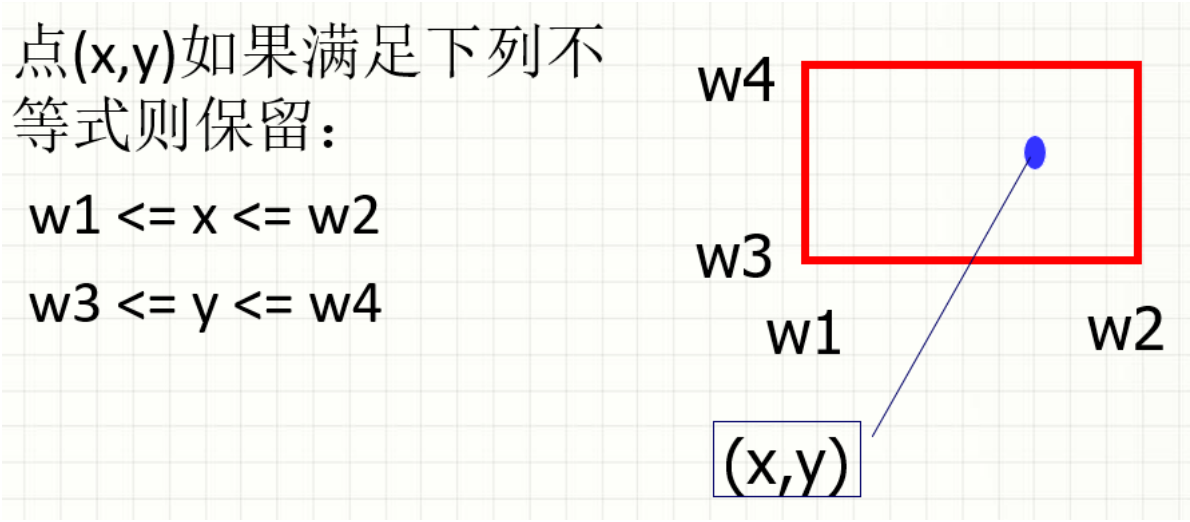
# 裁剪

消除指定区域内或区域外的图形部分的过程称为**裁剪算法**，简称裁剪

裁剪时机：

- 针对窗口裁剪：只有窗口内的部分映射到设备坐标系中，不用将多余图元变换到设备空间中
- 针对视口裁剪：映射后，用视口边界裁剪，可通过合并观察和几何变换矩阵来减少计算量

