



第1章 计算机图形学概述

1.1 计算机图形学及其研究内容

1.2 计算机图形学的相关领域和学科

1.3 计算机图形学的发展

1.4 计算机图形学的主要应用领域

1.5 计算机图形学当前的研究动态

习题





1.1 计算机图形学及其研究内容

1.1.1 计算机图形学

计算机图形学(Computer Graphics)是20世纪60年代以来发展迅速、应用广泛的综合性学科。世界各国的专家学者对计算机图形学有着各自的定义。德国的Wolfgang K Giloi给出的定义是：计算机图形学由数据结构、图形算法和语言构成。



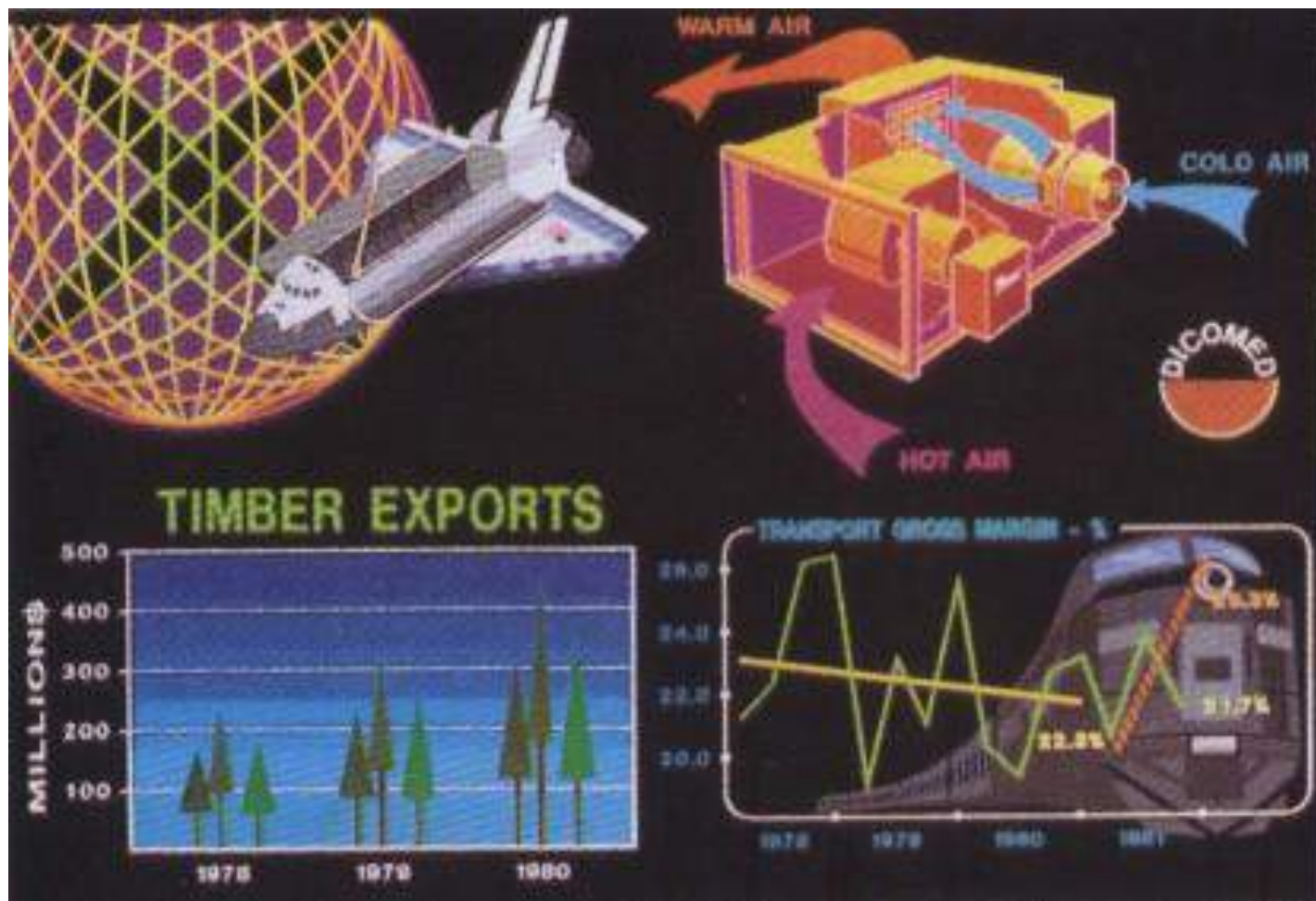


图 1.1 计算机图形学应用示例(DICOMED公司提供)





1. 图形

计算机图形学的研究对象是图形。广义的说,能够在人的视觉系统中形成视觉印象的客观对象都可称为图形。它既包括了各种几何图形以及由函数式、代数方程和表达式所描述的图形,也包括了来自各种输入媒体的图景、图片、图案、图像以及形体实体等。





2. 图形信息的特点

图形信息是一种重要的信息类型，它直接明了，含义丰富，具有以下特点：

1) 图形信息表达直观，易于理解

在科学技术高度发达的今天，图形信息显示出任何语言无法比拟的优越性，它能直接反映出客观世界变幻无穷的图像，供全人类所共享，不受语言限制。





2) 图形信息表示准确、精练

图形给人一瞬间把握整体的特点，它比文字更加简明精练，而文字要逐字逐句逐段联系起来才能理解，真是“一幅图胜过千言万语”。

3) 图形信息能“实时”地反应过程的变化规律

连续变化的图形信息能更“实时”地反映生产和科学研究与实验过程，并从中发现起决定作用的因素和关系。





4) 图形信息的信息量较大

“一幅图胜过千言万语”，这从另外一个角度也说明图形中包含的信息量较大，因此，图形如何在计算机中表示，也是计算机图形学研究的内容之一。





3. 图形在计算机中的表示

计算机中表示带有颜色及形状信息的图形常用以下两种方法:

1) 点阵法

点阵法通过枚举出图形中所有的点来表示图形，它强调图形由哪些点构成，这些点具有什么样的颜色，即点阵法是用具有灰度或色彩的点阵来表示图形的一种方法。在计算机中表示图形**最常用**的是点阵法。





2) 参数法

参数法用图形的形状参数和属性参数来表示图形。形状参数指的是描述图形的方程或分析表达式的系数、线段和多边形的端点坐标等。属性参数则包括颜色、线型等。





4. 计算机图形系统的概念

1) 计算机图形系统的组成

计算机图形系统是为了支持图形应用程序便于实现图形的输入、处理、输出而设计的计算机硬件和软件的组合体。没有绘图系统的支撑,就会使图形应用程序的编写极为困难,计算机图形学潜在的用途也难以开发。





计算机图形系统由硬件和软件两部分组成。计算机图形系统的基本物理设备统称为硬件，它包括主机及大容量外存储器、显示处理器、图形输出和图形输入设备。其中图形显示器、打印机、绘图机、键盘、数字化仪和光笔等供系统配置时由用户选用。



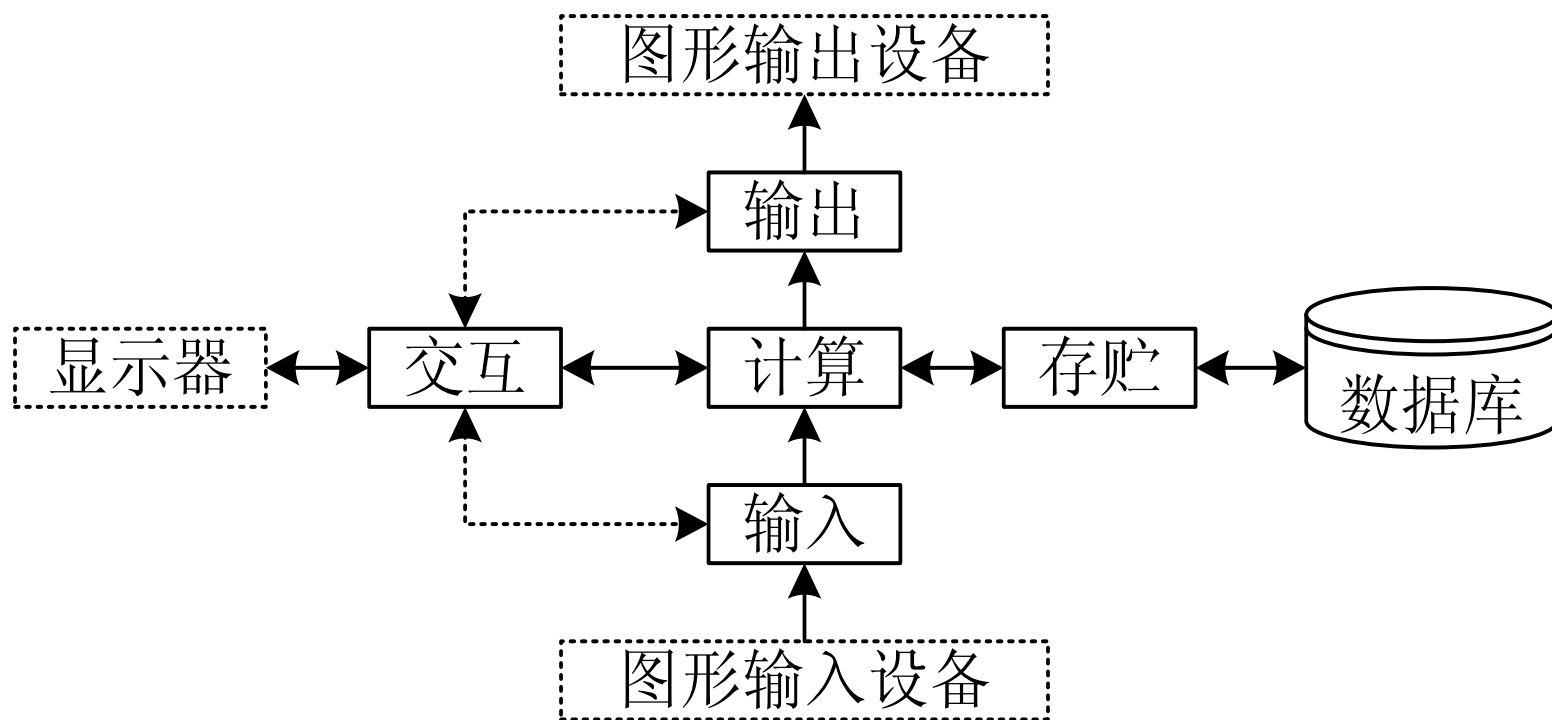


图 1.2(a) 计算机图形系统的功能



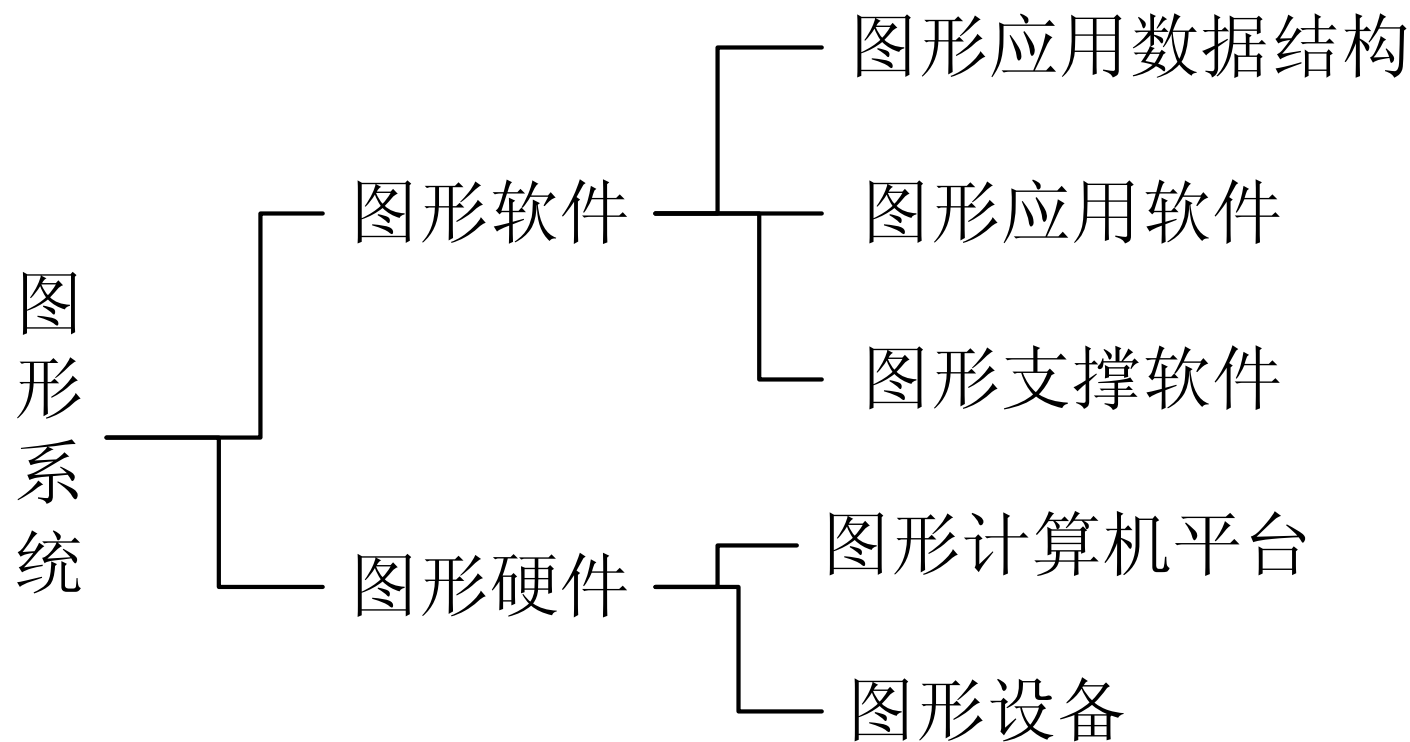


图 1.2(b) 计算机图形系统的结构



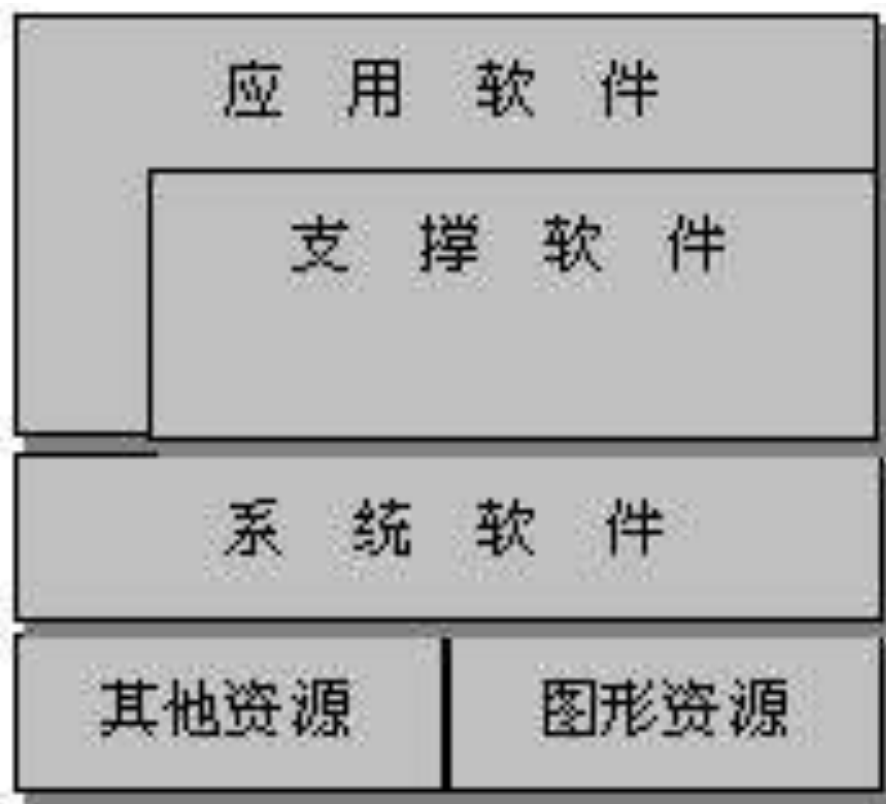


图 1.3 单主机模式计算机图形系统的软件组成





2) 计算机图形系统的工作方式

计算机图形系统的工作方式有被动式和交互式两种。

被动式主要以绘图机、打印机作为输出设备，通过编制一个图形程序，让计算机执行程序时控制绘图机或打印机绘制图形，在生成图形过程中，无法进行操纵和控制，若要修改所生成的图形必须修改图形程序或数据。

交互式图形系统则由设计人员(或用户)利用键盘、光笔、数字化仪、图形显示器等交互设备的有关功能，控制和操纵模型的建立和图形的生成过程。





3) 图形系统中三维物体输出的流水线

在计算机内描述、构造、修改二维或三维形体时，常用的几何形体表示方法有：线框模型、表面模型、实体模型等三种方法。下面以线框模型为例说明计算机图形系统中三维物体透视线架图输出的过程。

图形显示涉及到两个重要概念——窗口(window)和视区(viewport)。



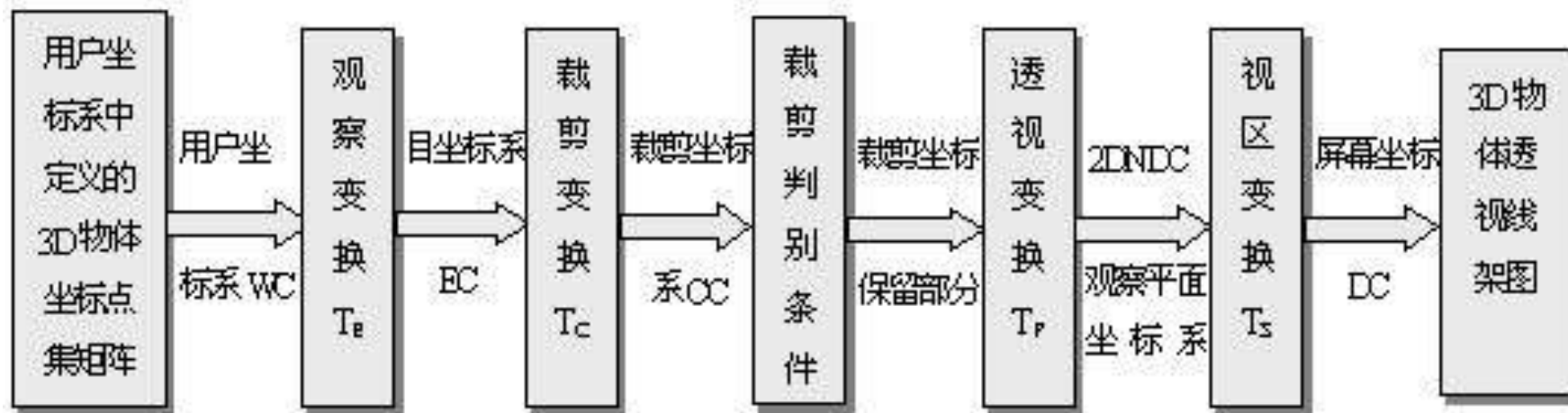


图 1.4 三维物体输出流水线图





1.1.2 计算机图形学的主要研究内容

除了理论和方法已经非常成熟的**基本图形元素生成算法**(也叫光栅图形学)和**图形变换**的内容之外,计算机图形学的主要内容还有**造型技术**、**真实感图形生成**及**人机交互技术**等三部分。





要在计算机屏幕上生成三维物体的一幅图像，首先必须在计算机中建立该物体的模型。构造这一模型的技术称为造型技术，包括**形体的表示**、**构造及运算**。最常用的是几何造型，即由一批几何数据及数据之间的关系来表示所要显示的形体，一般是规则形体。几何造型的可靠性及覆盖面是研究工作的重点。