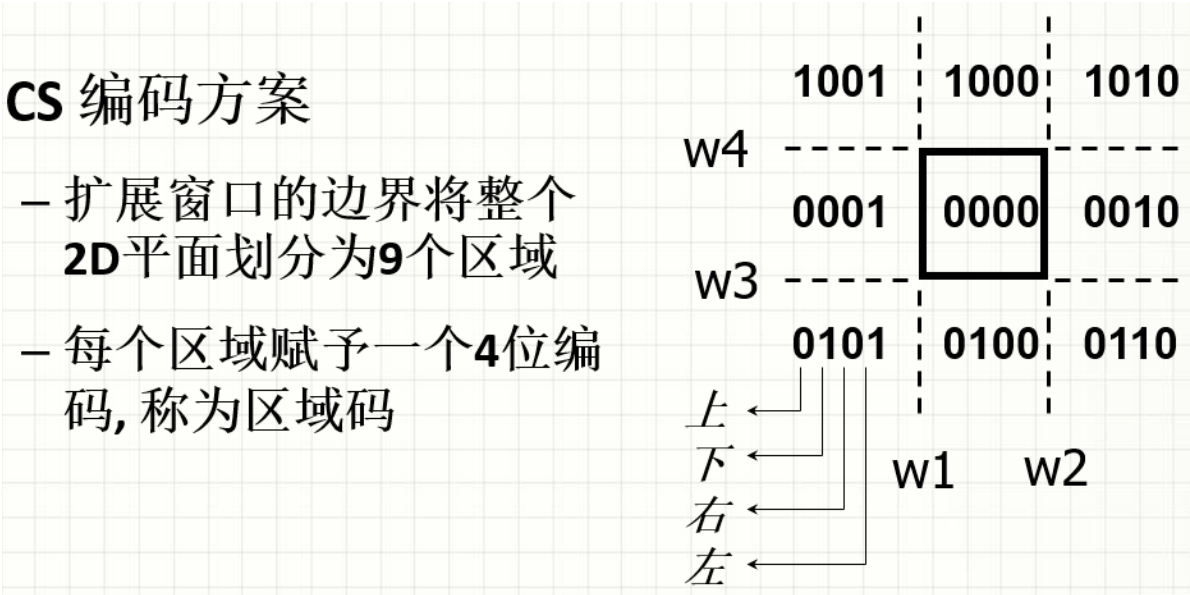
## 点裁剪




## 直线裁剪

### CS直线裁剪算法

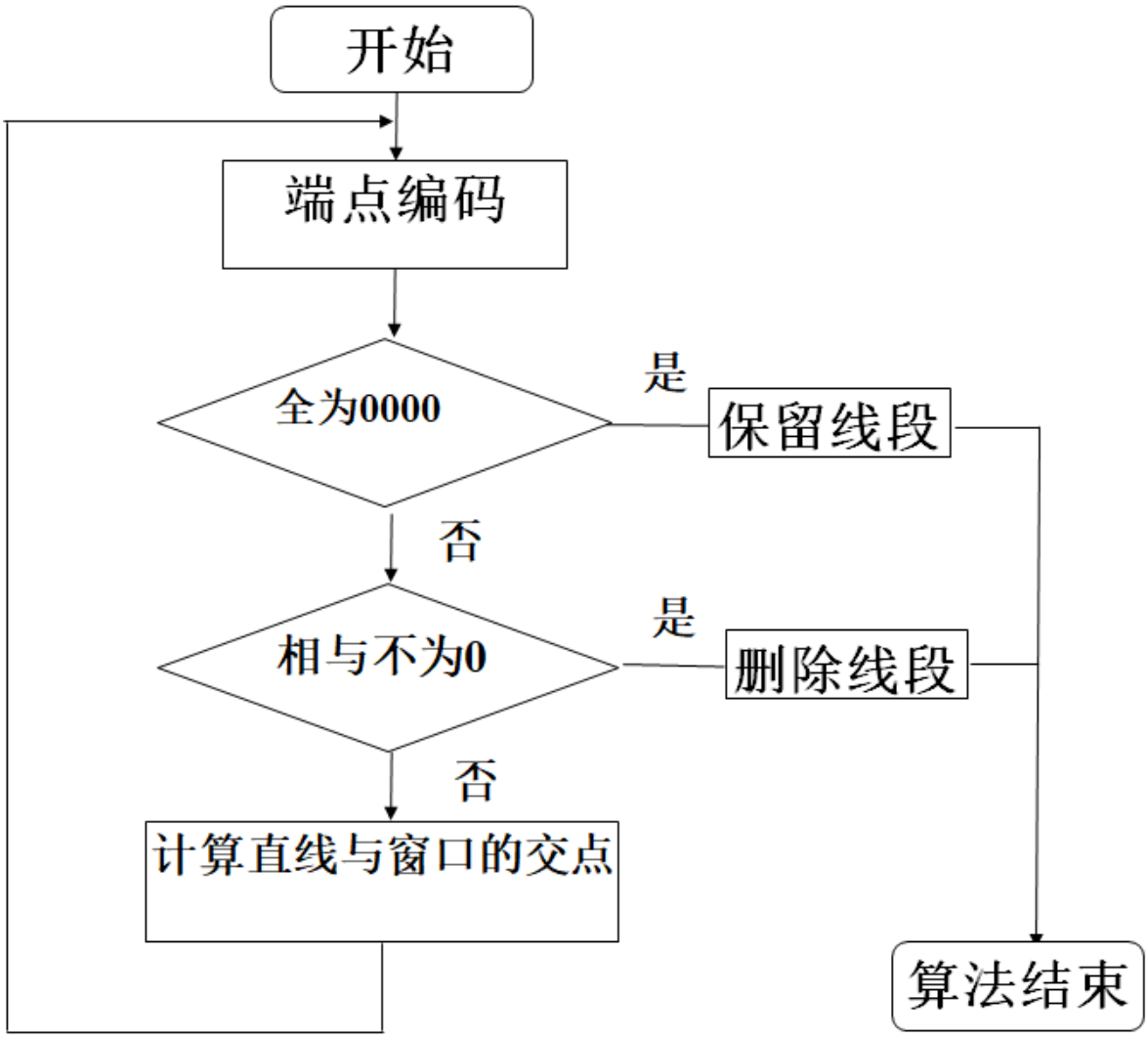
采用区域编码，编码方案如下：




算法描述：

1. 计算直线端点编码，$c_1$和$c_2$
2. 判断:
   - $c_1$**和**$c_2$**均为0000**，保留直线
   - $c_1 \& c_2$**不为零**，同在某一边界外，删除该直线
   - $c_1 \& c_2$**为零**，需要进一步求解交点
3. **先起点后终点，以L,R,B,T 为序**，找出端点区域码中第一位为1的位，将x=w1、w2或y=w3、w4代入直线方程，计算直线与窗口边界的交点，将交点和另一端点重复上述过程，直至线段保留或删除

算法流程图：




评价：

- 简单，易于实现
- 算法中求交点的次数决定了算法的速度

### LB直线裁剪算法

直线的起点为$P_1(x_1,y_1)$，终点为$P_2(x_2,y_2)$

直线的参数方程为$\begin{cases} x=x_1+(x_2-x_1)u \\ y=y_1+(y_2-y_1)u \end{cases}$ 其中 $u\in[0,1]$

直线在窗口内则满足如下不等式组：

$$\begin{cases} w_1\le x_1+(x_2-x_1)u\le w_2 \\ w_3\le y_1+(y_2-y_1)u\le w_4 \end{cases}$$

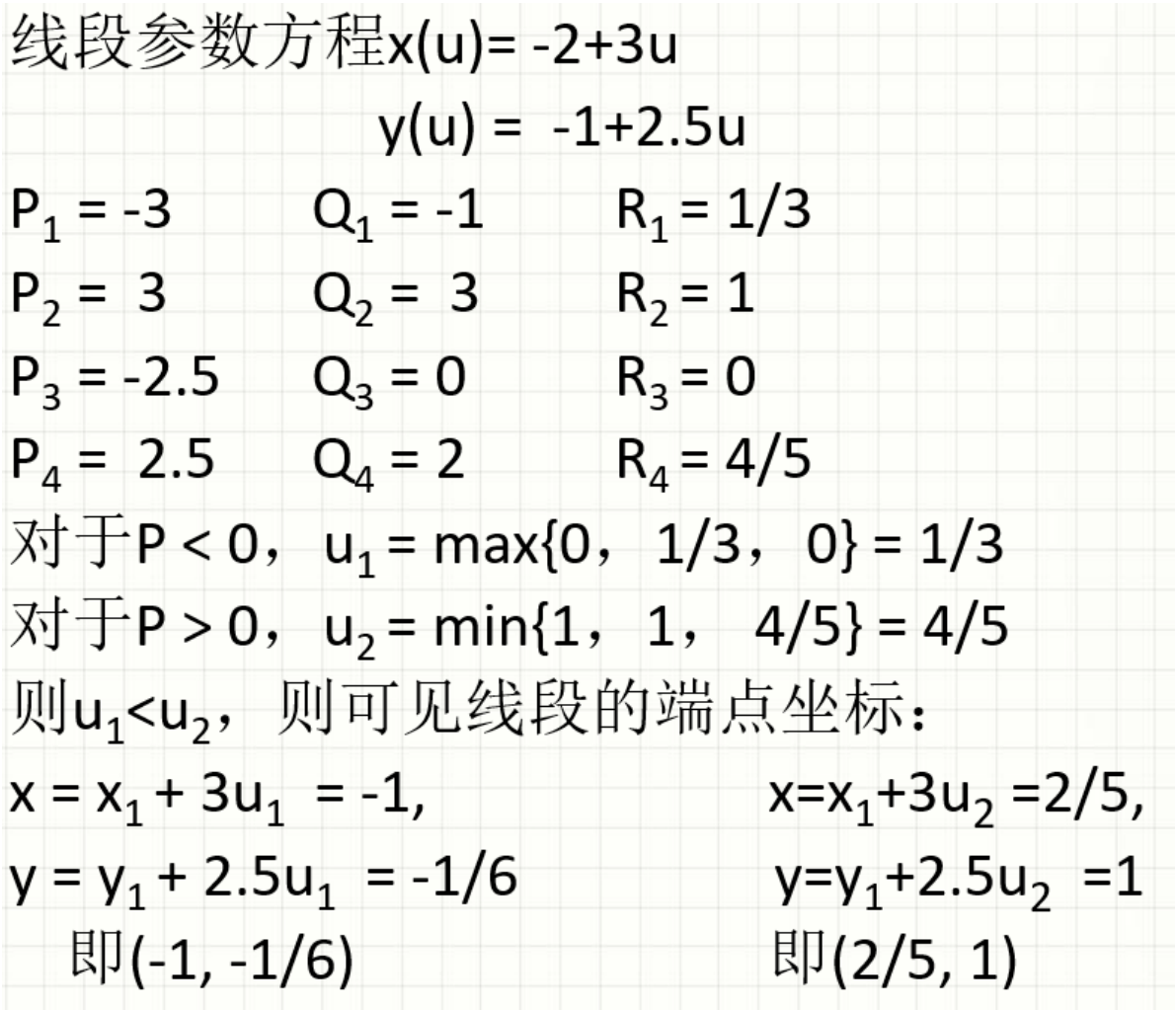
整理成关于 $u$ 的不等式组：

$$P_k\times u\le Q_k \qquad \text{k=1,2,3,4}$$
$$\begin{cases} P_1=-\Delta x & Q_1=x_1-w_1 \\ P_2=\Delta x & Q_2=w_2-x_1 \\ P_3=-\Delta y & Q_3=y_1-w_3 \\ P_4=\Delta y & Q_4=w_4-y_1 \end{cases}$$

算法描述：

1. 计算$P_k$和$Q_k$
2. 若存在$k$，使$P_k = 0$且$Q_k < 0$，则舍去直线
3. 对其他情况通过计算$Q_k / P_k$来确定交点所对应的$u$值
4. $\begin{cases} u_1 = \max(\{\frac{Q_k}{P_k} | P_k < 0\}, 0) \\ u_2 = \min(\{\frac{Q_k}{P_k} | P_k > 0\}, 1) \end{cases}$
5. 如果$u_1 > u_2$，则直线在窗口外，否则计算交点坐标$\begin{cases} x(u) = x_1 + \Delta x \cdot u \\ y(u) = y_1 + \Delta y \cdot u \end{cases}$

书写模板:




评价:

LB 效率高于 CS， 因为减少了交点计算次数。 参数u1和 u2的更新需要四次除法，交点坐标计算至多4次乘法。

LB和CS都易于拓展成三维裁剪算法

LB和CS经过调整约束形式也可用于非矩形窗口裁剪

### NLN直线裁剪算法

通过在裁剪窗口周围创立多个区域，并在求交运算之前进行更多的区域测试，从而避免对直线段进行多次剪裁

**仅仅适用于2D裁剪**

算法简述：将起点和矩形裁剪窗口的4个顶点相连构成4条射线，进而分成4个区域，通过判断终点的区域来确定使用哪条边来裁剪

总结：求交前的检测是为了避免过多的不必要的计算。

优点：对于必须求交点的情况，给出确定的相交边界，从而利用参数方程快速求取交点

## 多边形区域裁剪

### SH算法

以多边形顶点为初始集合，首先用窗口左边界剪裁多边形，产生新的顶点序列。新的顶点集依次传给右边界、下边界和上边界进行处理。 SH算法最终输出定义剪裁后的多边形边界的顶点序列