在计算机辅助设计和制造(CAD/CAM)中,为了使形体的表示方法与工程师描述形体的习惯相衔接,并便于实现CAD/CAM 的集成,近年来出现了特征造型技术,即用形状特征、材料特征、公差特征等描述一个产品。随着动画技术的发展,出现了基于物理的造型技术。在计算机艺术中,为了表示不规则的形体,又出现了分数维造型和基于文法的造型等。几何造型的例子如图1.5所示,其中图(a)是工程图,图(b)是线框图,图(c)是实体模型描述的例子。



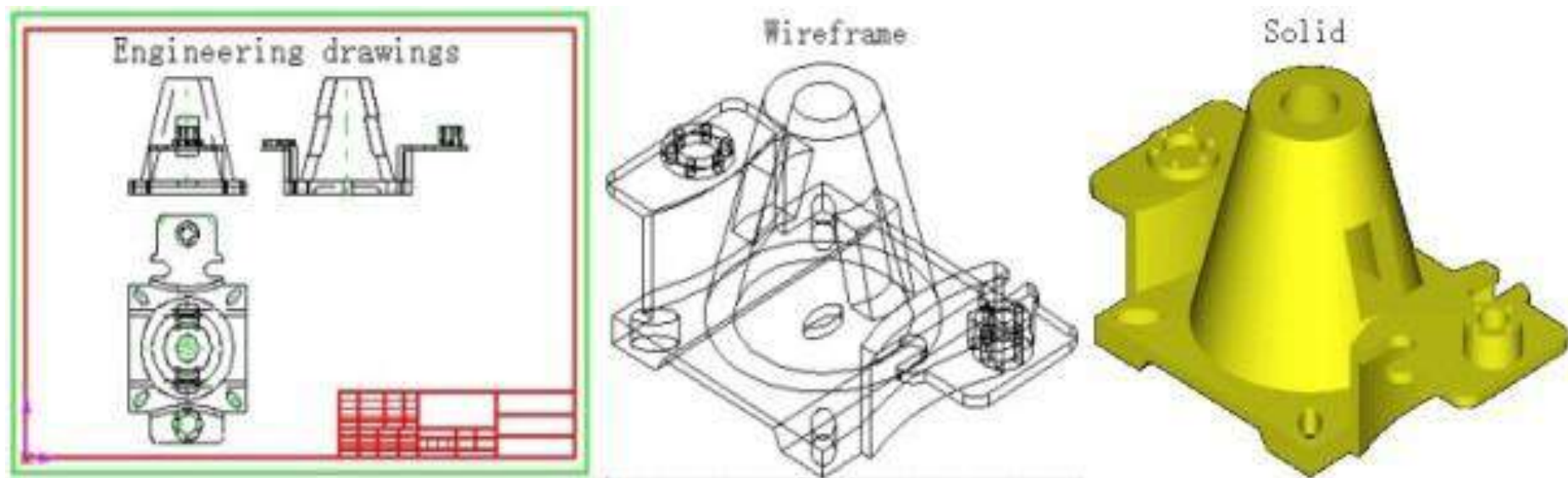


图 1.5 造型技术示例

(a) 工程图; (b) 线框图; (c) 实体模型图





除了不断提高计算机硬件的运算速度及图形软件的效率以外，并行计算是一个重要手段。图1.6是真实感图形生成示例。纹理映射技术的示例如图1.7所示。



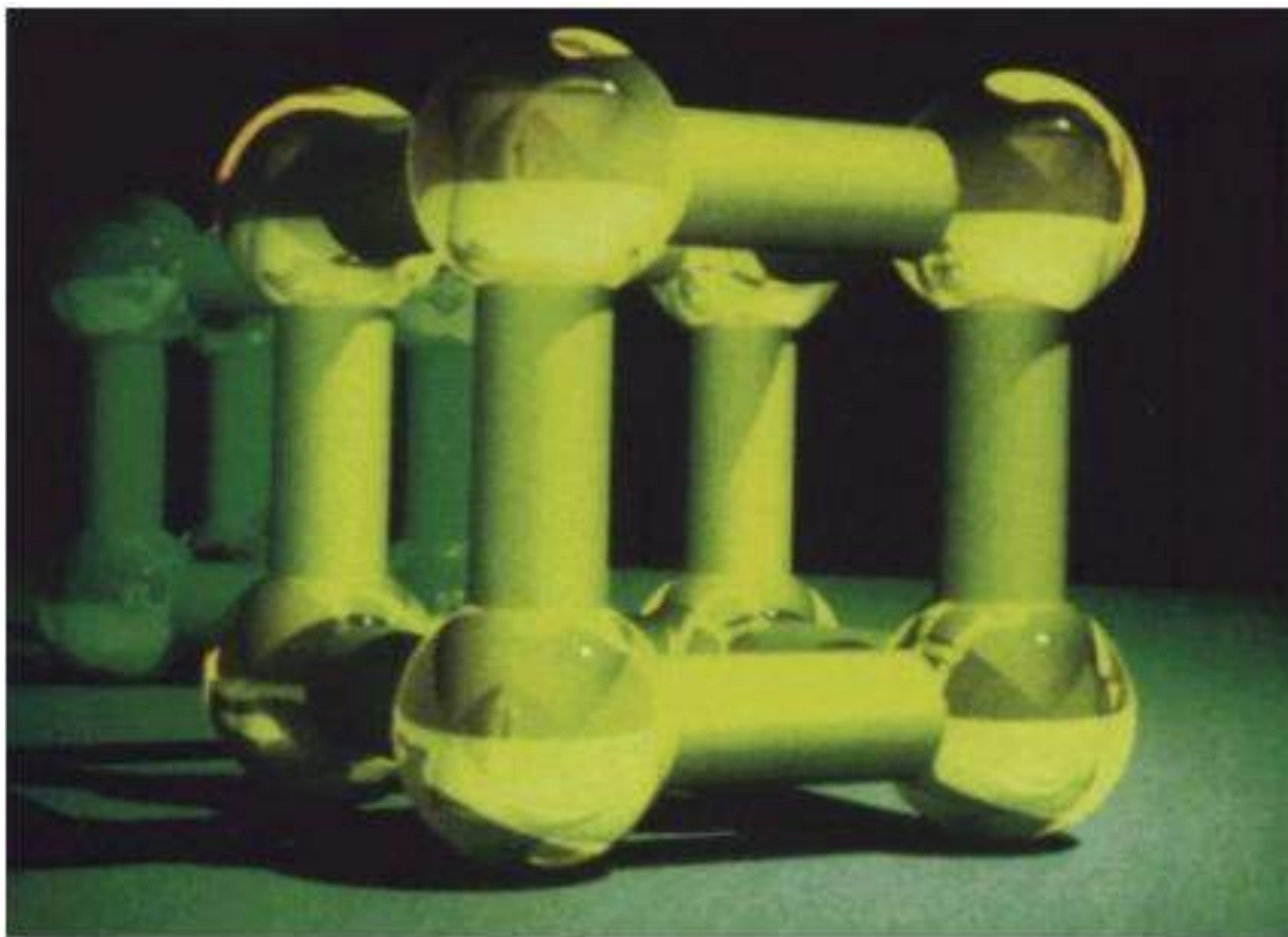
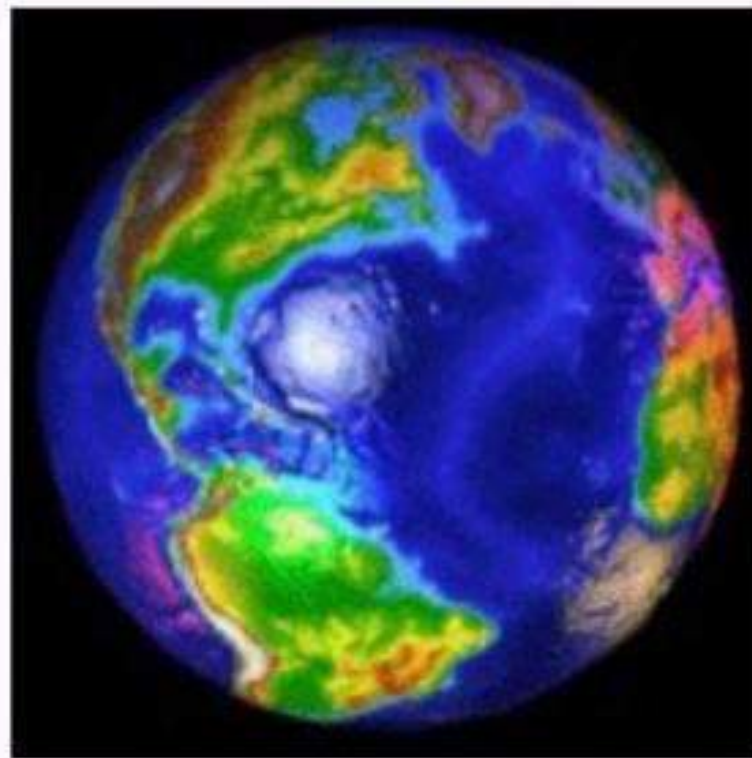


图 1.6 采用Whitted光线跟踪及光照明模型绘制的带反射和折射的球与柱体





(a)



(b)

图 1.7 纹理技术示例

(a) 带光照的纹理映射； (b) 用凹凸纹理技术绘制的地球





三维物体的造型过程和真实感图形生成过程都需要在一个操作方便、易学易用的用户界面下完成,其中包括:图元及造型方法的交互选择,形体、模型的交互操作,观察点、观察方向的交互设置,光照模型参数的交互选取,色彩的交互设定等。





1.2 计算机图形学的相关领域和学科

用计算机处理图形信息采用不同的方式和过程,使得图形图像处理技术的应用领域发展为五个联系密切的分支学科,即计算机图形学(CG: Computer Graphics)、数字图像处理(Image Processing)、模式识别(Pattern Recognition)、计算机视觉(Computer Vision)和计算机辅助几何设计(CAGD: Computer Aided Geometric Design)。它们之间既有一定联系,又有不同的研究目标、方法与技术,图1.8表示了它们之间的关系。



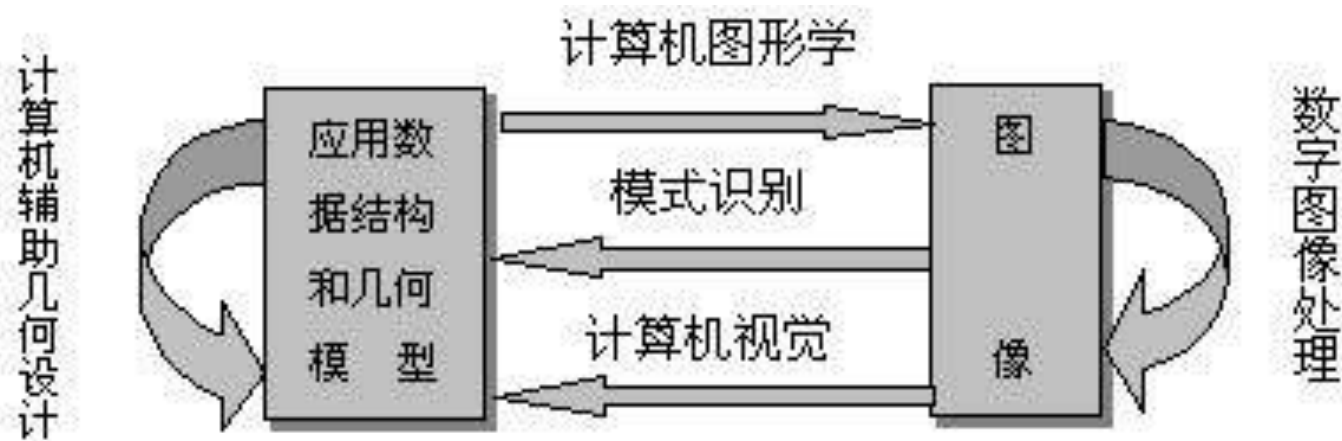


图 1.8 图形图像处理相关学科间的关系





数字图像处理将客观世界中原来存在的物体映像处理成新的数字化图像。如对照片图像扫描采样、量化、模 / 数转换后送入计算机，由计算机按应用的需要，对数字图像信息进行加工处理，从而改善图像的视觉效果。其主要处理内容有图像去噪、增强、复原、分割、提取特征、重建、识别以及存储、压缩编码和传输等。





模式识别(Pattern Recognition)研究的是计算机图形学的逆过程, 它主要讨论如何分析和识别输入的数字图像和图形, 并从中提取二维或三维的数据模型(特征)。





计算机视觉(Computer Vision)是研究用计算机来模拟生物外显或宏观视觉功能的科学和技术, 它模拟人对客观事物模式的识别过程, 是从图像到特征数据、对象的描述表达的处理过程。 其主要内容包括图像特征提取、 相机定标、 立体视觉、 运动视觉、 3D重建、 物体建模与识别、 距离图像分析等。





计算机辅助几何设计(CAGD-Computer Aided Geometric Design)着重讨论几何形体在计算机内的表示、分析和综合, 研究怎样方便灵活地建立几何形体的数学模型, 提高算法的效率, 在计算机内如何更好地存储和管理这些模型等。





1.3 计算机图形学的发展

1.3.1 计算机图形学的发展简史

自20世纪50年代以来, 计算机图形学的发展历程经历了50多年。 根据其发展的特点, 这50年可以分为酝酿yunniang期 (50年代)、 萌芽期 (60年代)、 发展期 (70年代)、 普及期 (80年代)和提高增强期 (90年代)等五个阶段。





1.3.2 硬件设备的发展

在计算机图形系统的硬件中，图形显示器是最关键的设备之一，它的发展也具有一定的代表性。20世纪60年代中期在计算机图形系统中使用的是随机扫描的显示器，它具有较高的分辨率和对比度，具有良好的动态性能。





1.3.3 图形软件的发展及软件标准的形成

随着计算机系统和图形硬件的发展, 计算机图形软件的开发、应用, 及其生成和处理图形的算法也都在迅速发展, 并在图形系统中占据越来越重要的地位。按其本身的特点和功能, 计算机图形软件系统可以分为三类:

1) 用现有的某种计算机语言写成的子程序库 (一般是有些软件人员[用户]开发的)

用户使用时按相应计算机语言的规定调用所需要的子程序生成各种图形。





2) 图形支撑软件（一般是开发环境本身提供的）

图形支撑软件由一组公用的图形函数(或子程序)所组成，它扩充了原有高级语言和操作系统的图形处理功能。给用户提供了描述、控制、分析和计算图形的语句，适用于用户设计有关图形方面的应用程序。如 Turbo C 中的 Graphics.h 等。





3) 专用的图形系统

对于某一种类型的设备，可以配置专用的图形生成语言。如果要求简单，可以采用在多功能子程序包的基础上加命令语言的方式。如果需要配置一个具有综合功能的较为复杂的图形生成语言，又要求有较快的执行速度，则应开发或配置一个完整的编译系统。比起简单的命令语言，它具有更强的功能；比起子程序包，它的执行速度较快，效率更高。但它的系统开发工作量大，且移植性较差。





计算机图形学所涉及的算法是非常丰富的，围绕着生成和表示物体的图形图像的**准确性、真实性与实时性**，其算法大致可分为以下几类：

(1) 基于图形设备的基本图形元素的生成算法，如用光栅扫描图形显示器生成直线、圆弧、二次曲线、封闭边界内的填充、填图案和反走样等（**光栅图形算法**）。

(2) 基本图形元素的几何变换、投影变换、窗口裁剪等（**图形变换**）。





(3) 自由曲线和曲面的插值、拟合、拼接、分解、过渡、光顺、整体或局部修改等（曲线、曲面生成）。

(4) 图形元素(点、线、环、面、体)的求交与分类以及集合运算（图形元素的布尔运算）。

(5) 隐藏线、面消除以及具有光照颜色效果的真实图形显示（真实感图形生成）。

(6) 不同字体的点阵表示，矢量中、西文字符的生成及变换。





- (7) 山、水、花、草、烟、云等模糊景物的生成（分数维图形生成）。
- (8) 三维或高维数据场的可视化。
- (9) 三维形体的实时显示和图形的并行处理。
- (10) 虚拟现实环境的生成及其控制算法等。





1.4 计算机图形学的主要应用领域

1. 计算机辅助设计与制造(CAD/CAM)

计算机辅助设计是计算机图形学最早、最重要、最活跃的一个应用领域，目前计算机辅助设计与制造(CAD/CAM)也是计算机图形学应用最为广泛的领域，计算机图形学理论、技术的产生和发展也与CAD/CAM密切相关。





可以说CAD/CAM已使传统的工程设计与产品制造发生了巨大变化,特别是人机交互式图形系统,可以将人的直观感觉和判断能力与图形系统十分有效地结合起来,再加上使用高效的方法库(包括优化设计等)、数据库技术等,更好地发挥人的才能、智慧和计算机的特长,显著地提高了设计质量,减少差错,缩短设计周期,降低成本,从而有效地提高了设计与制造的效率。





图1.9、图1.10、图1.11给出了传统的机械加工过程与计算机集成制造系统(CIMS)、快速自动成型制造(RPM-Rapid Prototyping Manufacturing)系统的比较。图1.12是建筑CAD中高层建筑的效果图。计算机图形学主要用于设计过程,尤其是工程和建筑系统,现在几乎所有的产品都是由计算机设计的。CAD/CAM方法现在已频繁地应用于建筑、汽车、飞机、轮船、宇宙飞船、计算机、纺织品和许许多多其它产品的设计与制造中。CAD/CAM在我国的机械、电子、铁道、建筑、服装设计等各种工程设计领域都有着广泛的应用,并取得明显的经济效益。