裁剪顺序：**左右下上**

将现有边依次输入裁剪器，输入的边是顶点按顺序两两组合形成的边集

**输出规则如下**：

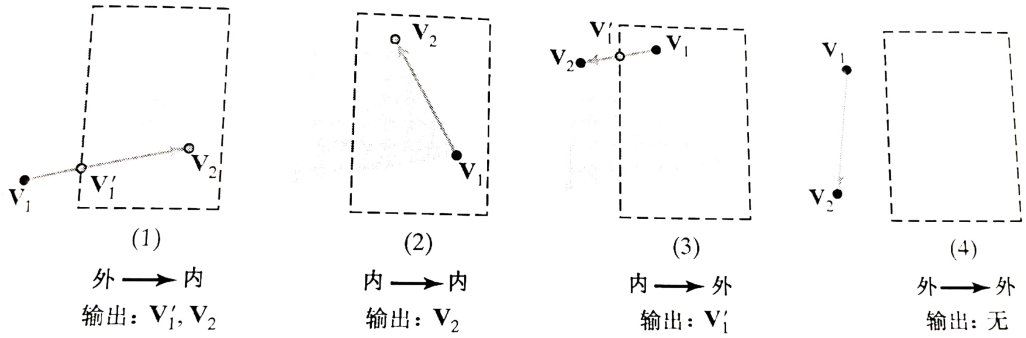



图 8.24　按一对端点相对于裁剪窗口左边界的位置，左裁剪器可能生成的四种输出

书写模板：

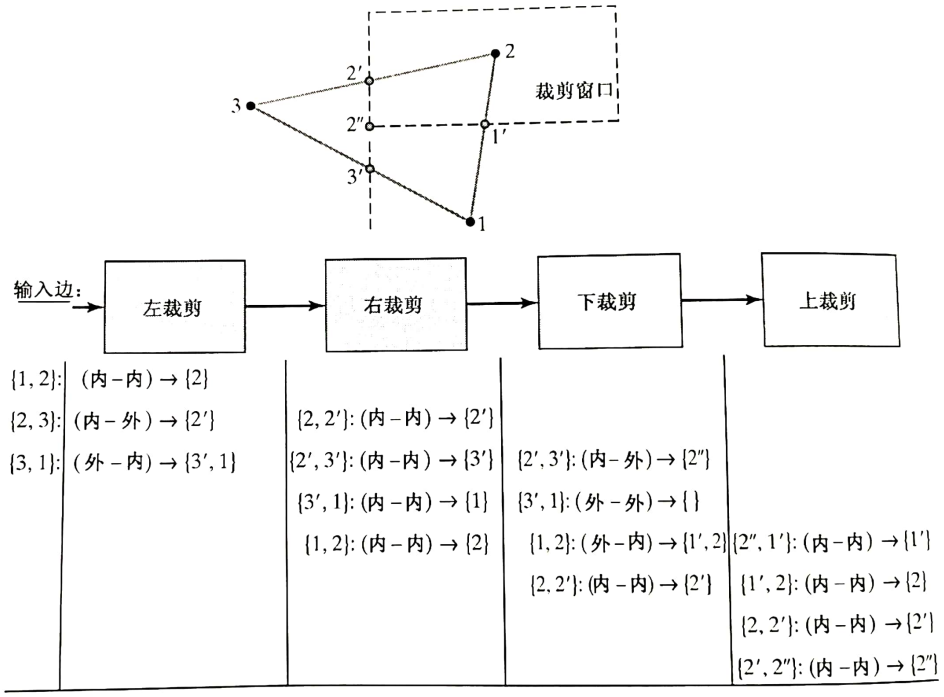



图 8.25　使用 Sutherland-Hodgman 算法对多边形顶点集{1，2，3}进行处理。最终裁剪结果是顶点集{1′，2，2′，2″}

问题：只适用于凸多边形的裁剪

改进：

1. 先裁剪成多个凸边形再用SH
2. 检测同一边界上是否有两个以上的点时，将顶点队列分成两个或多个

### WA算法

通过修改多边形顶点处理顺序(考虑顶点处理方向)，从而正确显示凹多边形。规则大致如下：

**对由外到内的顶点对**，沿着多边形边界的方向；

**对由内到外的顶点对**，沿着窗口边界的方向直到到达另一个与多边形的交点并且该交点是已处理的顶点

回到交点并继续按原顺序处理多边形的边

具体处理：按以上操作循环，将在裁剪范围内的点加入顶点队列，如果出现队列闭环，则说明裁剪出了一个封闭多边形了，此时创建一个新的队列来储存下一个封闭多边形的顶点集

非矩形窗口裁剪则同理，判断内外时需要进行内外检测

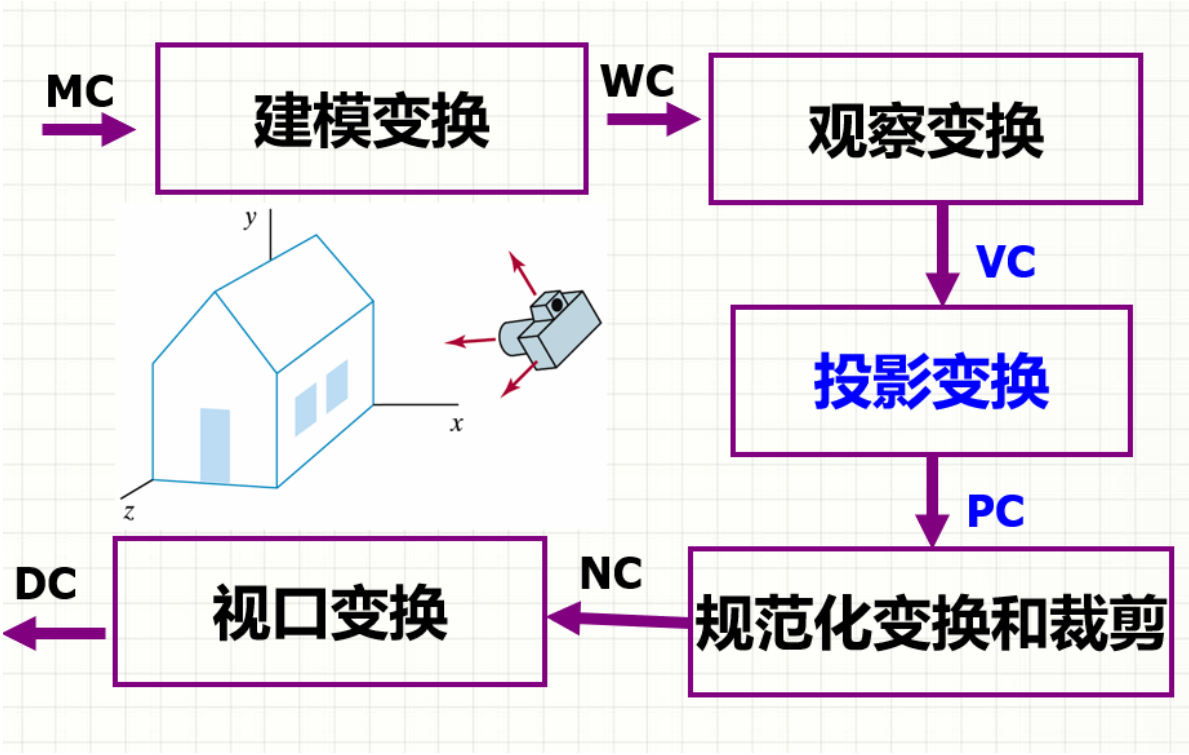
## 文本裁剪

裁剪策略：

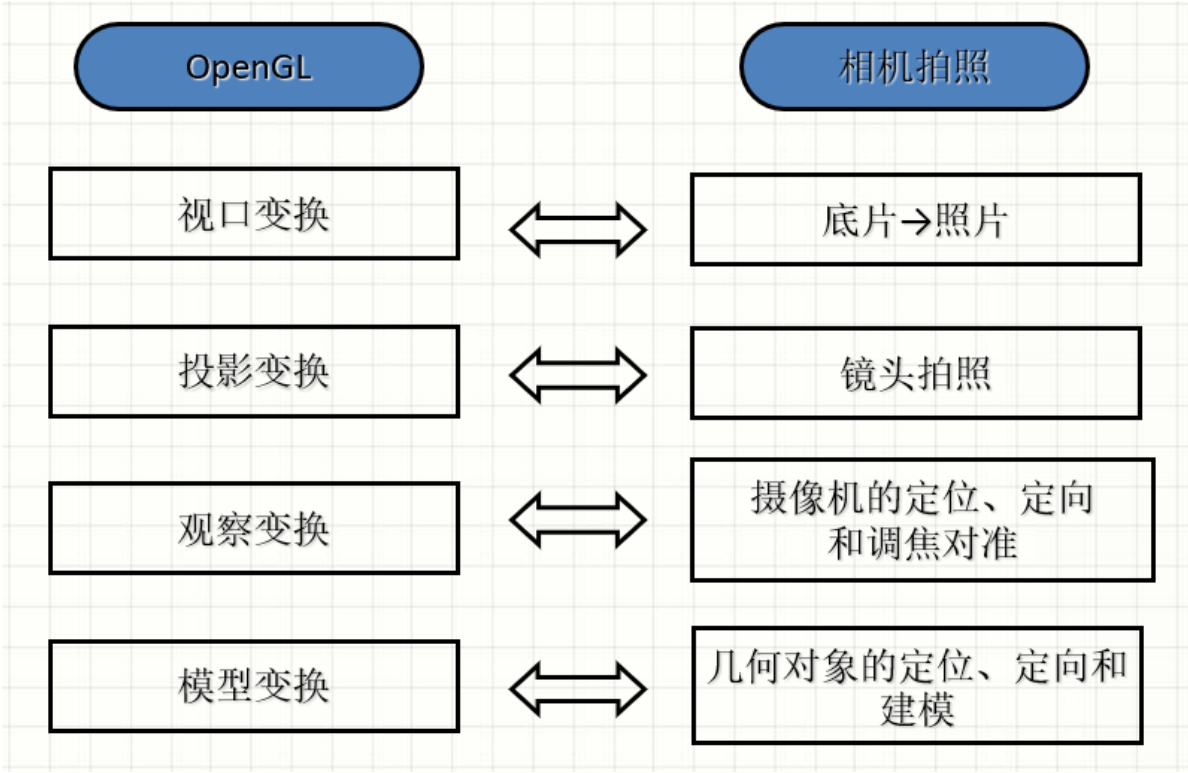
- 全有或全无字符串裁剪
- 全由或全无字符裁剪
- 单字符裁剪（笔画裁剪）

# 3D场景观察

## 三维观察流水线




各种变换的理解：