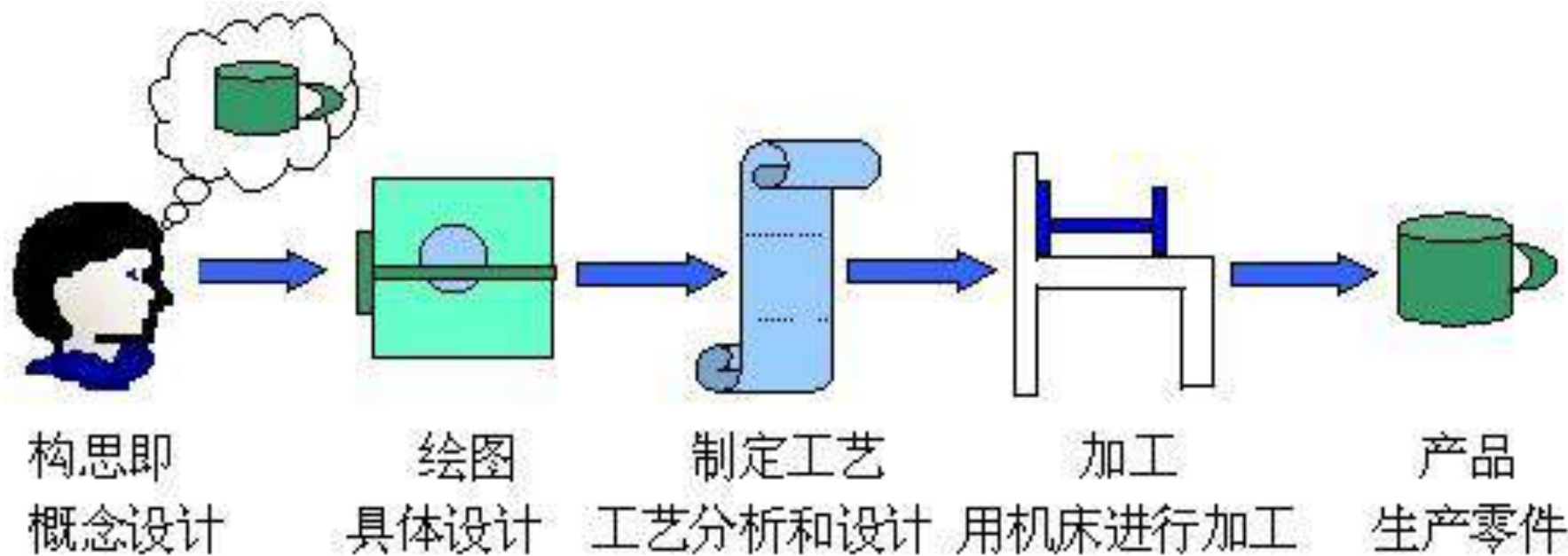


图 1.9 传统的机械加工过程



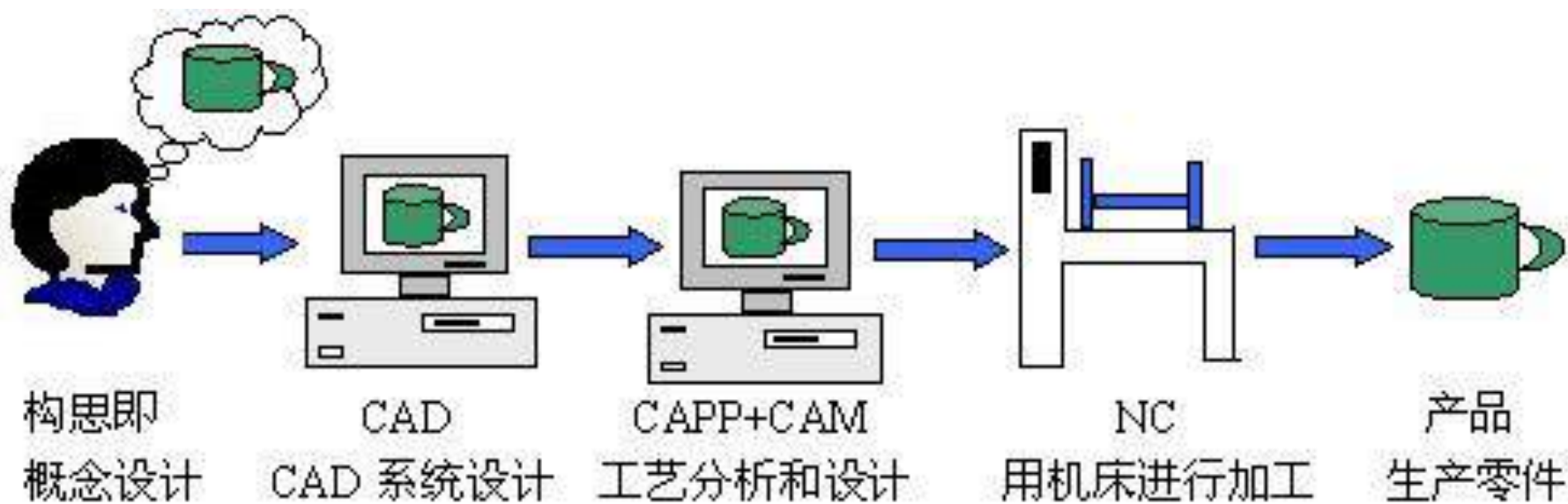


图 1.10 计算机集成制造系统(CIMS)



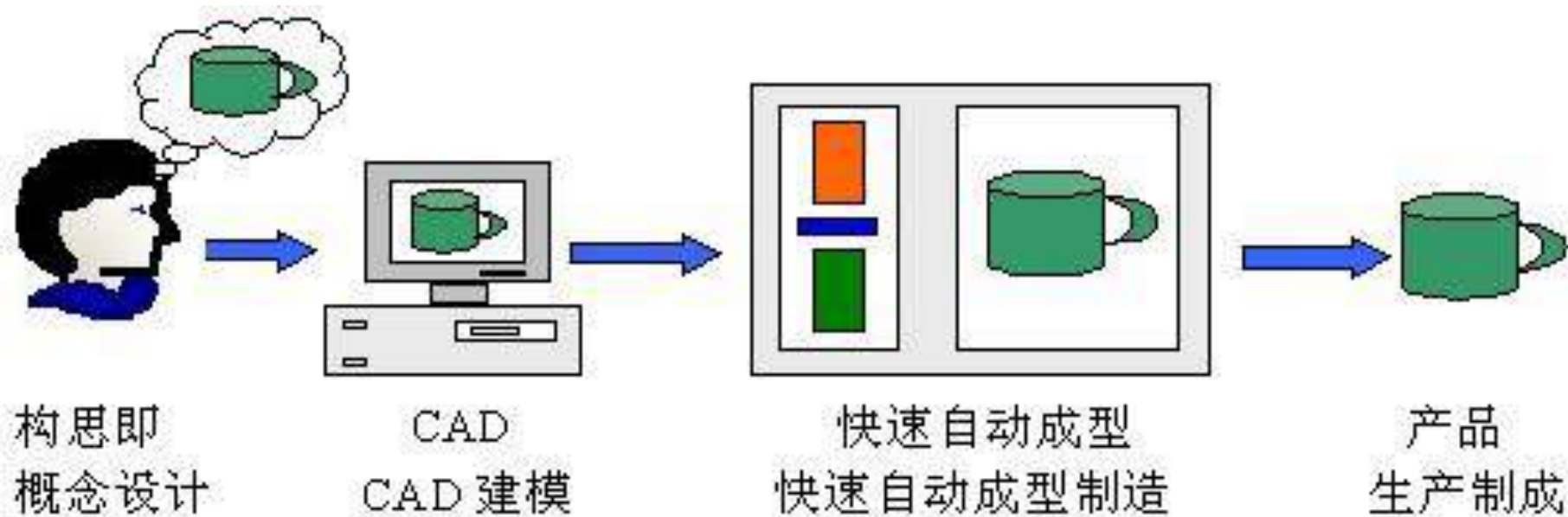


图 1.11 快速自动成型制造(RPM-Rapid Prototyping Manufacturing)系统





图 1.12 高层建筑效果图





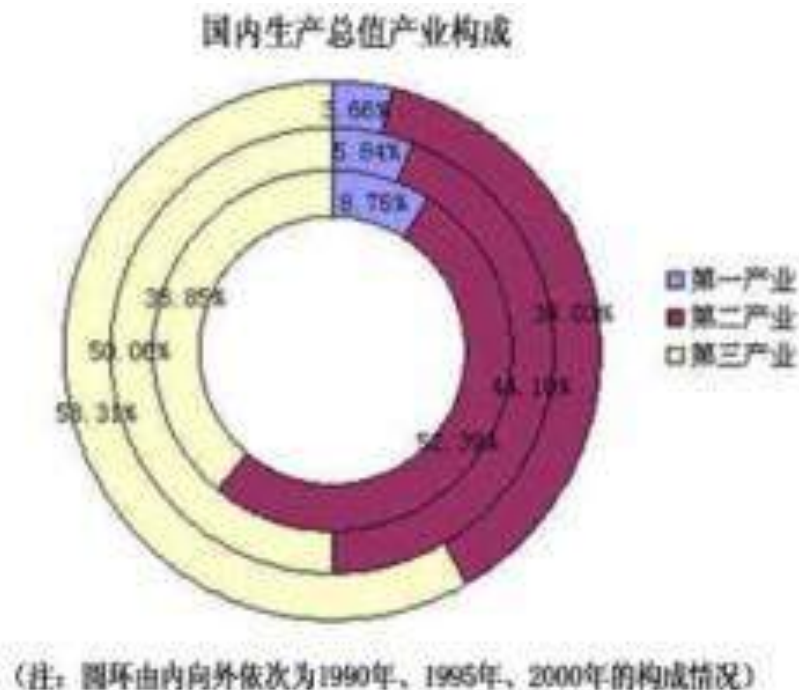
2. 计算机绘图(CD-Computer Drawing)

图形、图表和模型图等绘制是计算机图形学应用中的另一个重要方面。许多已经商品化的图形软件专门用于图形或者图表的生成。多数图形程序都具有二维或三维数据的处理能力,二维图形包括直方图、线条图、表面图或扇形图等;三维图形多用于显示多种形体间或者多种参数间的关系,如统计关系百分比图、分布关系图等。有的图形采用三维图形显示还可以表达数据的动态性质,如增长速度、变化趋势等。直方图、构成图、扇形图和表面图的例子如图1.13和图1.14所示。





(a) 3D直方图



(b) 构成图

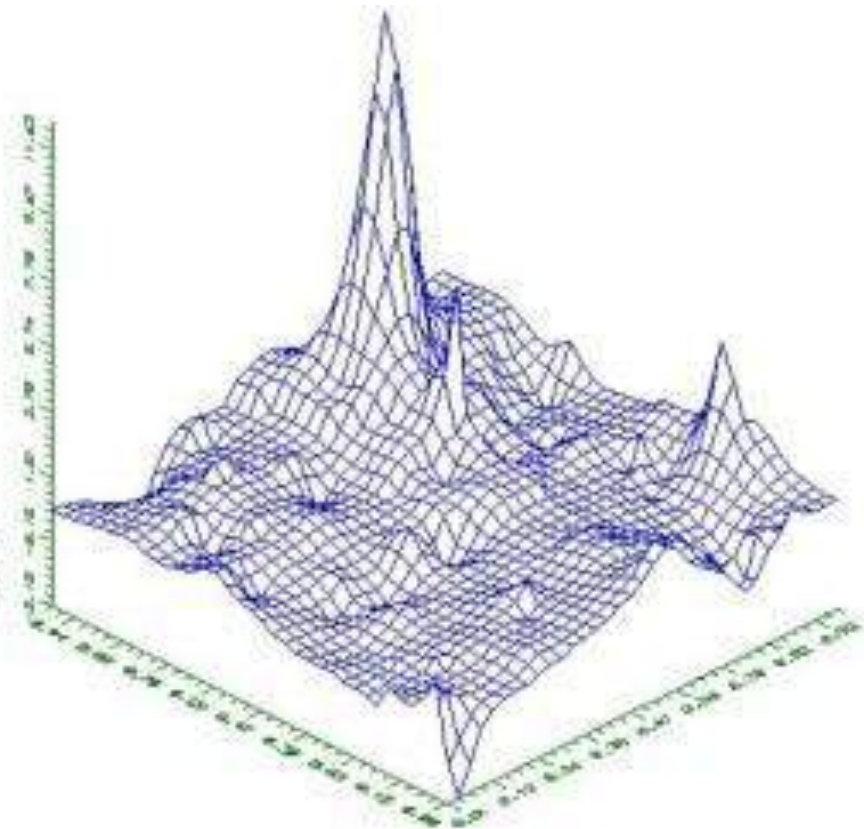
图 1.13 3D直方图与构成图

(a) 直方图; (b) 构成图





(a) 饼图



(b) 曲面的表面图

图 1.14 扇形图与表面图示例

(a) 扇形图； (b) 曲面的表面图





3. 科学计算可视化(ViSC-Visualization in Scientific Computing)

计算机图形学应用的一个典型例子是科学计算的可视化,当然, 科学计算可视化是计算机图形学的一个分支,也是当前计算机图形学的[研究热点之一](#)。传统的科学计算的结果是数据流,这种数据流不易理解也不易于检查其中的对错。





科学计算的可视化可以对[空间数据场](#)构造中间几何图素或用体绘制技术在屏幕上产生可见图像。近年来这种技术已用于有限元分析的后处理、分子模型构造、地震数据处理、大气科学及生物化学等领域。在测量数据的图形处理中,利用计算机制图精度高、速度快,用来绘制地形图、矿藏图、海洋图、天气预报图、人口密度分布图等。地理信息的绘制也是利用数据绘图的实例,它们用于显示不同的地理区域或者全球的数据统计信息。





例如,把来自各个气象观测站的数据经过专门的气象图处理程序集中起来,形成一种天气形势图、降雨图或者气压图。图1.15 是科学计算可视化的四个例子,图1.15(a)是二次方程 $5x^2+10y^2+2z^2+10yz+2y=c$ 的可视化图形,图1.15(b)是由自平方四元函数 $f(q)=\lambda q(1-q)(\lambda=1.475+0.906i)$ 生成的四维分形的三维投影,图1.15(c)是Fermat大定理 $n=5$ 的可视化图(Indiana大学计算机系Andrew Hanson绘制),图1.15(d)是以多种颜色混合绘制的数学函数曲线(Los Alamos国家实验室M.L.Prueitt)构成的图形。图1.16是地形地貌与海洋图。



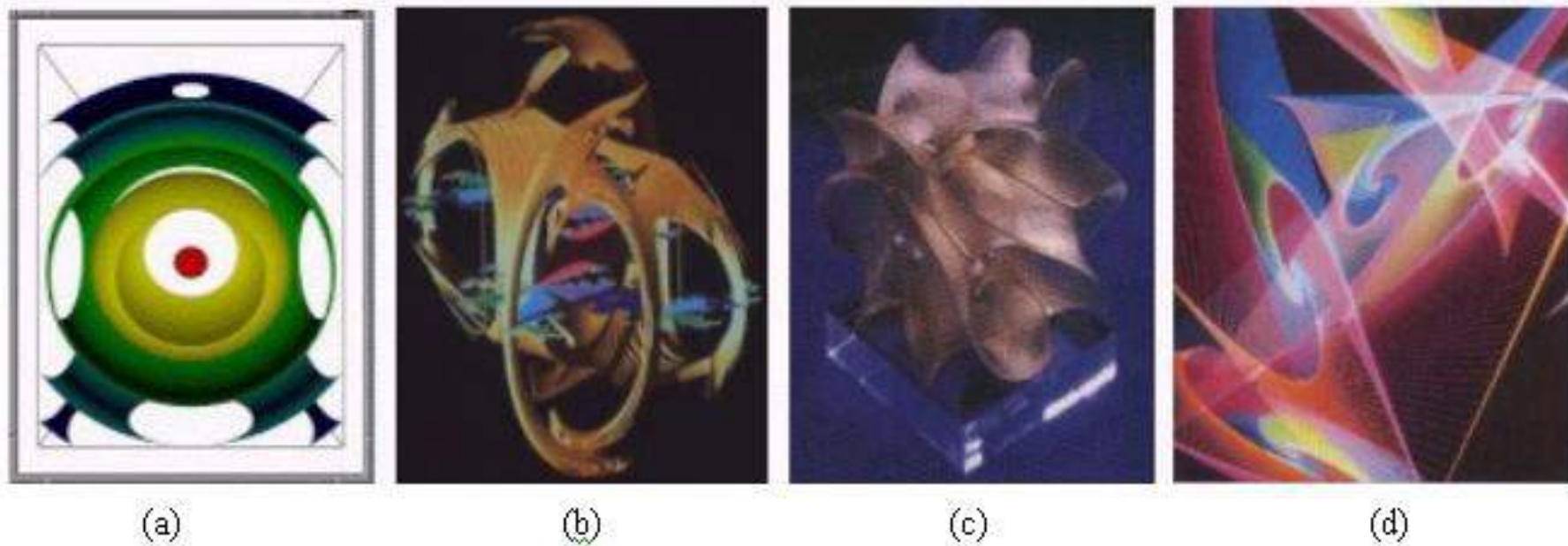
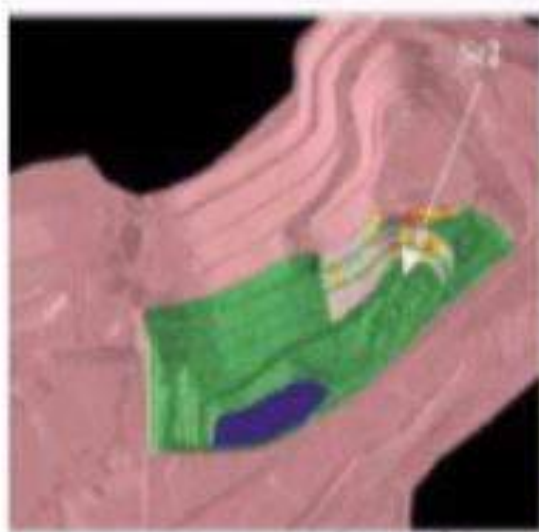


图 1.15 科学计算可视化的简单例子

(a) 二次方程可视化； (b) 4D分形的3D投影；

(c) $x^2+y^2=z^2$ 的可视化； (d) 数学函数曲线





(a)



(b)



(c)

图 1.16 地形地貌与海洋图

(a) 一个凹坑的地貌图； (b) Oslo部分地形地貌图； (c) 海洋温度的可视化





4. 计算机模拟与仿真(CS-Computer Simulation)

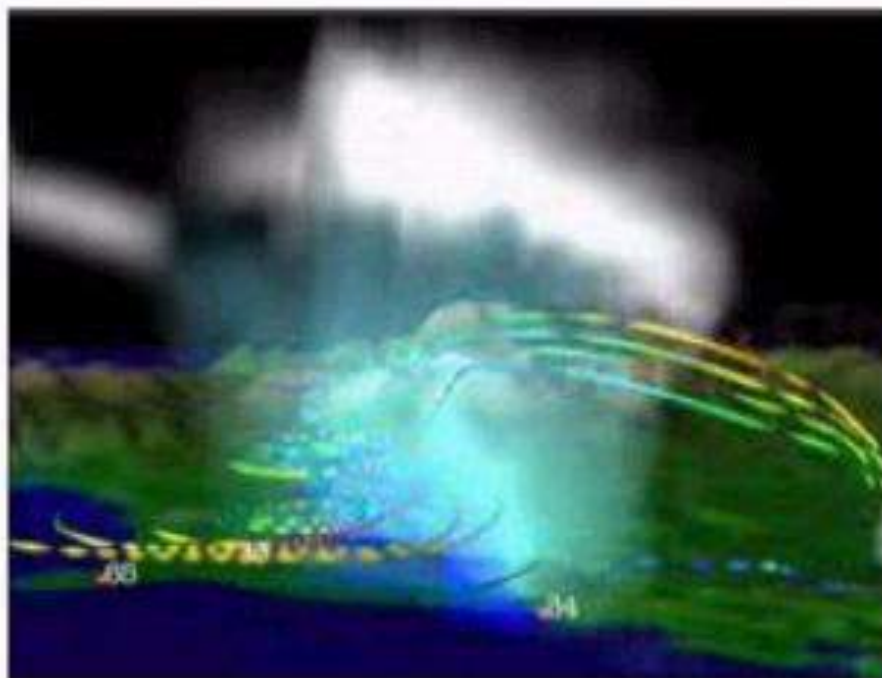
“模拟”是人类了解世界的手段，也是人类创造世界的方式，模拟技术化繁为简，将梦想变为现实，因而成为今日科学的基础，明日科学的趋势。计算机模拟与仿真是利用计算机模拟某个系统、某种效应和过程。把某个物理现象经数值模拟而数字模型化，再将该模型以图像的形式显示出来，通过模型再来研究物理现象或系统，这是进行系统分析和仿真的有效手段。图形显示在计算机模拟中有着重要作用。（如：战场模拟、飞行仿真）





计算机仿真可以用于研制产品或设计系统的全过程中，包括方案论证、技术指标确定、设计分析、生产制造、试验测试、维护训练、故障处理等各个阶段。当前计算机仿真的六大挑战性课题，包括核聚变反应、宇宙起源、生物基因工程、结构材料、社会经济和作战模拟等。未来十年的计算机仿真研究将集中于大规模复杂系统的仿真。战争指挥就是一类复杂系统仿真。未来的计算机仿真的主要特点是超大规模、模糊化、智能化。图1.17给出了气象过程和F16飞行仿真的示例。





(a)



(b)

图 1.17 计算机模拟与仿真示例

(a) 气象过程的模拟； (b) F16飞行仿真





5. 过程控制(PC-Process Control)

计算机过程控制是使用计算机系统实现生产过程的在线监视、操作指导、控制和管理的技术。计算机通过把与它连通的实时测试体、传感器等采集到的非图形信号,加工处理成图像,显示在屏幕上。图形显示清晰明了,操作人员观察审视操作过程中的各个环节状态情况,控制对象(如发电厂、化工厂等生产过程)操作既安全又方便,且可提高工作效率。在过程控制中,用户利用计算机图形学实现与其控制或管理对象间的相互作用。过程控制的应用领域十分广泛。例如产品和工程的设计、数控加工、石油化工、金属冶炼、矿井监测监控系统(如图1.18所示)、交通运输监控系统等。