## 三维观察坐标系

首先在世界坐标系中选取一个点$P_0 = (x_0, y_0, z_0)$作为观察原点，称为**观察点、观察位置、视点**或**照相机位置**

观察方向一般沿$z_{view}$正方向，因此**观察平面**有时候也称**投影平面**，$z_{view}$正方向定义为**观察平面法向量**$\vec{N}$

> $\vec{N}$的方向有两种求法：
>
> 1. 世界坐标系原点到某选定点的连线
> 2. 参考点$P_{ref}$到$P_0$的连线，参考点称为注视点

之后选择**观察向上向量**$\vec{V}$来确定$y_{view}$轴正方向，选择不平行于$\vec{N}$的向量即可，一般令$\vec{V}=(0,1,0)$

于是可得：

$$\begin{cases}\overrightarrow{X_{view}}=\vec{V}\times\vec{N}\\ \overrightarrow{Y_{view}}=\vec{N}\times\overrightarrow{X_{view}}\\ \overrightarrow{Z_{view}}=\vec{N}\end{cases}$$

**uvn观察坐标参考系统**就是将xyz轴单位化的向量组

$WC\to VC$ 变换矩阵如下（先平移后旋转）：

$$M_{WC,VC}=\begin{pmatrix}u_x & u_y & u_z & -\vec{u}\cdot\overrightarrow{P_0}\\ v_x & v_y & v_z & -\vec{v}\cdot\overrightarrow{P_0}\\ n_x & n_y & n_z & -\vec{n}\cdot\overrightarrow{P_0}\\ 0 & 0 & 0 & 1\end{pmatrix}$$