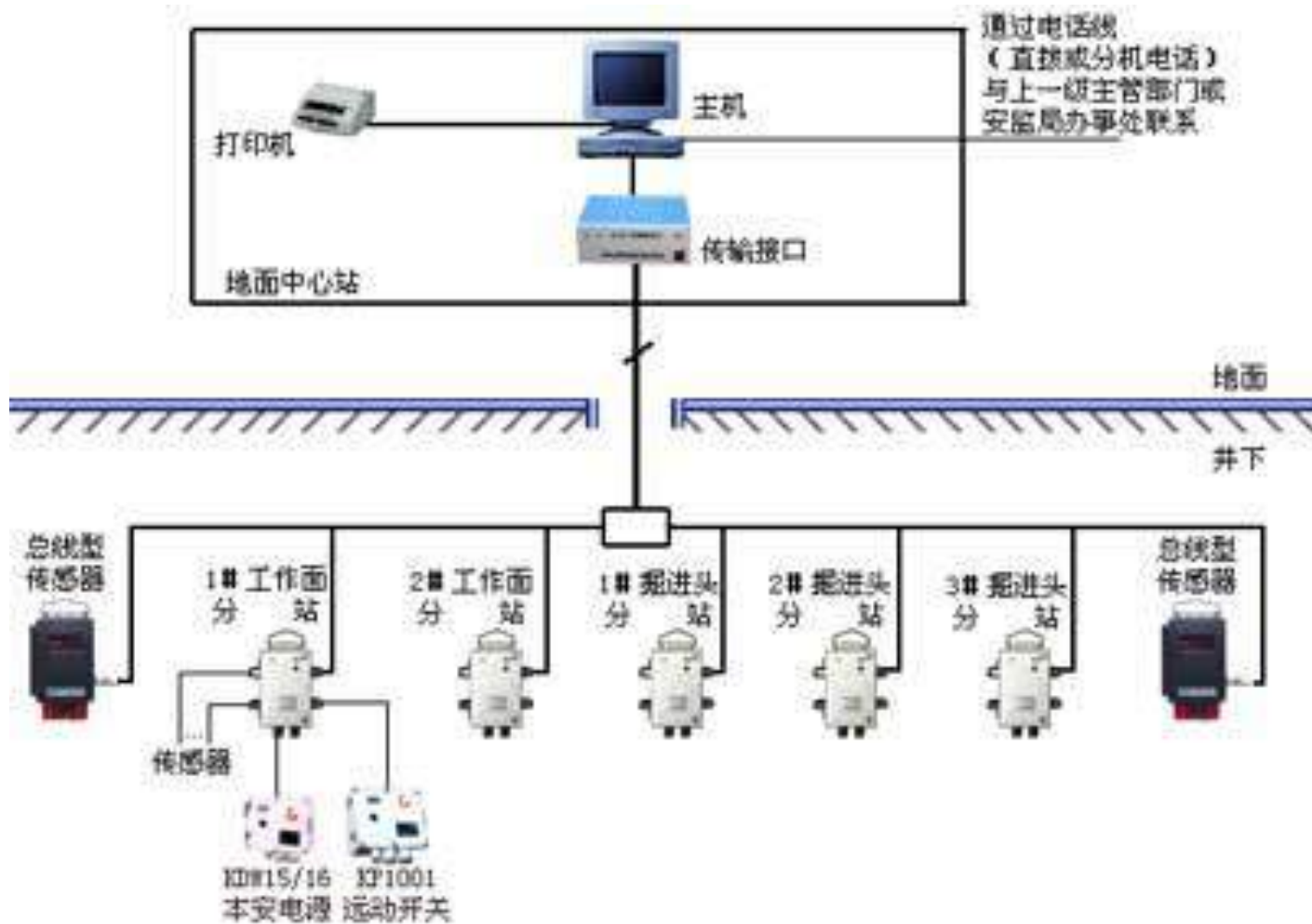


图 1.18 一个简单的矿井监测系统





6. 办公室自动化(OA)与电子出版技术(Electronic Publication)

办公自动化(OA)是一种解决特定行政办公类需求的信息系统。随着科学技术的迅速发展,在办公室繁琐的日常工作中,大量杂乱无章的文件数据分类、汇总、加工成不同要求的文字和图形报告,以及“电子邮件”通信等,都可以由价廉物美、易于操作,具有高质量的显示设备的微型计算机系统来完成。图形显示技术在办公自动化和事务处理中的应用,有助于数据及其相互关系的有效表达,有助于决策信息表达与传输,因而有利于人们进行正确的决策。






7. 计算机辅助教学(CAI)

计算机辅助教学系统利用图形显示设备或电视终端, 可以有声有色地生动地演示各个不同层次的教学内容, 让学生(用户)使用人机交互手段, 进行学习和研究, 绘图或仿真操作, 使整个教学过程直观形象, 有利于加深理解所学的知识, 并可自我考核打分。







目录

- 第一章 绪论
- 第二章 计算机绘图系统的硬件配置
- 第三章 图形函数和基本图形元素生成算法
 - § 3-1 图形函数
 - § 3-2 基本图形元素的生成算法
- 第四章 窗口变换与图形裁剪
- 第五章 图形变换与立体真实感显示
 - § 5-1 二维图形的变换
- 第六章 交互技术
- 第七章 计算机绘图技术应用举例
- 第八章 微型机通用绘图软件-Auto-CAD 简介
- 第九章 Windows 图形程序设计
- 第十章 C 语言程序运行环境

退出

$x_0=0, y_1=0, \quad x_8=8, y_8=4;$

$\epsilon = 1/\max(|x_8-x_0|, |y_8-y_1|) = 1/8$

$\therefore \epsilon \Delta x = (x_8-x_0)/8 = 1;$

$\epsilon \Delta y = (y_8-y_1)/8 = 0.5;$

$x = x_1 + 0.5 = 0.5, y = y_1 + 0.5 = 0.5$

$\therefore x_{i+1} = x_i + \epsilon \Delta x, y_{i+1} = y_i + \epsilon \Delta y$

迭代过程为: 演示

生成的直线

(1) (0, 0)

(2) (1, 1)

(3) (2, 1)

(4) (3, 2)

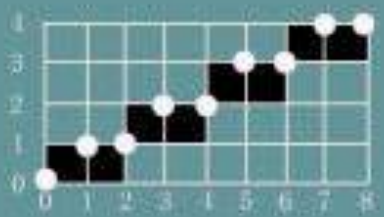
(5) (4, 2)

(6) (5, 3)

(7) (6, 3)

(8) (7, 4)

(9) (8, 4)



上一页 下一页

图 1.19 一个《计算机图形学》课件示意图





8. 计算机动画(Computer Animation)

随着计算机图形学技术的迅速发展,计算机在动画中的应用也不断扩大,计算机动画的内涵也在不断扩大。计算机动画发展到今天,主要分为两个阶段(或分为两大类),这就是计算机辅助动画和计算机生成动画。计算机辅助动画(computer-assisted animation),也叫“二维动画”,计算机生成动画(computer-generated animation),也叫“三维动画”。图1.20是“变形”(Morpher)的图形处理方法示例。



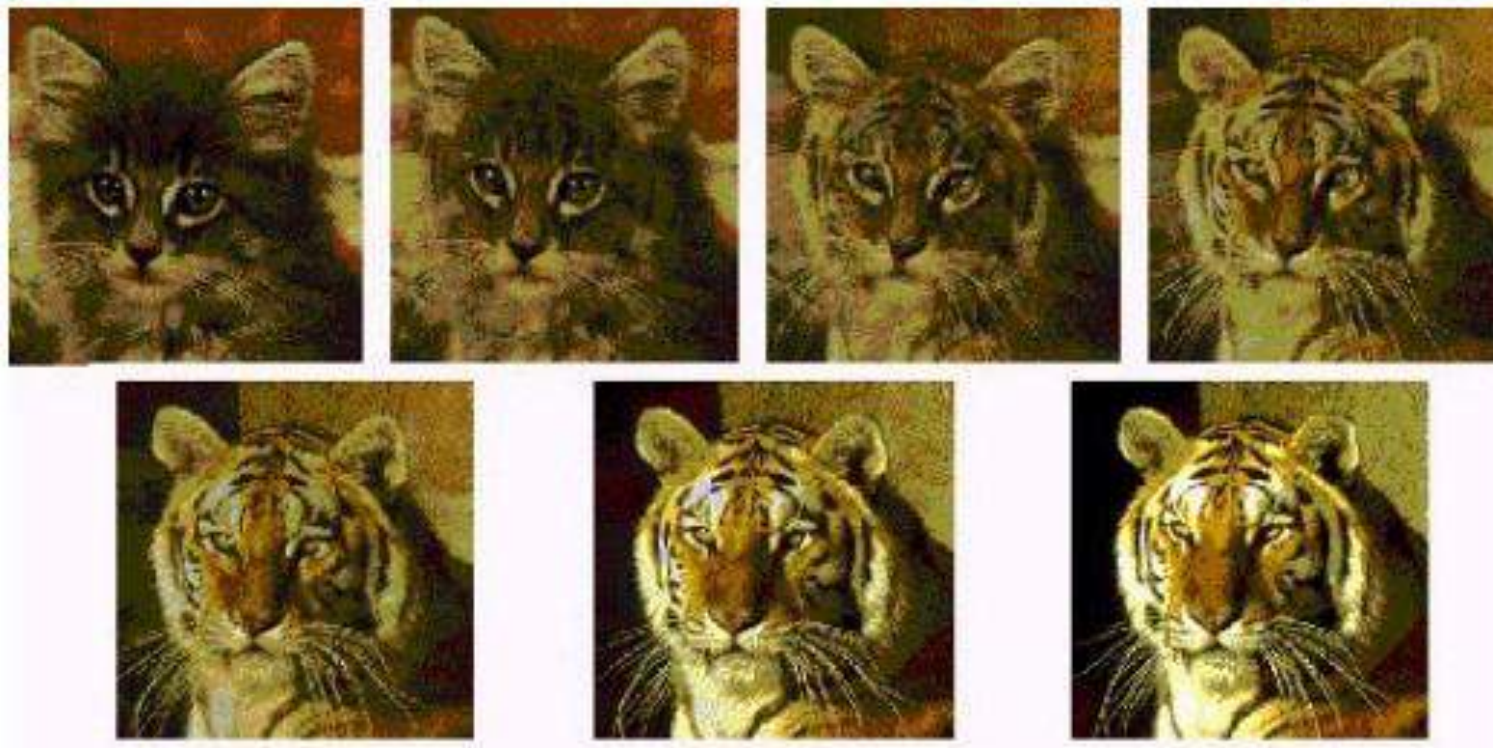


图 1.20 “变形” (Morpher)的图形处理方法示例





9. 计算机艺术(Computer Art)

计算机艺术是科学与艺术相结合的一门新兴的交叉学科。在设计领域,对改变、更新传统的设计思想和方法,提高产品设计质量,缩短设计周期,提高设计的艺术性和科学性,加速产品更新换代都有重要意义。图 1.21 是德国Magdeburg大学的Oliver Deussen绘制的素描树(Siggraph'2000)。



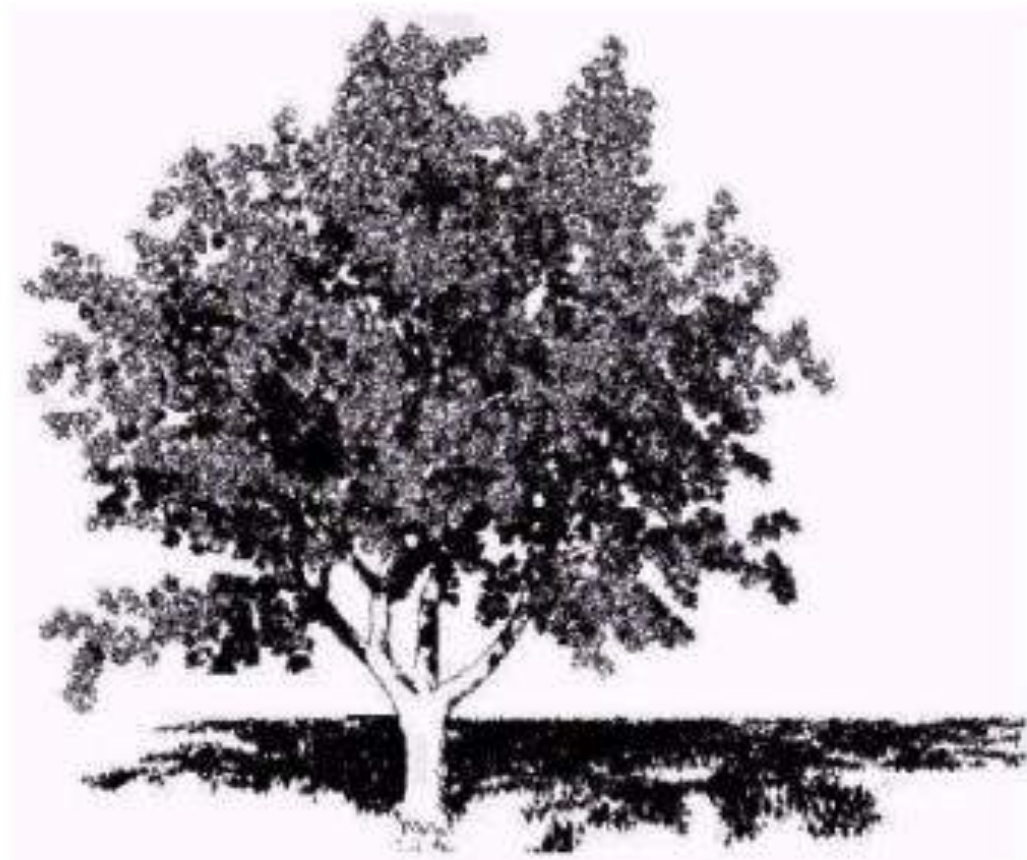


图 1.21 Oliver Deussen绘制的素描树(Siggraph'2000)





10. 人体造型与动画(Human Body Modeling and Animation)

用计算机构造人体模型，有着非常广阔的应用前景。
人机工程中需要考察人和机器以及周围环境的关系，工业设计中要使生活用具的造型适应人的生理和心理特征。服装设计中要将人体作为效果分析的对象等。





图1.22(a)是智能虚拟环境中人沿路径行走的动画，图1.22(b)是服装设计中模特的服装效果分析图。目前国内外不少单位正在研制人体模拟系统，这使得在不久的将来把历史上早已去世的著名人物重新搬上屏幕成为可能。





(a)

(b)

图 1.22 人体造型与动画示例

(a) 智能虚拟环境中沿路径行走动画； (b) 服装效果





11. 用户界面(User Interface)

介于用户与计算机之间,完成人与计算机通信工作的部件称人机界面(HCI-Human Computer Interface),它由软件部分和硬件部分组成。随着计算机技术以及图形学技术的发展,人机界面从最原始的由指示灯和机械开关组成的操纵板界面,过渡到由终端和键盘组成的字符界面,并发展到现在基于多种输入设备和光栅图形显示设备的图形用户界面 (GUI-Graphical User Interface)。典型的图形用户界面包含一个窗口管理程序、菜单显示和图符等。

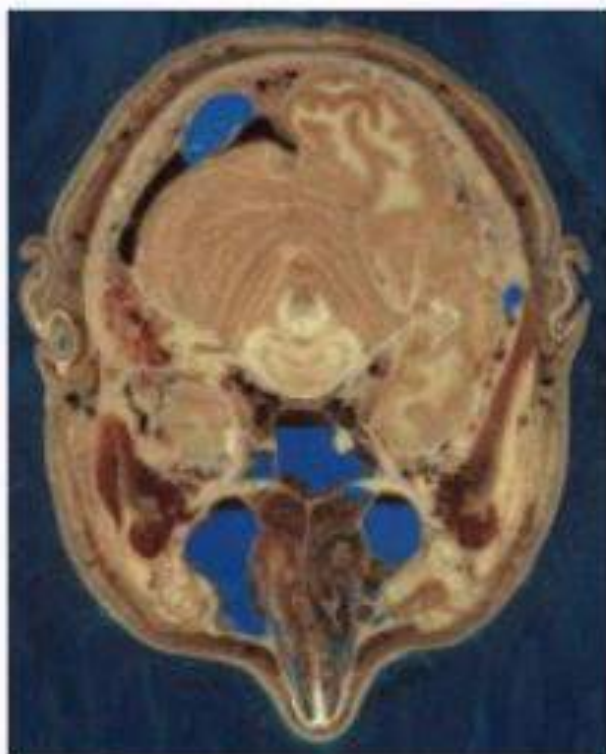




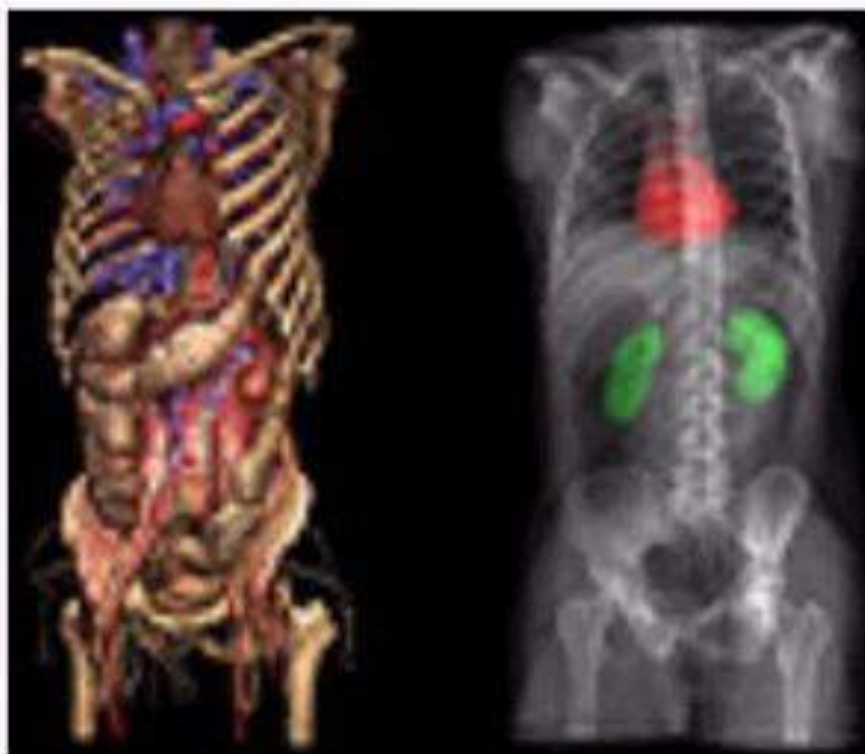
12. 医疗卫生方面的应用

计算机图形学在医疗卫生方面的应用包括：用来做病历的检索和统计以及医疗方案的研究；用显示设备显示病历，显示各种药物的剂量、性能；对某种病的治愈率作统计分析；对病人的医疗方案(如放射线照射)进行研究，以提高治疗效率；计算机辅助手术(Computer Aided Surgery)；远程医疗/手术系统等。医学上还往往结合[图像处理](#)和[计算机图形学](#)来建模和研究物理功能。例如设计人造肢体、计划和练习手术等。其应用示例如图1.23所示。





(a)



(b)

图 1.23 计算机图形学在医学中的应用

(a) 数字虚拟人头部横截面； (b) 由数字虚拟人重建的人体内部模型





1.5 计算机图形学当前的研究动态

1.5.1 真实感图形显示

从20世纪80年代初开始，真实感图形生成与显示技术一直是计算机图形学研究的前沿领域，而且发展很快。真实感图形是一种光栅图形。

真实感图形生成(Realistic Graphics Generation)是在计算机图形系统中生成具有色彩、纹理、阴影、层次等真实感的三维空间物体图形的技术。 它还可以称作真实感图形综合。





其目的是对于空间的各种物体和自然景物, 利用计算机图形生成技术产生就像拍出的照片一样的真实感效果图。为了产生图形的真实感, 一般需要解决以下几方面的图形综合技术问题:

- (1) 利用消隐技术在图形中消除在特定观察点看不见的物体或部分物体, 从而产生空间物体的层次感;
- (2) 利用纹理映射技术在物体表面生成各种各样的纹理, 以增强物体的质感;





(3) 利用光照模型、光线跟踪、辐射度技术尽可能精确地模拟光源照射的物理效果，使空间物体具有像拍照片一样的光照效果和明暗层次；

(4) 模拟透明物体的效果；

(5) 利用图形保真技术在显示设备有限的离散精度范围内尽量保持图形具有自然的光影过渡和连续性。



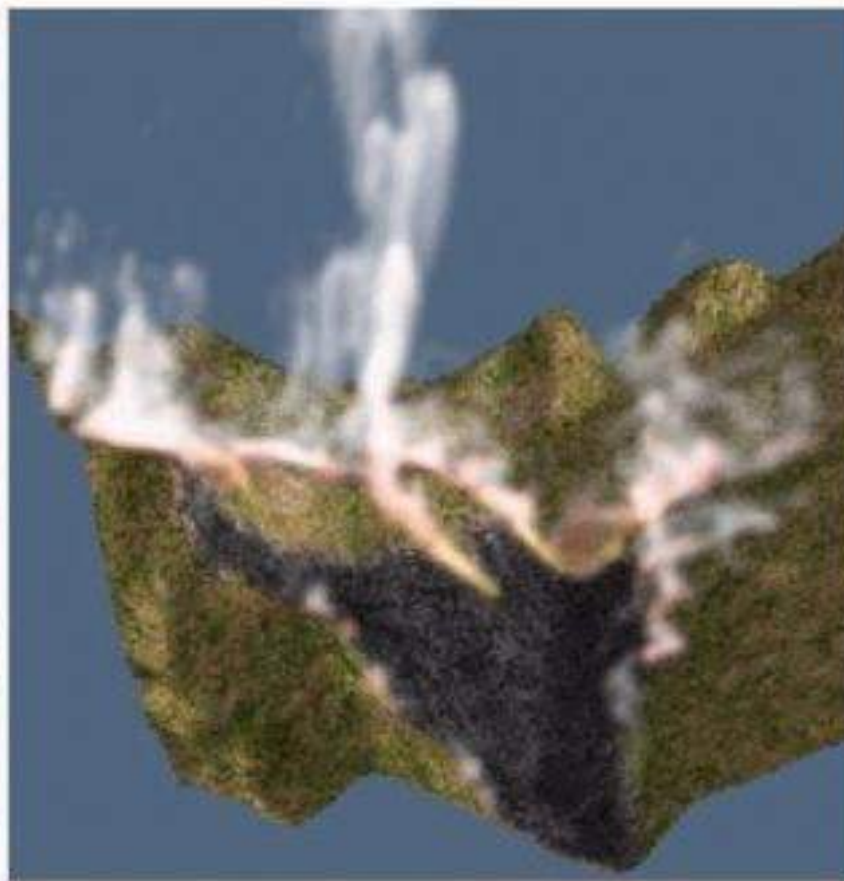


真实感图形的生成一般需经历场景造型、取景变换、视域裁剪、消除隐藏面、纹理映射、可见面光亮度计算等步骤。图1.24(a)是用辐射度方法生成的Chartres教堂的侧廊(3Deye公司, J.Walace等), 图1.24(b)是森林大火的计算机模拟。





(a)



(b)

图 1.24 计算机真实感图形显示实例

(a) Chartres教堂(3Deye公司J.Walace等);

(b) 森林大火的模拟





1.5.2 人机交互技术

人机交互技术(Human-computer Interaction Techniques)是通过计算机输入/输出设备,以有效的方式实现人与计算机对话的技术。人机交互技术是计算机用户界面设计中的重要内容之一。它与认知学、人机工程学、心理学等学科领域有密切关系。





常用交互技术(计算机图形学研究的内容)可以分成如下几类:

- (1) 构造技术。
- (2) 命令技术。
- (3) 选取技术。
- (4) 直接操纵技术。





1.5.3 计算机动画

计算机动画(Computer Animation)是利用计算机生成一系列可供实时演播的画面的技术。它可辅助传统卡通动画片的制作,也可通过对三维空间中虚拟摄像机、光源及物体运动和变化(形状、色彩等)的描述,逼真地模拟客观世界中真实的或虚构的三维场景随时间而演变的过程。





计算机动画中运动控制和描述的技术可归纳为三个方面:

- 动画控制方法。
- 动画控制设施。
- 动画控制的层次结构。





1.5.4 与计算机网络技术的结合

计算机网络与多媒体技术的迅速发展,使地理上相隔千万里的人们能够通过互联网(Internet)交换信息,实现信息共享,由此出现了各种应用网络化的趋势。信息的载体称为媒体,它包括文字、声音等,图形、图像是其中最重要的两种。 (如: Web GIS)





1.5.5 科学计算可视化

科学计算可视化(Visualization in Scientific Computing)是将科学计算过程中及计算结果的数据转换为图形及图像显示在屏幕上的方法与技术。它综合运用计算机图形学、数字图像处理、计算机视觉、计算机辅助设计及人机交互技术等几个领域中的相关技术。





科学计算可视化按其实现的功能,可以分为三个层次:

(1) 结果数据的后处理, 即对计算数据或测量数据进行脱机处理, 然后用图像显示出来, 这一层次的功能对计算能力的需求相对较低。

(2) 中间数据或结果数据的实时跟踪处理及显示。

(3) 中间数据或结果数据的实时跟踪处理、显示及交互控制。 这一层次的功能不但能对数据进行实时跟踪显示, 而且还可以交互式地修改原始数据、边界条件或其它参数, 以使计算结果更为满意。





为了实现这三个层次的功能，科学计算可视化的主要内容研究内容有以下几个方面：

- (1) 标量、 矢量、 张量场的显示;
- (2) 数值模拟和计算过程的交互控制和引导;
- (3) 面向图形的程序设计环境;
- (4) 高带宽网络及其协议;
- (5) 用于图形和图像处理的向量、 并行算法及特殊硬件结构。





科学计算可视化的过程有以下四个步骤：

- (1) 数据预处理(数据操纵);
- (2) 可视化映射;
- (3) 绘制;
- (4) 显示。

它的模型如图1.25所示。



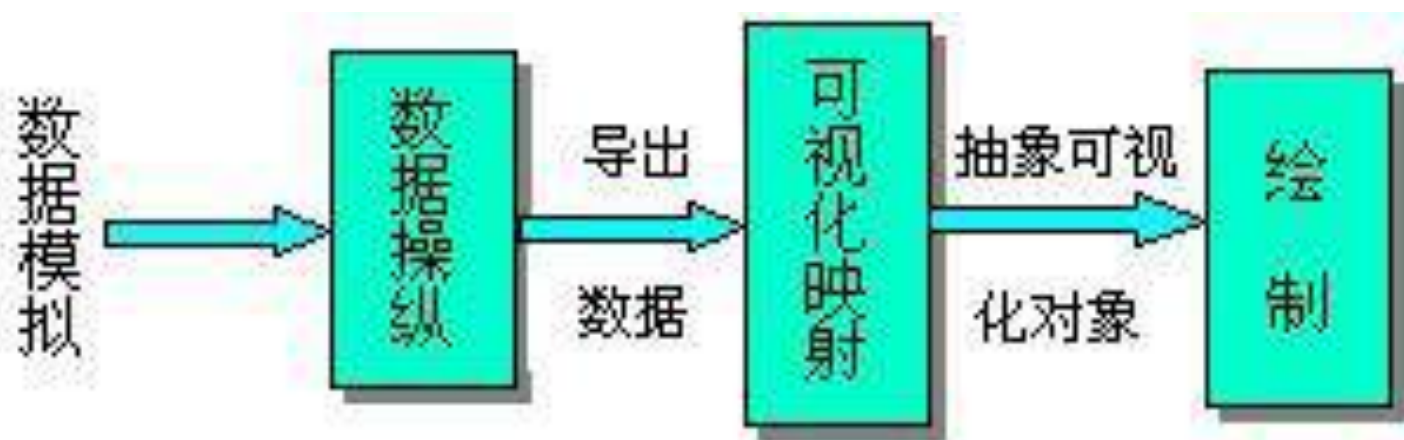


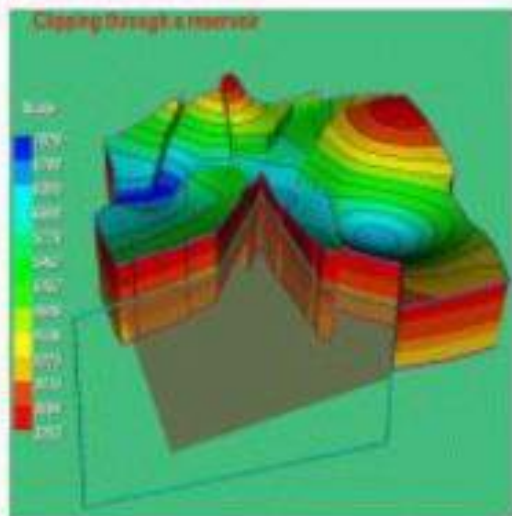
图 1.25 可视化过程模型



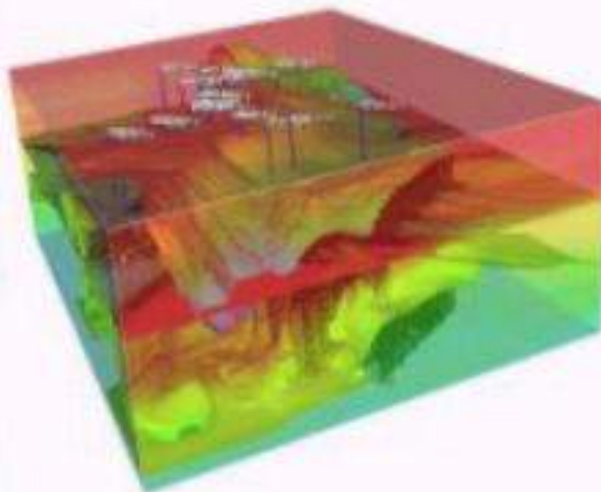


科学计算可视化的应用领域主要有：计算流体力学；有限元分析；分子模型；医学图像；空间探测；天体物理；地球科学；数学；软件开发以及其它。科学计算可视化应用的实例如图1.26所示。

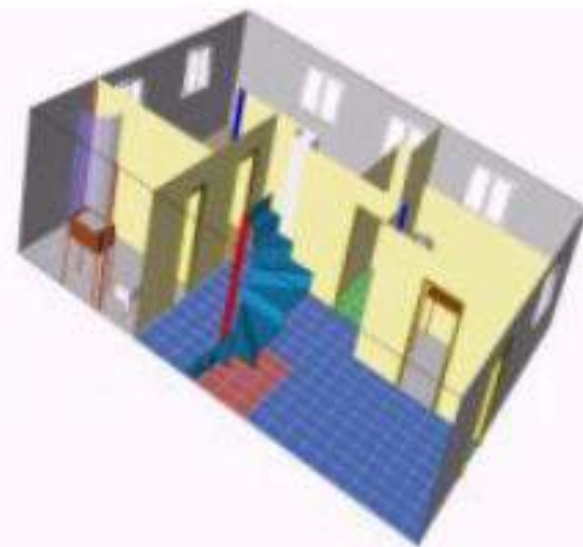




(a)



(b)



(c)

图 1.26 可视化应用示例

(a) 水库模型的裁剪； (b) 地形地貌3D图； (c) 房屋结构的可视化





1.5.6 虚拟现实

虚拟现实的特征(VR)有四个主要特征,用以区别相邻技术,如计算机图形学、多媒体技术、仿真技术、科学计算可视化技术等。这四个主要特征分别是:

- (1) 多感知性(Multi-Sensation)。
- (2) 沉浸感(Immersion)。
- (3) 交互性(Interaction)。
- (4) 自主性(Autonomy)。



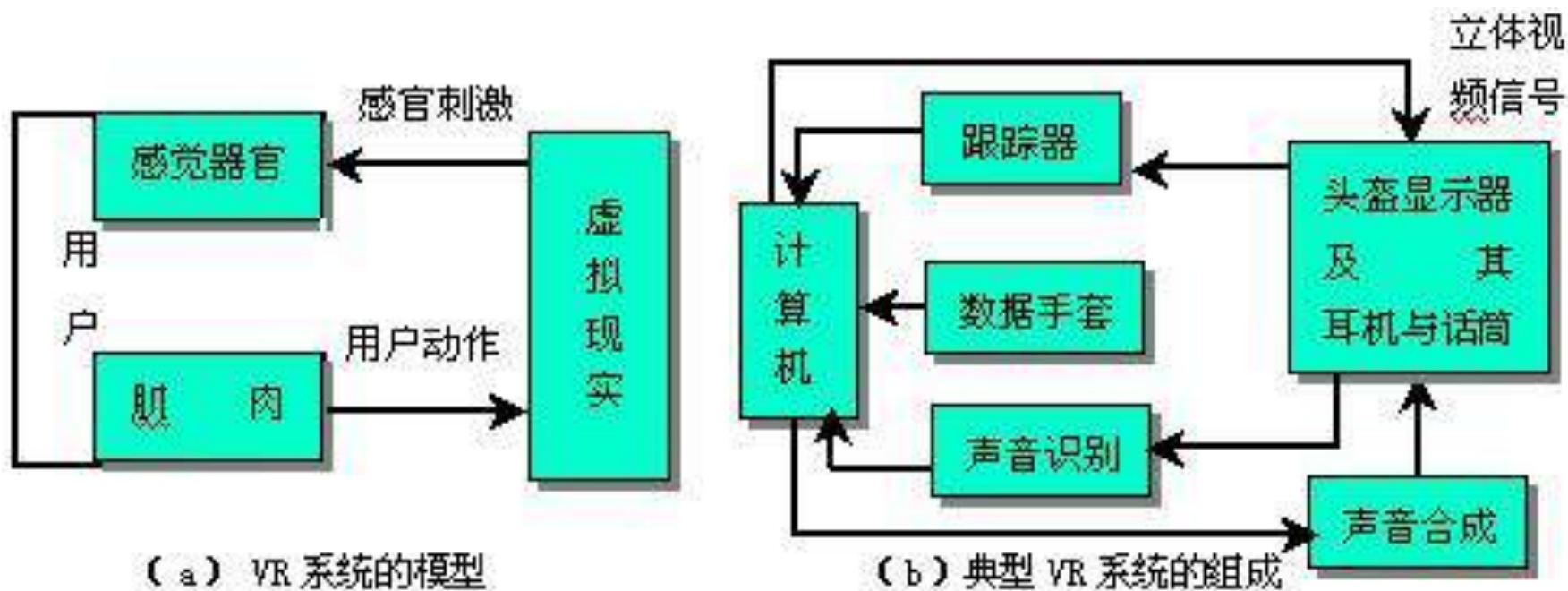


图 1.27 VR系统的模型与组成

(a) VR系统的模型； (b) 典型VR系统的组成





VR系统中的关键技术主要有：实时动态真实感图形绘制技术；宽视场、立体显示及立体声效果等临场感技术；快速高精度的三维跟踪技术；手势、姿态、言语、声音等的辨识技术；各种传感技术等。它们所涉及的研究内容有如下四个领域：感知领域；人-机接口；软件(如视觉建模、听觉建模、触觉建模、各种软件工具等)；硬件。





1.5.7 地理信息系统

地理信息系统(GIS-Geographic Information System)是用来获取、储存、管理、分析和显示空间数据及空间实体的属性数据的信息系统。除计算机科学与技术以及图形学外,它还涉及地理、测绘、制图等学科。





1.5.8 并行图形处理

图形并行处理的发展首先表现在计算机体系结构上。并行计算机体系结构大致分为六种：单指令多数数据流 (SIMD); 并行向量处理机 (PVP); 对称多处理机 (SMP); 大规模并行处理机 (MPP); 工作站集群 (COW) 以及分布共享存储器 (DSM)多处理机。 SIMD计算机大都为专用, 其余各种模型全为MIMD计算机。近年来, 包含工作站网络(NOW-Network of Workstation)或工作站集群 (Workstation Cluster)的计算机机群成为并行体系结构中最为引人注目的研究主流之一。





1.5.9 图形图像技术的融合

计算机图形学研究如何从计算机模型出发, 把真实的或想像的物体图形描绘出来。而图像处理中的图像重建进行的却是与此相反的过程, 它是基于画面进行二维或二维物体模型的重建, 这在很多场合都是十分重要的。如高空监测摄影、宇航探测器收集到的月球或行星的慢速扫描电视图像、工业机器人“眼”中测到的电视图像、染色体扫描、X射线图像、断层扫描、指纹分析等, 都需要图像处理技术。





习 题

1. 名词解释: 图形、图像、点阵法、参数法。
2. 图形包括哪两方面的要素?在计算机中如何表示它们?
3. 什么是计算机图形学?简述计算机图形学、 数字图像处理 and 计算机视觉学科间的关系。
4. 简述图形信息的特点。
5. 简述计算机图形系统的组成。





6. 计算机图形系统的工作方式有几种？ 分别是什么？
7. 简述计算机图形学的主要研究内容。
8. 有关计算机图形学的软件标准有哪些？
9. 试从科学发展历史的角度分析计算机图形学以及硬件设备的发展过程。
10. 试发挥你的想象力，举例说明计算机图形学有哪些应用？解决的问题是什么？





11. 一个交互式计算机图形系统应该具有哪几种功能?其结构如何?
12. 试说明你认为计算机图形学将来的发展方向, 并进行设计和分析。
13. 简述计算机图形学当前的研究动态, 并简要说明其含义。
14. 简述简单的虚拟现实系统的构成。
15. 什么是科学计算可视化? 简述科学计算可视化的研究内容。
16. 什么是计算机动画? 它对运动描述与控制的技术有哪些?