# 投影

## 平行投影

### 正平行投影

对象描述沿与投影平面法向量$\vec{N}$平行的方向到投影平面的变换称为**正投影**或**正交投影**

可以显示对象多个侧面的正投影为**轴测正投影**，分为：

1. 正等轴测投影：三个轴向伸缩系数均相等（p= q=r）的正轴测投影
2. 正二轴测投影：两个轴向伸缩系数相等（p=q≠r或p=r≠q或q=r≠p）的正轴测投影
3. 正三轴测投影：三个轴向伸缩系数均不相等（p≠q≠r）的正轴测投影

> 伸缩系数正比于平面与坐标轴交点到原点的距离

投影矩阵的求法：将投影平面旋转至于坐标轴平面平行，再进行忽略深度的正投影

正投影规范化变换：

由于屏幕坐标经常指定为左手系，所以要将正投影观察体映射到左手坐标系的规范化立方体（$x,y,z\in[-1,1]$）

> 右手系到左手系只需将z轴取负值即可

### 斜平行投影

# – 斜平行投影

$$V_p = (V_{px}, V_{py}, V_{pz})$$

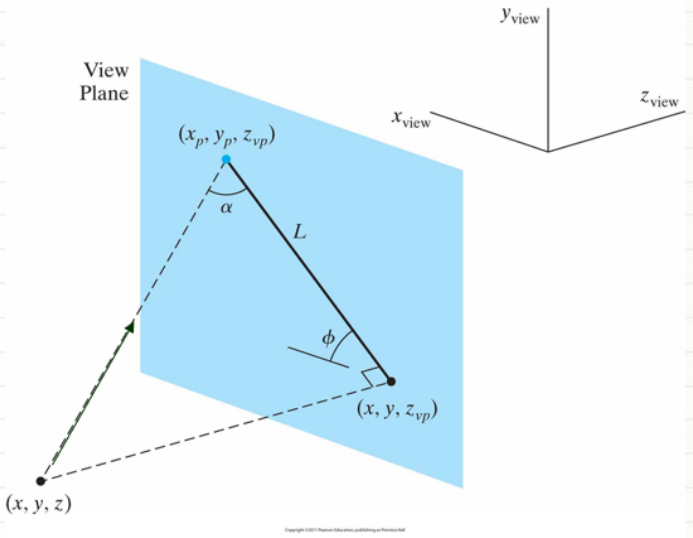
$$\frac{V_{py}}{V_{px}} = \tan \emptyset$$

$$\frac{x_p - x}{z_{vp} - z} = \frac{V_{px}}{V_{pz}}$$

$$\frac{y_p - y}{z_{vp} - z} = \frac{V_{py}}{V_{pz}}$$

$$x_p = x + (z_{vp} - z)\frac{V_{px}}{V_{pz}}$$

$$y_p = y + (z_{vp} - z)\frac{V_{py}}{V_{pz}}$$




$$M_{oblique} = \begin{pmatrix} 1 & 0 & -\frac{V_{px}}{V_{pz}} & z_{vp}\frac{V_{px}}{V_{pz}} \\ 0 & 1 & -\frac{V_{py}}{V_{pz}} & z_{vp}\frac{V_{py}}{V_{pz}} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

分类：

1. 正等轴测投影
2. 正二轴测投影
3. 正三轴测投影

特殊的斜投影：

1. 斜等测投影：α等于45°，生成的视图。所有垂直于投影平面的线条投影后长度不变
2. 斜二测投影：投影角满足tanα=2（63.4 °），生成的视图。垂直于投影平面的线条投影后得到一半长度

## 透视投影

投影点的计算：

1. 写出对象上的点到**投影中心（投影参考点）**的直线参数方程
2. 计算直线与投影平面的交点

透视投影类型：

1. 灭点：3D物体的一组平行线投影后收敛于一点，此点称为灭点
2. 主灭点：3D物体平行于坐标轴的平行线收敛产生的灭点**（最多3个）**

3. 一点、二点、三点透视
4. 如何控制主灭点数目？(投影平面与坐标轴的相交的数目)

### 3D场景观察体

1. 无穷平行管道
2. 矩形平行六面体
3. 对称的透视投影锥体
4. 斜透视投影棱台

## 3D裁剪

三维裁剪的裁剪窗口：是一个六面体，包括左侧面、右侧面、顶面、底面、前面和后面

### 平面方程裁剪

将一条直线段的端点坐标代入边界平面方程中，通过判断两端点的在视景体的内外来确定输出

### 编码裁剪（CS）

六位区域码

$b_6b_5b_4b_3b_2b_1$

编码原则:

左： if(x<xwmin) b1=1

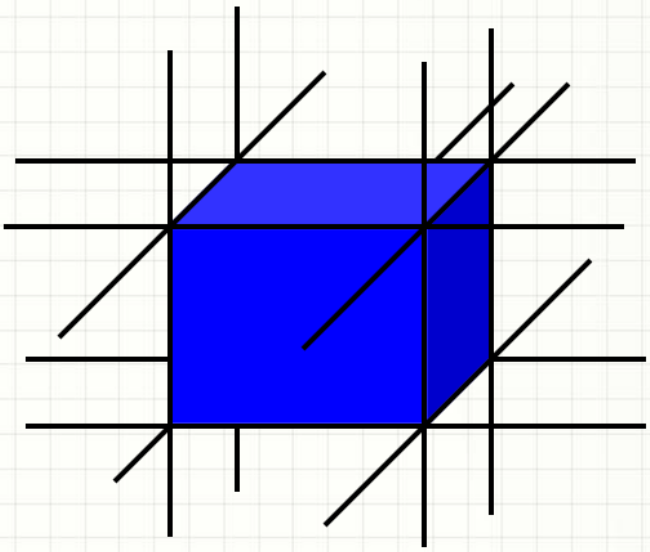
右： if(x>xwmax) b2=1

下： if(y<ywmin) b3=1

上： if(y>ywmax) b4=1

前： if(z<zwmin) b5=1

后： if(z>zwmax) b6=1



编码裁剪原则

- $C_1==0$ && $C_2==0$ → 保留
- $(C_1\&C_2)\neq 0$ → 删除
- $(C_1\&C_2) == 0$ → 求解直线与边界的交点

**参数方程裁剪（LB）**

LB算法描述

- 计算 $P_k$, $Q_k$, k=1~6
- 判断
  - $P_k$=0，表示直线平行于窗口某边界
    - $Q_k < 0$，直线在窗口外，剪裁
    - $Q_k > 0$，直线在平行边界内
  - $P_k$!=0, 用$Q_k/P_k$计算交点所对应的U值

- 计算参数$u_1$和$u_2$

$u_1$=Max{0, $Q_k/P_k$}，当$P_k < 0$

$u_2$=Min{1, $Q_k/P_k$}，当$P_k > 0$

- $u_1 > u_2$, 则直线在窗口外
- $u_1 > u_2$, 否则计算交点坐标，即得到剪裁后的线段的端点

**深度提示**

较远线段暗一些，较近线段亮一些

# 可见线与可见面识别

可见线用实线，不可见线用虚线（常用于工程制图）

绘制面时平面中的像素仅包含前向面的颜色图案

# 可见面判别算法

物空间算法和像空间算法一般都利用**排序和连贯性**来提高算法性能

- 排序主要用在深度比较
- 连贯性利用图形的规则性

## 物空间算法

在场景坐标系(图形定义空间)中实现。

将各对象和对象的某些部分彼此比较，通过有关几何运算，以确定哪些线或表面是不可见的。然后仅显示可见线，以此实现消隐。

算法精度较高，生成的图形可以放大多倍而仍然令人满意，适用于精密的工程应用领域

### 后向面判定

判断视线向量与面的法向量间点积的正负性

为正则为后向面

为负则为前向面

### 画家算法

画家算法，算法**同时运用物空间与像空间操作**

思想：若场景中任何多边形表面在深度上均不贯穿或循环遮挡，则各多边形的优先级顺序可完全确定

深度排序算法：

- 将表面按深度递减方向排序
- 由深度最大的表面开始，逐个对表面进行扫描转换

## 像空间算法

在显示图形的屏幕坐标系中实现

对投影平面或显示屏幕上的每一个像素位置逐点地检测其可见性。

算法比较粗糙，而且按像空间算法得到的画面在放大后往往不能令人满意。

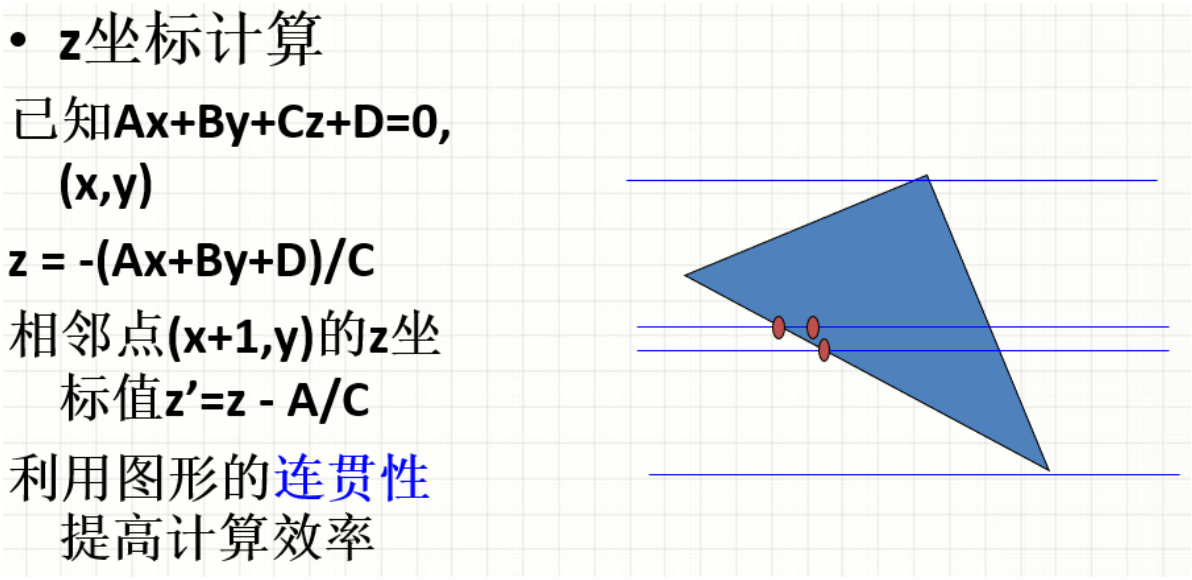
算法计算效率比较高，因为在光栅扫描过程中可以充分利用画面的连贯性等性质

## z-buffer

对投影平面上每个像素所对应的表面深度进行比较，选择离观察点最近表面上的点的信息决定像素点信息

最终结果：深度缓冲器中保存的是可见面的深度值，帧缓冲器中保存的是这些表面的对应属性值

**z-buffer对连贯性的应用**：




评价：

- 易于实现
- 空间需求大
- 每个象素点只能找到一个可见面，因此只能处理非透明表面
- 对于复杂场景的性能较好

## A-buffer

深度缓冲算法的改进算法，满足透明表面处理的需求。基本方法是扩展深度缓冲器，使每个像素位置对应一个表面链表

**A的含义**：

- 反走样(antialiased)
- 区域平均(area-averaged)
- 累积缓冲器(accumulation-buffer)