直线光栅化算法

- DDA算法
- Bresenham算法

圆光栅化算法

- 中点算法
- 中点整数算法
- 中点整数优化算法





2.1 直线光栅化法

DDA 算法 (Digital Differential Analyzer)

- David F. Rogers 的描述 (适用于所有象限)
- James D. Foley 的描述 (只适用于第一象限 , 且 $K < 1$)
- 本教程的描述 (适用于所有象限及任何端点)

Bresenham 算法

- 基本原理
- Bresenham 算法
- 整数 Bresenham 算法
- 一般整数 Bresenham 算法





2.1.1 DDA算法算法

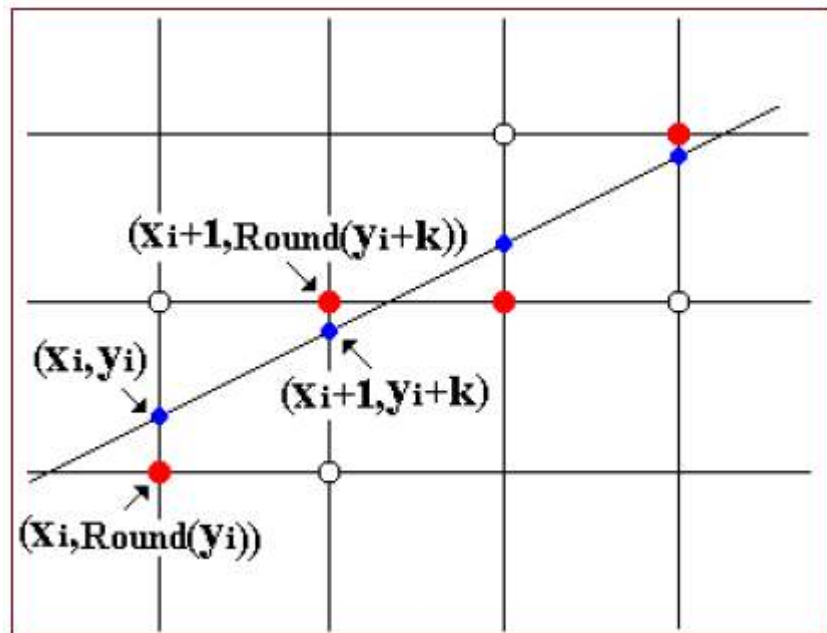
1) David F. Rogers 描述描述

直线的基本微分方程是：

$$\frac{dy}{dx} = \text{常数 } (k)$$

设直线通过点 $P1(x1,y1)$ 和 $P2(x2,y2)$,
则直线方程可表示为：

$$\frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1} = k$$





1) David F. Rogers 描述描述

- ◆ 如果已知第 i 点的坐标，可用步长 StepX 和 StepY 得到

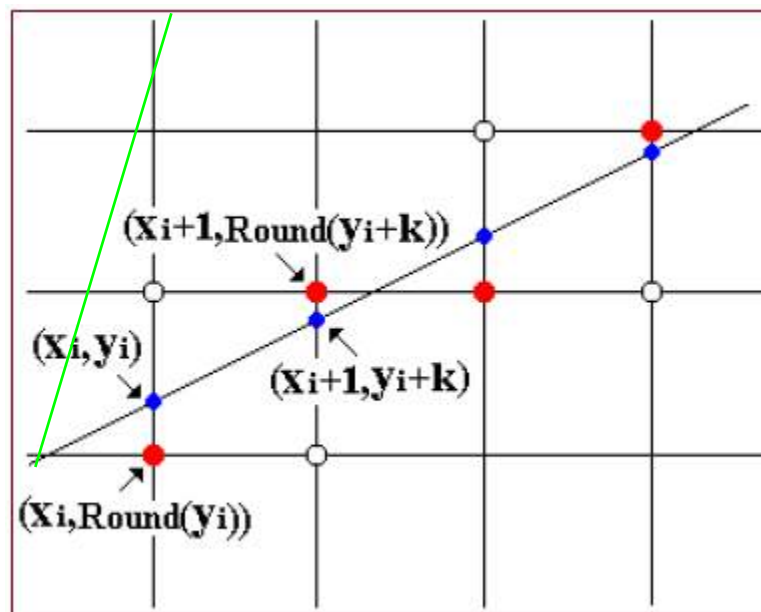
第 $i+1$ 点的坐标为：

- $x_{i+1} = x_i + \text{StepX}$
- $y_{i+1} = y_i + \text{StepY}$ 或 $y_{i+1} = y_i + k * \text{StepX}$

- ◆ 例图中

- $k < 1$
- $\text{StepX} = 1$
- $\text{StepY} = k$

- ◆ 将算得的直线上每个点的当前坐标，按四舍五入得到光栅点的位置





1) David F. Rogers 描述

// Digital Differential Analyzer (DDA) routine for rasterizing a line

// The line end points are (xs,ys) and (xe,ye) assumed not equal.

// Round is the function. Note: Many Round functions are floor functions, i.e Round (-8.5)=-9 rather than -8. The algorithm assumes this is the case.

// Approximate the line length

If $(|xs - xe| \geq |ys - ye|)$ then //插补长度

Length $\leftarrow |xs - xe|;$

else

Length $\leftarrow |ys - ye|;$

end if





1) David F. Rogers 描述描述

// Select the larger of Δx or Δy to be one raster unit.

StepX = ($x_e - x_s$) / Length;

StepY = ($y_e - y_s$) / Length;

x = x_s ; //首点

y = y_s ;

i = 1; // Begin main loop

while (i ≤ Length)

 WritePixel (Round(x), Round(y) ,value));

 x = x + StepX;

 y = y + StepY;

 i++;

end while





2) James D.Foley 描述描述

□ 令
$$k = \frac{y_2 - y_1}{x_2 - x_1}$$

有：

$$y_{i+1} = y_i + k * \text{StepX}$$

□ 若 $0 < k < 1$ ，即 $\Delta x > \Delta y$

□ 因光栅单位为 1，

□ 可以采用每次 x 方向增加 1，

□ 而 y 方向增加 k 的办法得到下一个直线点。





2) James D.Foley 描述描述

```
void Line ( //设  $0 \leq k \leq 1, x_s < x_e$ 
```

```
    int xs,ys; //左端点
```

```
    int xe,ye; //右端点
```

```
    int value) //赋给线上的象数值
```

```
{
```

```
    int x; //x以步长为单位从 xs增长到 xe
```

```
    double dx =xe-xs;
```

```
    double dy =ye-ys;
```

```
    double k =dy/dx; // 直线之斜率 k
```

```
    double y =ys;
```

```
    for (x=xs; x<=xe; x++) {
```

```
        WritePixel(x,Round(y),value); //置象数值为 value
```

```
        y+=k; // y移动步长是斜率 k
```

```
    } // End of for
```

```
} // Line
```





3) 已有算法描述分析

Rogers 描述 :

- ◆ 采用 $x = x + \text{StepX}$, $y = y + \text{StepY}$,
- ◆ 逼近点并不是直线的一个最好的逼近 ;

D.Foley描述 : 可能引起积累误差

- ◆ 未分析直线端点不在象素点上的情况 ;
- ◆ 只给出 $0 - 45^\circ$ 第一个八卦限的描述 。

为避免引起积累误差 , **D.Foley**描述中采用

- ◆ `double dx =xe-xs;`
- ◆ `double dy =ye-ys;`
- ◆ `double k =dy/dx; // 直线之斜率 k`





4)本教程描述——任意方向直线插补算法

```
void DDALine (  
    float xs, ys; //起点  
    float xe, ye; //终点  
    int value) //赋给线上的象数值  
{  
    int n, ix, iy, idx, idy ;  
    int Flag; //插补方向标记  
    int Length; //插补长度  
    float x, y, dx, dy;
```





第 2 章 基本图形生成算法 (I)

```
dx=xe-xs;          dy=ye-ys;
if (fabs(dy)<fabs(dx)) { //X方向长 , 斜率 <=1
    Length=abs(Round(xe)-Round(xs));
    Flag=1; //最大的插补长度和方向标记
    ix= Round(xs); //初始 X点
    idx=isign(dx); //X方向单位增量
    y= ys+dy/dx*((float)(ix)-xs); //初始 Y点修正
    dy=dy/fabs(dx); //Y方向斜率增量
}
else { // Y方向长 , 斜率 >1
    Length=abs(Round(ye)-Round(ys));
    Flag=0;
    iy= Round(ys); //初始 Y点
    idy=isign(dy); //Y方向单位增量
    x= xs+dx/dy*((float)(iy)-ys); //初始 X点修正
    dx=dx/fabs(dy); //X方向斜率增量
}
```





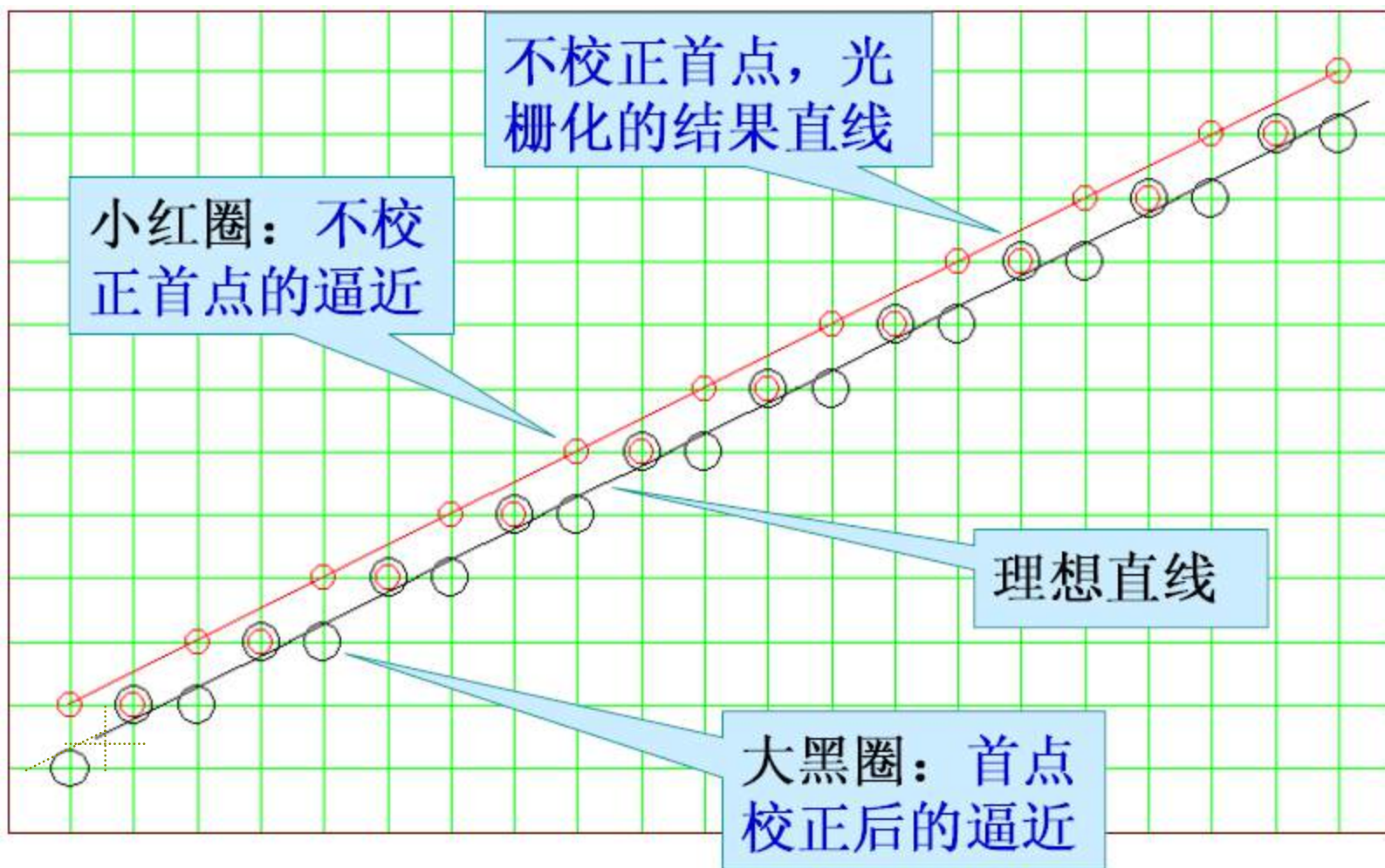
第 2 章 基本图形生成算法 (I)

```
if (Flag) { //X方向单位增量
    for (n=0; n<= Length; n++) { //X方向插补过程
        WritePixel(ix, Round(y), value);
        ix+=idx;
        y+=dy;
    } //End of for
} //End of if
else { //Y方向斜率增量
    for (n=0; n<= Length; n++) { //Y方向插补过程
        WritePixel (Round(x), iy, value);
        iy+=idy;
        x+=dx;
    } //End of for
} //End of else
} //Finish
```





5) 本教程描述 —— 首点校正对逼近的影响





2.1.2 Bresenham算法

- Bresenham算法是计算机图形学典型的直线光栅化算法 。
- 从另一个角度看直线光栅化显示算法的原理 ：

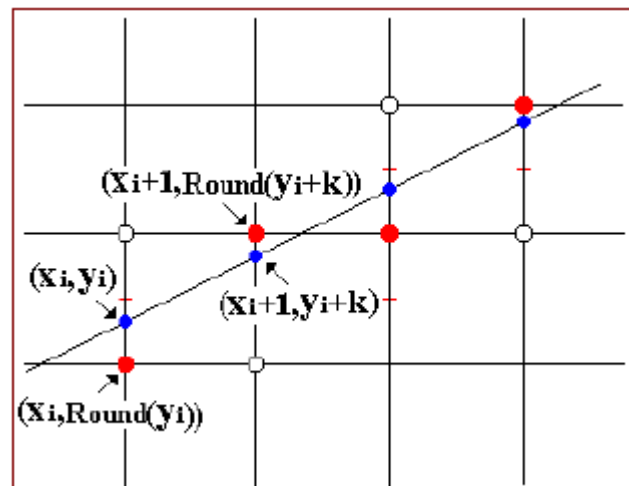
由直线的斜率确定选择在 x 方向或 y 方向上每次递增（减）1个单位，另一变量的递增（减）量为 0 或 1，它取决于实际直线与最近光栅网格点的距离，这个距离的最大误差为 0.5。





1) Bresenham的基本原理

- 假定直线斜率 k 在 $0 \sim 1$ 之间。此时，只需考虑 x 方向每次递增 1 个单位，决定 y 方向每次递增 0 或 1。
- 设直线的
当前点为 (x_i, y_i)
当前光栅点为 (x_i, y_i)
- 下一个
直线的点应为 $(x_{i+1}, y_i + k)$
直线的光栅点
 - 或为右光栅点 (x_{i+1}, y_i) (y 方向递增量 0)
 - 或为右上光栅点 $(x_{i+1}, y_i + 1)$ (y 方向递增量 1)





第 2 章 基本图形生成算法 (I)

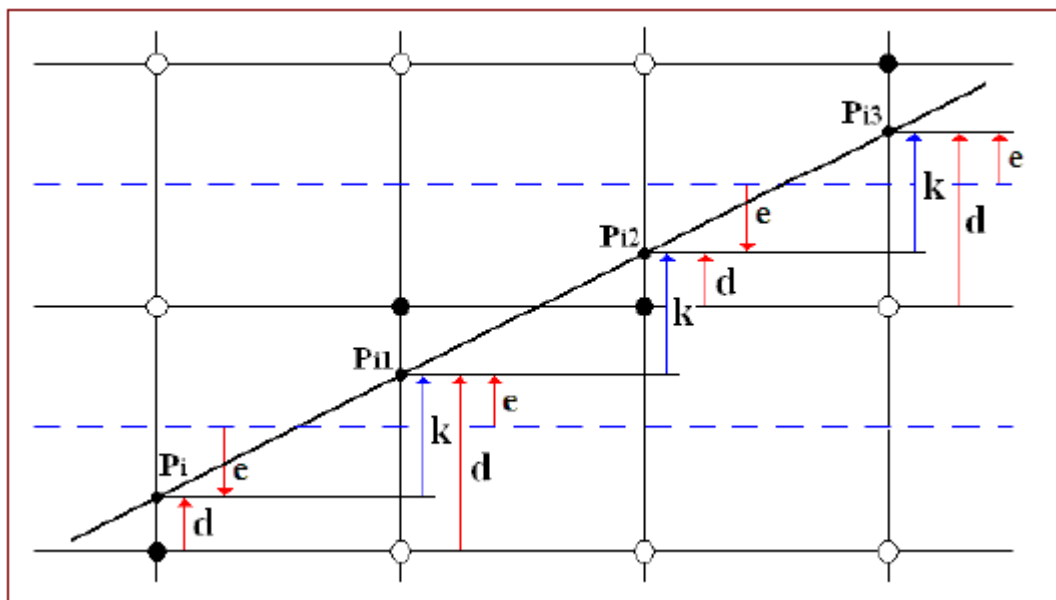
□ 记直线与它垂直方向最近的下光栅点的误差为 d ,

有： $d = (y + k) - y_i$ ，且

□ $0 \leq d \leq 1$

□ 当 $d < 0.5$ ：下一个象素应取右光栅点 (x_{i+1}, y_i)

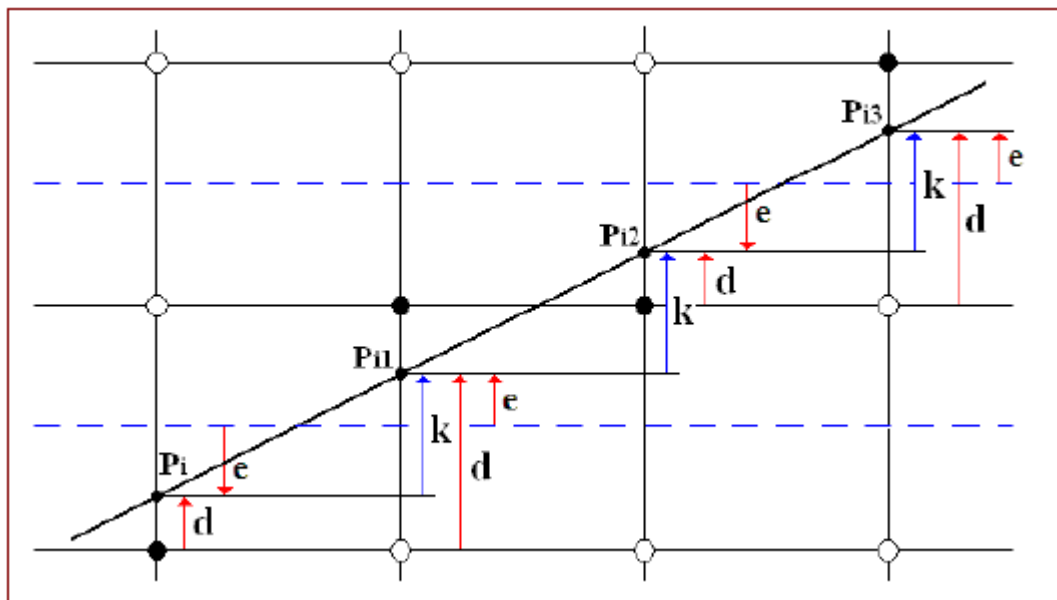
□ 当 $d \geq 0.5$ ：下一个象素应取右上光栅点 (x_{i+1}, y_{i+1})





1) Bresenham的基本原理

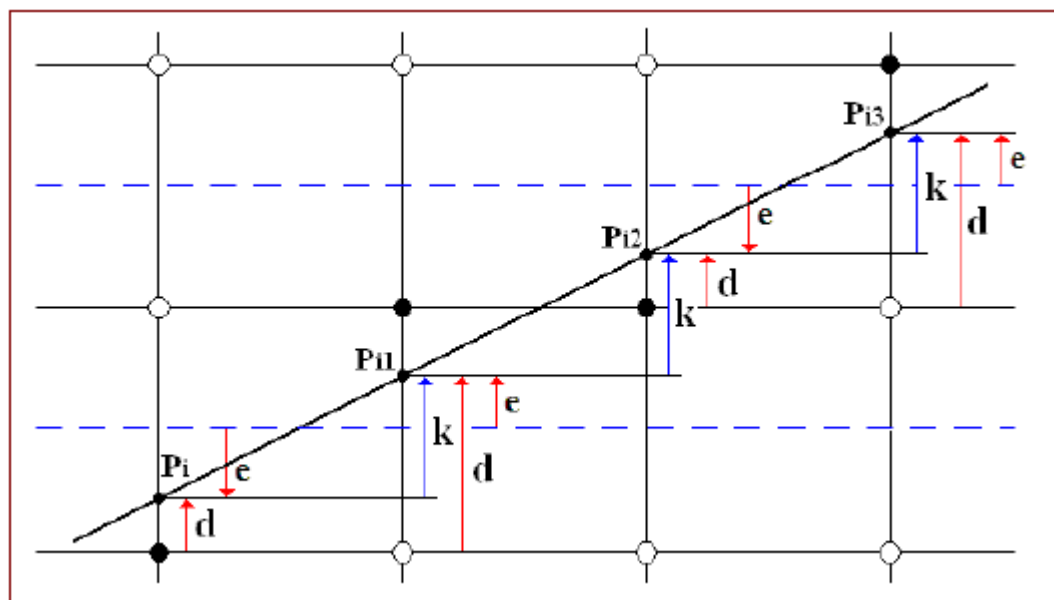
- 如果直线的（起）端点在整数点上，误差项 d 的初值： $d_0 = 0$
- x 坐标每增加 1， d 的值相应递增直线的斜率值 k ，
即： $d = d + k$
- 一旦 $d \geq 1$ ，就把它减去 1，保证 d 的相对性，且在 0-1 之间。





第 2 章 基本图形生成算法 (I)

- 令 $e=d-0.5$ ，关于 d 的判别式和初值可简化成：
 - e 的初值 $e_0 = -0.5$ ，增量亦为 k ;
 - $e < 0$ 时，取当前像素 (x_i, y_i) 的右方像素 (x_{i+1}, y_i) ;
 - $e > 0$ 时，取当前像素 (x_i, y_i) 的右上方像素 $(x_{i+1}, y_i + 1)$;
 - $e = 0$ 时，可任取上、下光栅点显示。





1) Bresenham的基本原理

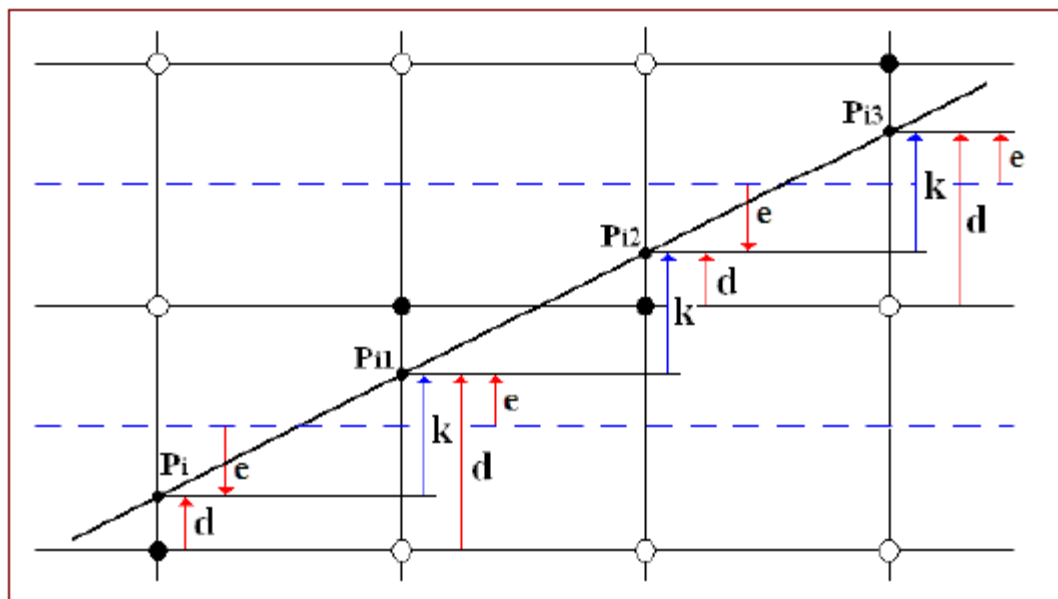
- Bresenham算法的构思巧妙：它引入动态误差 e ，当 x 方向每次递增 1 个单位，可根据 e 的符号决定 y 方向每次递增 0 或 1。
 - $e < 0$ ， y 方向不递增
 - $e > 0$ ， y 方向递增 1
 - x 方向每次递增 1 个单位， $e = e + k$





1) Bresenham的基本原理

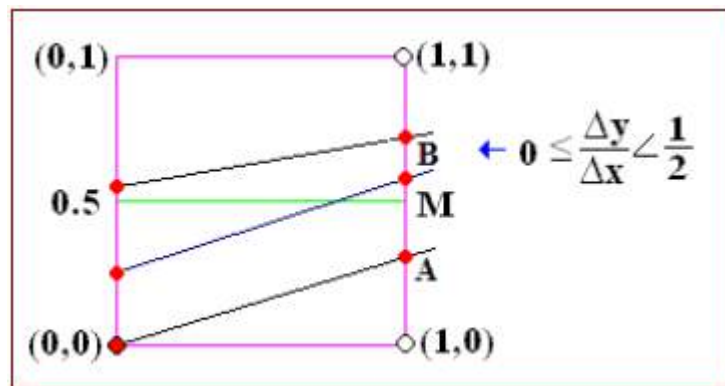
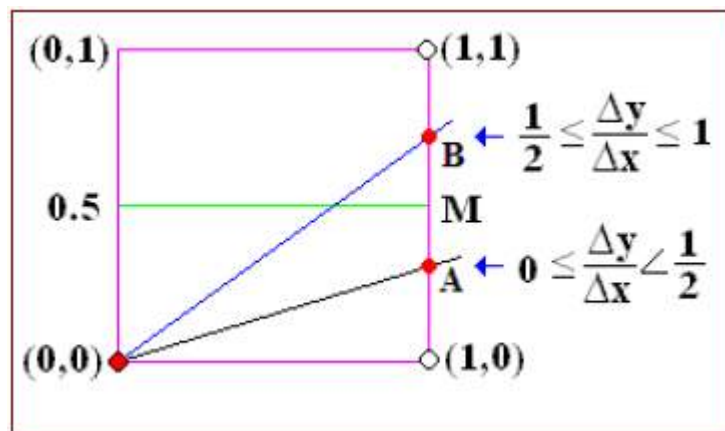
□ 因为 e 是相对量，所以当 $e > 0$ 时，表明 e 的计值将进入下一个参考点（上升一个光栅点），此时须： $e = e - 1$





2) Bresenham算法的实施——Rogers 版

- 通过 (0,0) 的所求直线的斜率大于 0.5，它与 $x=1$ 直线的交点离 $y=1$ 直线较近，离 $y=0$ 直线较远，因此取光栅点 (1,1) 比 (1,0) 更逼近直线；
- 如果斜率小于 0.5，则反之；
- 当斜率等于 0.5，没有确定的选择标准，但本算法选择 (1,1)。





2) Bresenham算法的实施 ——Rogers 版

```
//Bresenham's line rasterization algorithm for the first octal.  
//The line end points are (xs,ys) and (xe,ye) assumed not equal.  
// Round is the integer function.  
// x,y,  $\Delta x$ ,  $\Delta y$  are the integer, Error is the real.  
//initialize variables  
x=xs  
y=ys  
 $\Delta x = x_e - x_s$   
 $\Delta y = y_e - y_s$   
//initialize e to compensate for a nonzero intercept  
Error =  $\Delta y / \Delta x - 0.5$ 
```





2) Bresenham 算法的实施 —— Rogers 版

//begin the main loop

for i=1 to Δx

 WritePixel (x, y, value)

 if (Error ≥ 0) then

 y=y+1

 Error = Error -1 提问学生 why ?

 end if

 x=x+1

 Error = Error + $\Delta y / \Delta x$

next i

finish





3) 整数 Bresenham 算法

- 上述 Bresenham 算法在计算直线斜率和误差项时要用到浮点运算和除法，采用整数算术运算和避免除法可以加快算法的速度。
- 由于上述 Bresenham 算法中只用到误差项（初值 $\text{Error} = \Delta y / \Delta x - 0.5$ ）的符号
- 因此只需作如下的简单变换：
$$\text{NError} = 2 * \text{Error} * \Delta x$$
- 即可得到整数算法，这使本算法便于硬件（固件）实现





3) 整数 Bresenham 算法

//Bresenham's integer line rasterization algorithm
for the first octal.

//The line end points are (xs,ys) and (xe,ye) assumed
not equal. All variables are assumed integer.

//initialize variables

$x = x_s$

$y = y_s$

$\Delta x = x_e - x_s$

$\Delta y = y_e - y_s$

//initialize e to compensate for a nonzero intercept

$NError = 2 * \Delta y - \Delta x$ (**Error** = $\Delta y / \Delta x - 0.5$)





3) 整数 Bresenham 算法

//begin the main loop

for i=1 to Δx

WritePixel (x, y)

if (NError ≥ 0) then

y=y+1

NError = NError - 2* Δx (Error = Error -1)

end if

x=x+1

NError = NError + 2* Δy (Error = Error + $\Delta y / \Delta x$)

next i

finish





4)一般 Bresenham算法算法

- 要使第一个八卦的 Bresenham算法适用于一般直线，只需对以下 2点作出改造：
 - 当直线的斜率 $|k|>1$ 时，改成 y 的增量总是 1，再用 Bresenham误差判别式确定 x 变量是否需要增加 1；
 - x 或 y 的增量可能是 “+1”或 “-1”，视直线所在的象限决定。





第 2 章 基本图形生成算法 (I)

//Bresenham's integer line rasterization algorithm for all quadrants

//The line end points are (xs,ys) and (xe,ye) assumed not equal. All variables are assumed integer.

//initialize variables

x=xs

y=ys

$\Delta x = \text{abs}(xe - xs)$

$$\Delta x = xe - xs$$

$\Delta y = \text{abs}(ye - ys)$

$$\Delta y = ye - ys$$

sx = isign(xe - xs)

sy = isign(ye - ys)

//Swap Δx and Δy depending on the slope of the line.

if $\Delta y > \Delta x$ then

 Swap($\Delta x, \Delta y$)

 Flag=1

else

 Flag=0

end if





第 2 章 基本图形生成算法 (I)

//initialize the error term to compensate for a nonzero

intercept

$NError = 2 * \Delta y - \Delta x$

//begin the main loop

for $i=1$ to Δx

WritePixel(x, y , value)

if ($NError \geq 0$) then

if (Flag) then // $\Delta y > \Delta x$

$x = x + sx$

else

$Y = Y + 1$

$y = y + sy$

end if // End of Flag

$NError = NError - 2 * \Delta x$

end if // End of NError





4)一般 Bresenham算法算法

if (Flag) then $\Delta y > \Delta x$

$y = y + sy$

else

$X = X + 1$

$x = x + sx$

end if

$NError = NError + 2 * \Delta y$

next i

finish





2.2 圆光栅化算法

2.2.1 利用圆的八方对称性画圆

□ 对圆的分析均假定圆心在坐标原点， 因为即使圆心不在原点， 可以通过一个简单的平移即可， 而对原理的叙述却方便了许多

□ 即考虑圆的方程为： $x^2+y^2=R^2$

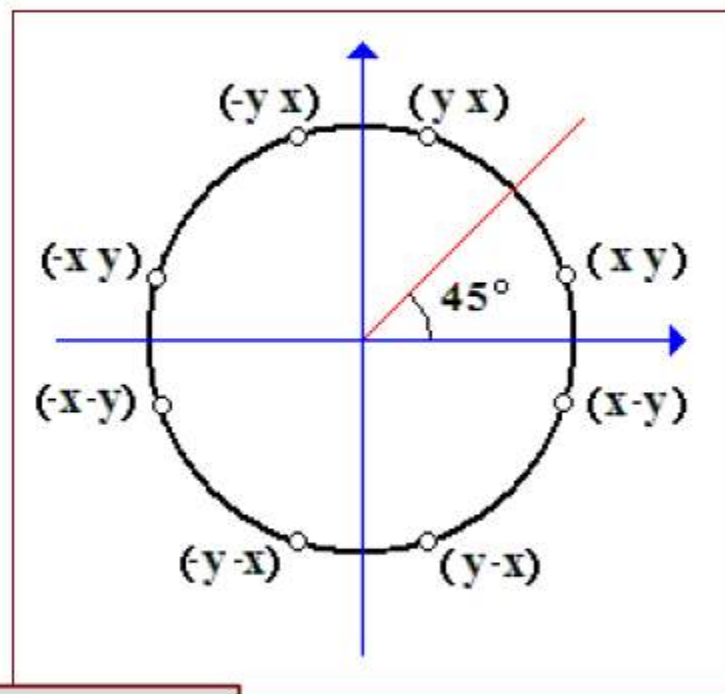




2.2.1 利用圆的八方对称性画圆

***void* CirclePoints (*int* x,*int* y, *int* value)**

```
{  
    WritePixel (x, y, value);  
    WritePixel (-x, y, value);  
    WritePixel (-x, -y, value);  
    WritePixel (x, -y, value);  
    WritePixel (y, x, value);  
    WritePixel (-y,x, value);  
    WritePixel (-y, -x, value);  
    WritePixel (y, -x, value);  
}
```



显然，当 $x=0$ 或 $x=y$ 或 $y=0$ 时，圆上的对称点只有4个，因此，CirclePoints()需要修正。



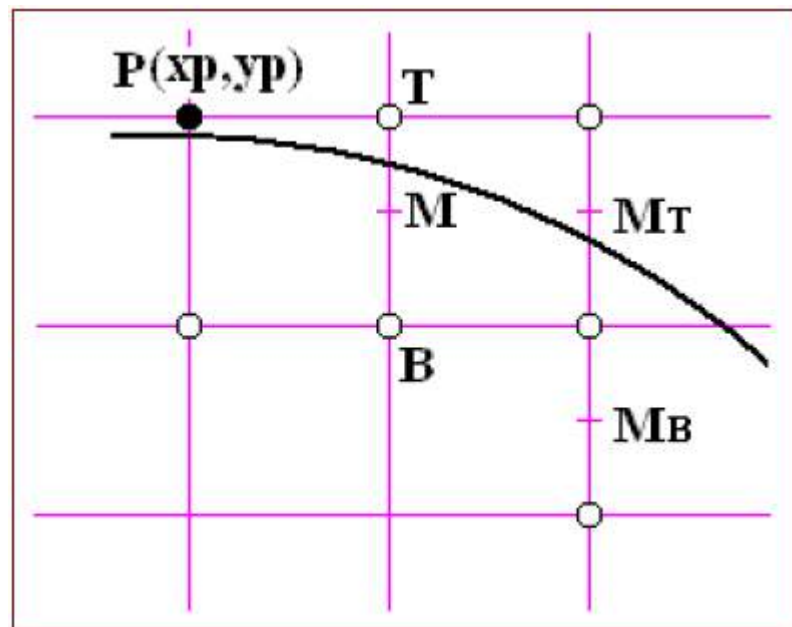


2.2.2 中点圆算法 —— 原理

□ 设 d 是点 $p(x,y)$ 到圆心的距离，有：

$$d = F(x,y) = x^2 + y^2 - R^2$$

□ 按照 Bresenham 算法符号变量的思想，以圆的下 2 个可选像素中点的函数值 d 的符号 决定选择 2 个可选像素 T 和 B 中哪一个更接近圆而作为圆的显示点？

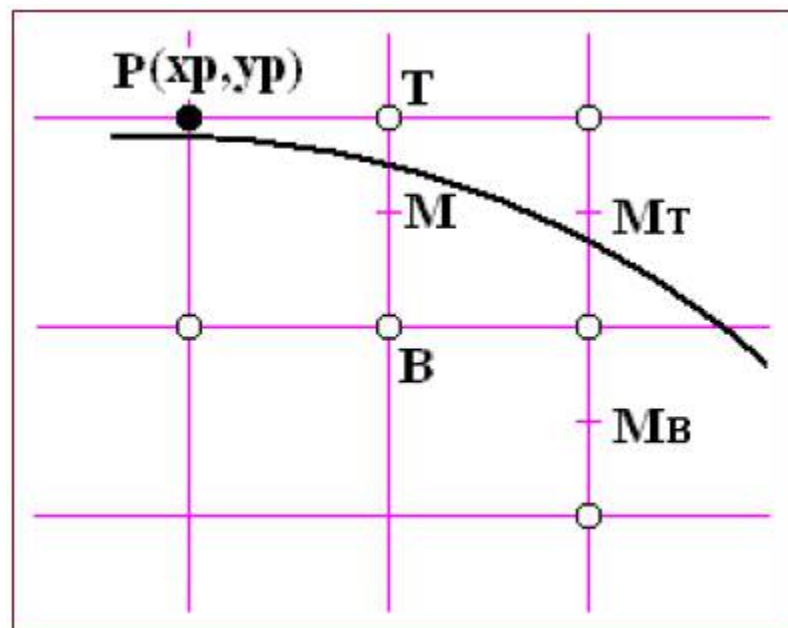


□ $d_M = F(x_M, y_M) = F(x_p + 1, y_p - 0.5) = (x_p + 1)^2 + (y_p - 0.5)^2 - R^2$

□ $d_{MT} = F(x_{MT}, y_{MT}) = F(x_p + 2, y_p - 0.5) = (x_p + 2)^2 + (y_p - 0.5)^2 - R^2$

□ $\Delta d_{MT} = d_{MT} - d_M = 2x_p + 3$

注意: $x_p^2 + y_p^2 - R^2$ 并不等于零





第 2 章 基本图形生成算法 (I)

如果 $d_M > 0$ ，表示下一中点 M 在圆外，用 B 点逼近，得

□ $d_{MB} = F(x_{MB}, y_{MB}) = F(x_p + 2, y_p - 1.5) = (x_p + 2)^2 + (y_p - 1.5)^2 - R^2$

□ $\Delta d_{MB} = d_{MB} - d_M = 2x_p - 2y_p + 5$

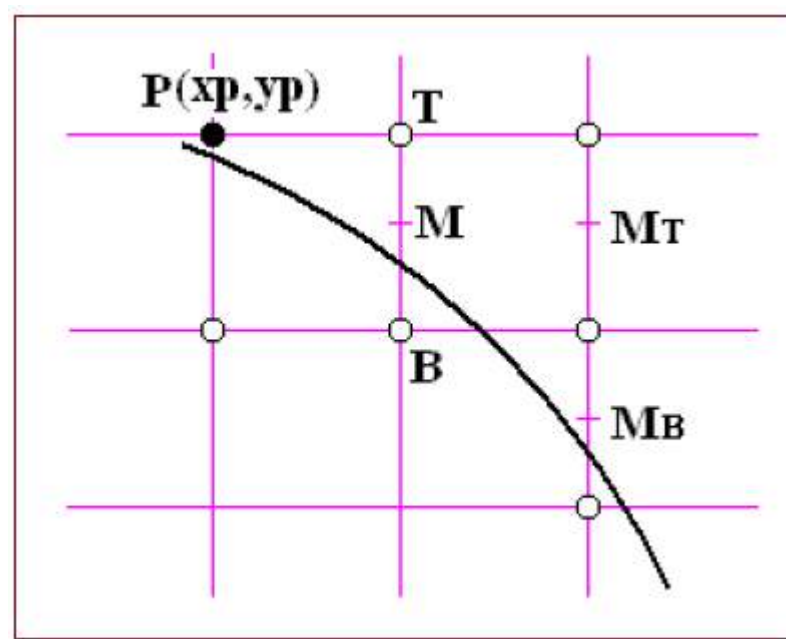
□ 结论：

□ 根据中点 d 的值，决定

□ 显示的光栅点（ T 或 B ）

□ 新的 Δd （ Δd_{MT} 或 Δd_{MB} ）

□ 更新 d

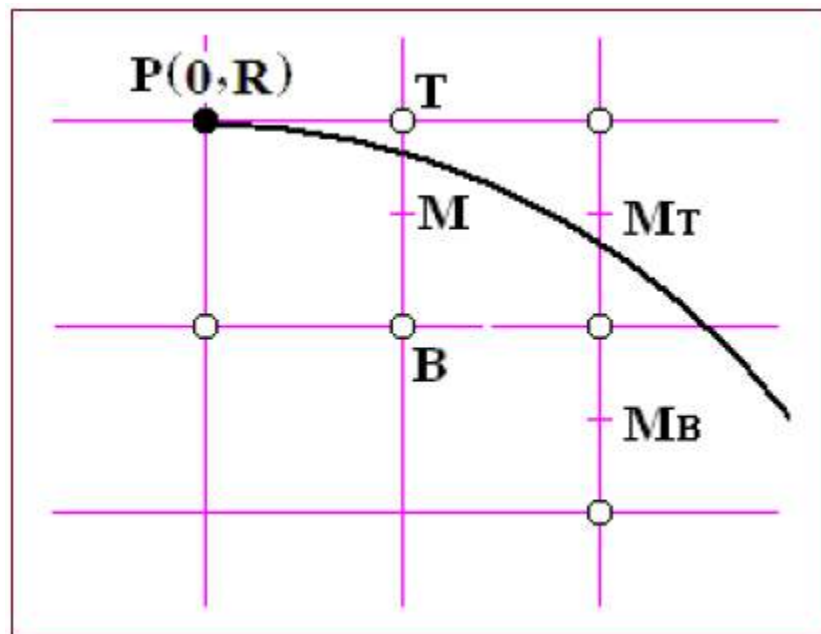




2.2.2 中点圆算法 —— 原理

初值

- 由 $x_0=0$, $y_0=R$
- 得 $x_{M0}=0+1$, $y_{M0}=R-0.5$
- $d_{M0}=F(x_{M0}, y_{M0})=F(1, R-0.5)=1^2+(R-0.5)^2-R^2=1.25-R$