A-buffer的存储结构：

- **Data structure**

A缓冲器每个单元包含两个域：深度域、强度域。

深度域存储一个正的或负的实数，若为正表示只有一个表面对该像素点产生影响；反之表示多个面共同作用于该像素点。

强度域存储包括RGB强度分量、表面标识名、透明度、表面绘制参数、深度、覆盖度、指向下一表面的指针。


评价：

- 是深度缓冲器算法的改进
- 可以处理透明表面
- 可以对物体的边界进行反走样处理

### 扫描线算法

将3D对象各表面投影到投影平面，计算扫描线与投影多边形的相交区间。相交区间属于同一表面，直接显示该表面的强度信息；而当出现重叠区间时，采用深度测试确定采用哪个表面的强度信息。

# 样条表示

## 三维对象表示

三维对象表示方法通常可分为两类

- **边界表示** Boundary representations
  - 使用一组曲面/平面描述三维对象
  - 曲面/平面将物体分为内外两部分
  - 典型例子：多边形平面、样条曲面
- **空间分区** Space-partitioning representations
  - 用来描述物体内部性质
  - 将包含物体的空间区域划分成一组较小的、非重叠的、邻接的实体
  - 如：八叉树表示

**多边形表面**数据表分为两组进行组织：

- 几何表：顶点坐标和用来标识多边形表面空间方向的参数
- 属性表：指明物体透明度及表面反射度的参数和纹理特征

**细分方法**：

1. loop细分
2. Catmull-Clark细分
3. Doo-Sabin细分

# 样条相关概念

**曲线曲面的生成方法**：

- 给定一组数学函数
- 给定的一组控制点(数据点)

一旦给定函数，图形包将指定曲线方程投影到显示平面上，且沿着投影函数路径绘制像素位置

**样条**：通过指定一组控制点而生成平滑曲线的柔性带

- 插值样条曲线：选取的多项式使得曲线通过每个控制点
  - 自然三次样条插值、Hermite样条插值、Cardinal样条插值、Kochanek_Bartels样条插值
- 逼近样条曲线：选取的多项式不一定使曲线通过每个控制点
  - Bezier曲线、B-样条曲线

样条曲线在计算机图形学中的含义：

- 由**多项式曲线段**连接而成的曲线
- 在每段的边界处满足特定的**连续性条件**

**样条曲面**：使用两组正交样条曲线进行描述

**凸壳**：包含一组控制点的凸多边形边界

凸壳的作用：

- 提供了曲线或曲面与包围控制点的区域之间的偏差的测量
- 以凸壳为界的样条保证了多项式沿控制点的平滑前进

参数连续性：连接点的左右对应阶数导数值相等

几何连续性：连接点的左右对应阶数导数值成比例

# Bezier曲线

- **Bézier 构造公式**

  假定给出n+1控制点: $P_k=(x_k,y_k,z_k)$, k取值范围为0到n，这些坐标值用于合成向量 P(u),

$$P(u)=\sum_{k=0}^{n} P_k \times B_{k,n}(u)$$

- **Blending Functions合成函数**

$$B_{k,n}(u)=C(n,k)\times u^k \times (1-u)^{n-k}$$

$$C(n,k)=\frac{n!}{k!\times(n-k)!}$$

- **Bézier曲线特性**
  - **Bézier多项式次数＝控制点个数-1**
  - **Bézier曲线总是通过第一和最后一个控制点**

    $P(0)=p_0$, $P(1)=p_n$
  - 任何 **Bézier曲线总是落在控制点的凸壳内**

$$\sum_{k=0}^{n} B_{k,n}(u)=1$$

> 注意：$u \in [0,1]$

**一阶参数连续Bezier曲线的构造**:

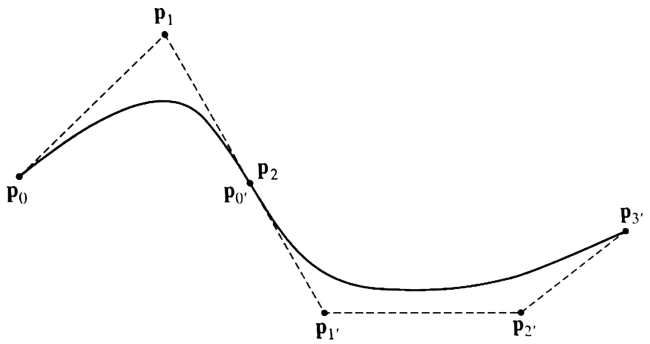



图 14.24　由两个 Bézier 曲线段形成的分段逼近曲线。设 $\mathbf{p}_0' = \mathbf{p}_2$，使 $\mathbf{p}_{1'}$、$\mathbf{p}_1$、$\mathbf{p}_2$ 共线，可以得到两条曲线段之间的0阶和一阶连续性

**Bezier曲面的构造**:

- 使用两组正交的**Bézier**曲线来设计**(m+1)*(n+1)**个控制点

$$P(u,v)=\sum_{j=0}^{m}\sum_{k=0}^{n}P_{j,k}B_{j,m}(v)B_{k,n}(u)$$

## 八叉树

- 三维形体的分解
- 对三维空间进行前后、左右、上下等分为**8**个小立方体，
- 小立方体单元均质，则停止分解；
- 小立方体单元非均质，需进一步分解为**8**个子立方体
- 直至所有小立方体单元均质，或已分解到规定的分解精度为止。

# 分形

**分形的特点**:

1. 每点具有无限细节
2. 对象整体和局部之间的自相似性

**分形图形的生成过程**：重复使用指定的变换函数作用于空间区域中的点的过程

**分形分类**:

- 自相似分形：组成部分是整个物体的收缩形式
- 自仿射分形：组成部分为不同坐标方向上的不同缩放因子形成
- 不变分形集：由非线性变换形成
    - 自平方分形
    - 自逆分形：由自逆过程形成。

**分形的维数**：描述分形对象细节的变化量，是对象粗糙性或细碎性的度量。 用$D = \frac{\ln n}{\ln(1/s)}$描述

其中：n是再分数目、s是缩放因子