2.2.3 中点圆算法 —— 实施

//中点圆算法 (假设圆的中心在原点)

```
void MidPointCircle(int radius,int value)
```

```
{
```

```
    int x=0;
```

```
    int y= radius;
```

```
    double d=1.25- radius;
```





2.2.3 中点圆算法 —— 实施

```
While (y>x) {  
    CirclePoints(x, y,value);  
    if (d<0) //选择 T  
        d+=2.0*x+3.0;  
    else { //选择 B  
        d+=2.0*(x-y)+5.0;  
        y--;  
    }  
    x++;  
} //End of while  
}
```





2.2.4 中点圆整数算法 —— 原理

- 中点圆算法 的半径是整数，而用于该算法
符号判别的变量 d （初值 $d=1.25-\text{radius}$ ）
采用浮点运算，会花费较多的时间。
- 为了将其改造成整数计算，定义新变量：
 $D = d - 0.25$
- 那么判别式 $d < 0$ 等价于 $D < -0.25$ 。
- 在 D 为整数情况下， $D < -0.25$ 和 $D < 0$ 等价
- 仍将 D 写成 d （新的初值 $d=1-\text{radius}$ ），可
得到 中点圆整数算法。





2.2.4 中点圆整数算法 —— 实施

//中点圆算法 (假设圆的中心在原点)

```
void MidPointCircle(int radius,int value)
```

```
{
```

```
    int x=0;
```

```
    int y= radius;
```

```
    int d=1- radius;    d=1.25-
```

```
    //CirclePoints(x, y, value);    radius
```





2.2.4中点圆整数算法 —— 实施

While (y>x) {

 CirclePoints(x, y,value);

 if (d<0) //选择 T

 d+=2*x+3;

d+=2.0*x+3.0

 else { //选择 B

 d+=2*(x-y)+5;

d+=2.0*(x-y)+5.0

 y--;

 } //End of else

 x++;

 //CirclePoints(x, y,value);

} //End of while

}





2.2.5 中点圆整数优化算法 — 原理

□ 用 Δd 修正 d

□ 1) 选择 T 点 ($x_p \leftarrow x_p + 1$):

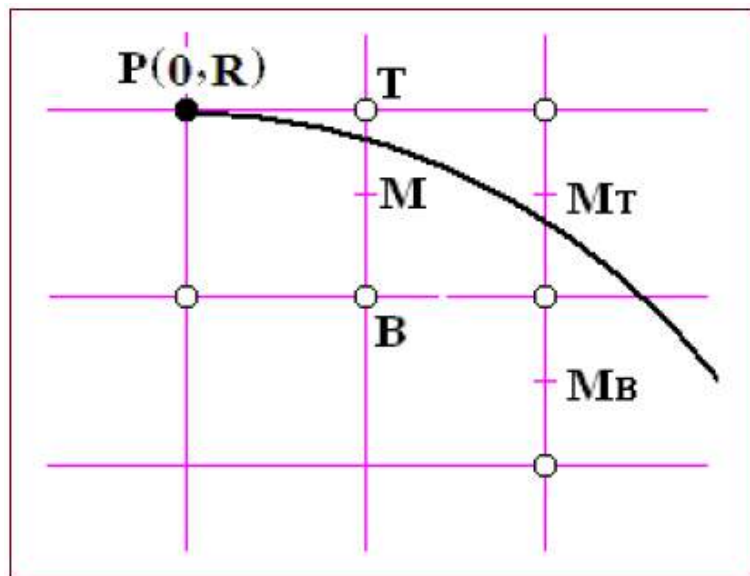
□ d 的增量 (一次差分):

$$\triangleright \Delta d_T = 2x_p + 3$$

□ Δd 的增量 (二次差分):

$$\triangleright \Delta^2 d_T = 2(x_p + 1) + 3 - (2x_p + 3) = 2$$

$$\triangleright \Delta^2 d_B = 2(x_p + 1) - 2y_p + 5 - (2x_p - 2y_p + 5) = 2$$





2.2.5 中点圆整数优化算法 — 原理

□ 2) 选择 B 点 ($x_p \leftarrow x_p + 1$, $y_p \leftarrow y_p - 1$) :

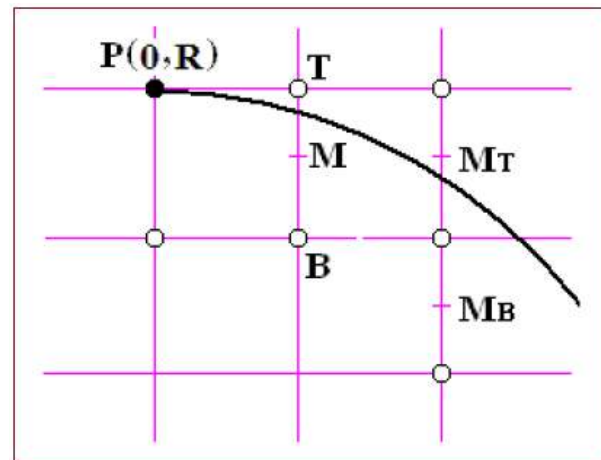
□ d 的增量 (一次差分) :

$$\triangleright \Delta d_B = 2x_p - 2y_p + 5$$

□ Δd 的增量 (二次差分) :

$$\triangleright \Delta^2 d_T = 2(x_p + 1) + 3 - (2x_p + 3) = 2$$

$$\triangleright \Delta^2 d_B = 2(x_p + 1) - 2(y_p - 1) + 5 - (2x_p - 2y_p + 5) = 4$$





2.2.5 中点圆整数优化算法 — 实施

//中点圆整数优化算法 （ 假设圆的中心在原点 ）

```
void MidPointCircleInt(int radius,int value)
```

```
{
```

```
    int x=0;
```

```
    int y= radius;
```

```
    int d=1- radius;
```

```
    int dt=3;
```

```
    int db= -2*radius+5;
```





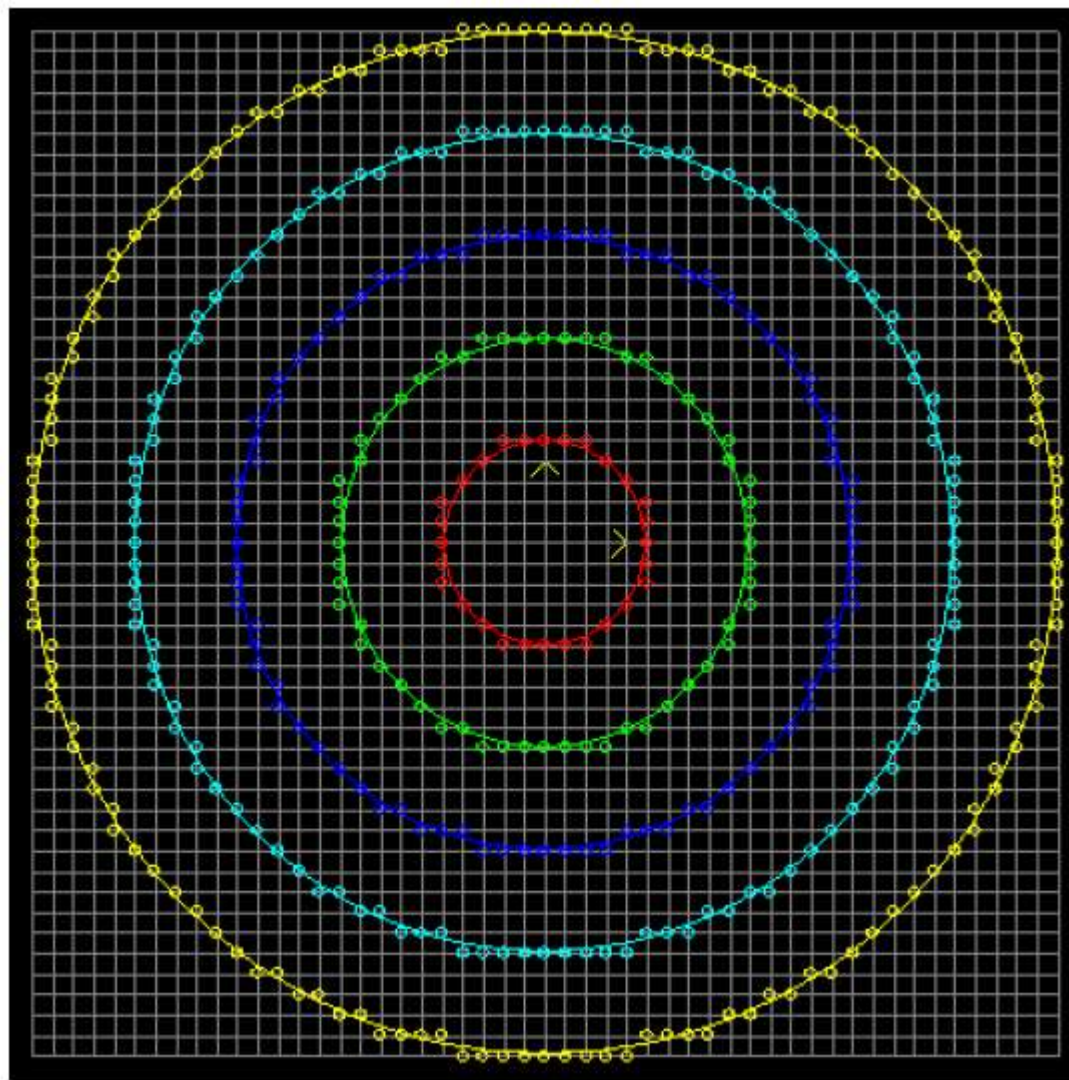
第 2 章 基本图形生成算法 (I)

```
While (y>=x) {  
    CirclePoints(x, y,value);  
    if (d<0) //选择 T  
        d=d+dt;  
        dt+=2;  
        db+=2;  
  
    else { //选择 B  
        d=d+db;  
        dt+=2;  
        db+=4;  
        y--;  
    }  
    x++;  
} //End of while  
} //Finish
```





2.2.5 中点圆整数中点圆整数优化算法 — 例子





总结

- 直线光栅化算法
 - DDA算法
 - Bresenham算法
- 圆光栅化算法
 - 中点算法
 - 中点整数算法
 - 中点整数优化算法
- 基本方法
 - 增量算法
 - 符号算法





2.3 椭圆光栅化算法





2.3.1 椭圆的扫描转换

中点画圆法可以推广到一般二次曲线的生成, 下面以中心在原点的标准椭圆的扫描转换为例说明。 设椭圆的方程为

$$F(x,y)=b^2x^2+a^2y^2-a^2b^2=0$$

其中, a 为沿 x 轴方向的长半轴长度, b 为 y 轴方向的短半轴长度, a 、 b 均为整数。 不失一般性, 我们只讨论第一象限椭圆弧的生成。 需要注意的是, 在处理这段椭圆时, 必须以弧上斜率为-1的点(即法向量两个分量相等的点)作为分界把它分为上部分和下部分, 如图2.6所示。



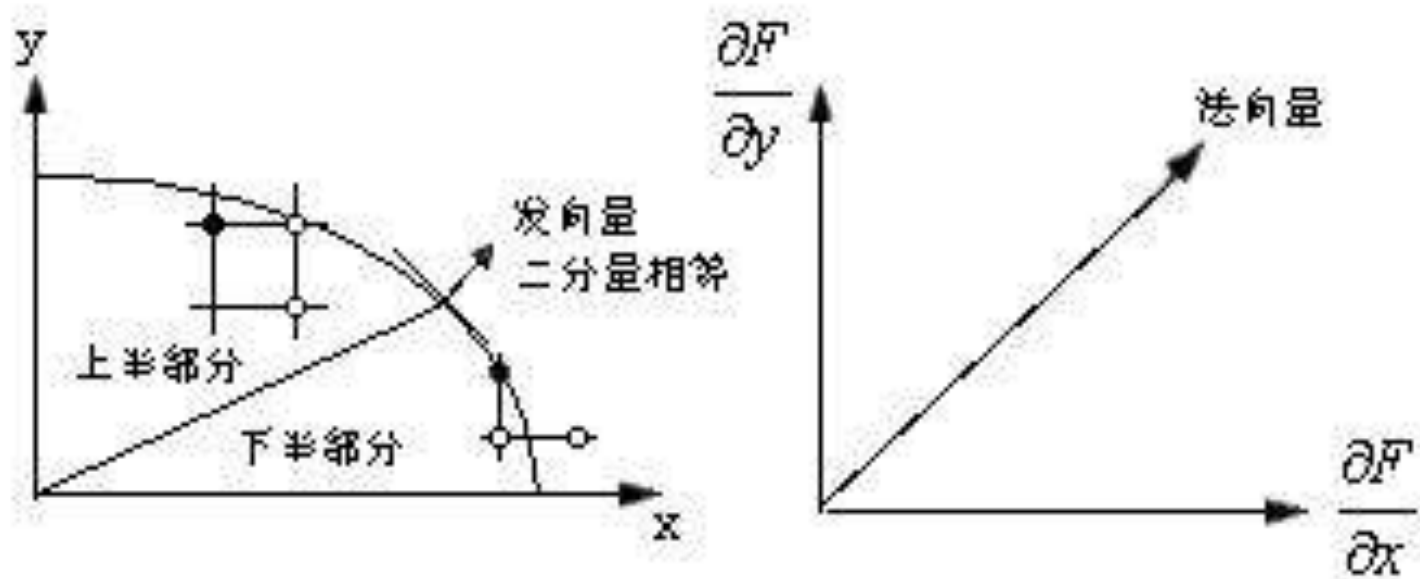


图 2.6 第一象限的椭圆弧





该椭圆上一点 (x, y) 处的法向量为

$$N(x, y) = \frac{\partial F}{\partial x} i + \frac{\partial F}{\partial y} j = 2b^2 xi + 2a^2 yj$$





其中, i 和 j 分别为沿 x 轴和 y 轴方向的单位向量。从图2.6可看出, 在上部分, 法向量的 y 分量更大, 而在下部分, 法向量的 x 分量更大, 因而, 在上部分若当前最佳逼近理想椭圆弧的像素 (x_p, y_p) 满足下列不等式

$$b^2(x_p+1) < a^2(y_p-0.5)$$

而确定的下一个像素不满足上述不等式, 则表明椭圆弧从上部分转入下部分。





在上部分，假设横坐标为 x_p 的像素中与椭圆弧更接近点是 (x_p, y_p) ，那么下一对候选像素的中点是 $(x_p+1, y_p-0.5)$ 。因此判别式为

$$d_1 = F(x_p+1, y_p-0.5) = b^2(x_p+1)^2 + a^2(y_p-0.5)^2 - a^2b^2$$

若 $d_1 < 0$ ，中点在椭圆内，则应取正右方像素，且判别式应更新为

$$\begin{aligned} d'_1 &= F(x_p+2, y_p-0.5) = b^2(x_p+2)^2 + a^2(y_p-0.5)^2 - a^2b^2 \\ &= d_1 + b^2(2x_p+3) \end{aligned}$$





当 $d_1 \geq 0$, 中点在椭圆之外, 这时应取右下方像素, 并且更新判别式为

$$\begin{aligned} d'_1 &= F(x_P+2, y_P-1.5) = b^2(x_P+2)^2 + a^2(y_P-1.5)^2 - a^2b^2 \\ &= d_1 + b^2(2x_P+3) + a^2(-2y_P+2) \end{aligned}$$

由于弧起点为 $(0, b)$, 因此, 第一中点是 $(1, b-0.5)$, 对应的判别式是

$$\begin{aligned} d_{10} &= F(1, b-0.5) = b^2 + a^2(b-0.5)^2 - a^2b^2 \\ &= b^2 + a^2(-b+0.25) \end{aligned}$$





在下部分,应改为从正下方和右下方两个像素中选择下一像素。如果在上部分所选择的最后一像素是 (x_p, y_p) , 则下部分的中点判别式 d_2 的初始值为

$$d_{20}=F(x_p+0.5, y_p-1)=b^2(x_p+0.5)^2+a^2(y_p-1)^2-a^2b^2$$

d_2 在正下方向与右下方向的增量计算与上部分类似, 这里不再赘述。下部分弧的终止条件是 $y=0$ 。





第 2 章 基本图形生成算法 (I)

第一象限椭圆弧的扫描转换中点算法的伪C描述如下:

```
void MidpointEllipse(a, b, color)
int a, b, color;
{ int x, y;
  float d1, d2;
  x=0; y=b;
  d1=b*b+a*a*(-b+0.25);
  putpixel(x, y, color);
  while(b*b*(x+1)<a*a(y-0.5))
  { if(d1<0)
    { d1+=b*b*(2*x+3);
      x++;
    }
  }
```





第 2 章 基本图形生成算法 (I)

```
else { d1+=(b*b*(2*x+3)+a*a*(-2*y+2));
```

```
    x++; y--;
```

```
}
```

```
    putpixel(x, y, color);
```

```
}/*上半部分*/
```

```
d2=sqr(b*(x+0.5))+sqr(a*(y-1))-sqr(a*b);
```

```
while(y>0)
```

```
{ if(d2<0)
```

```
{ d2+=b*b(2*x+2)+a*a*(-2*y+3);
```

```
    x++;
```

```
    y--;
```

```
}
```

```
else { d2+=a*a*(-2*y+3);
```

sqr() 为平方函数





```
y--;  
    }  
    putpixel(x, y, color);  
}  
}
```





第 2 章 基本图形生成算法 (I)



第三章 基本图形生成算法(Ⅱ)

- 如何在指定的输出设备上根据坐标描述构造基本二维几何图形（点、直线、圆、椭圆、多边形域、字符串及其相关属性等）。

图形生成的概念

- 图形的生成：是在指定的输出设备上，根据坐标描述构造二维几何图形。
- 图形的扫描转换：在光栅显示器等数字设备上确定一个最佳逼近于图形的象素集的过程。

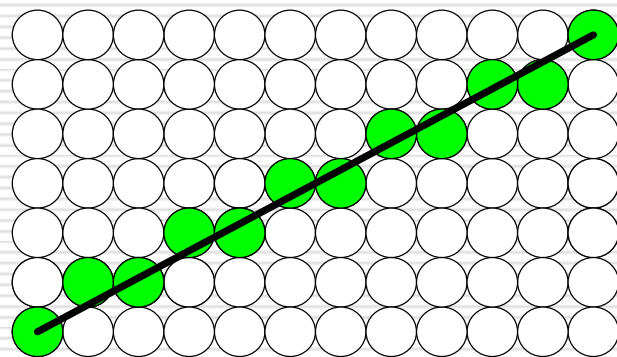


图3.1 用象素点集逼近直线

3.1 多边形的扫描转换与区域填充

- 多边形的扫描转换主要是通过确定穿越区域的扫描线的覆盖区间来填充。
- 区域填充是从给定的位置开始涂描直到指定的边界条件为止。

多边形的扫描转换与区域填充

- ☐ 多边形的扫描转换
- ☐ 边缘填充算法
- ☐ 区域填充
- ☐ 相关概念

多边形的扫描转换

- 顶点表示用多边形的顶点序列来刻划多边形。
- **点阵表示**是用位于多边形内的象素的集合来刻划多边形。
- **扫描转换多边形**：从多边形的顶点信息出发，求出位于其内部的各个象素，并将其颜色值写入帧缓存中相应单元的过程。

多边形的扫描转换

- X-扫描线算法
- 改进的有效边表算法

X-扫描线算法——原理

- 基本思想：按扫描线顺序，计算扫描线与多边形的相交区间，再用要求的颜色显示这些区间的所有像素。

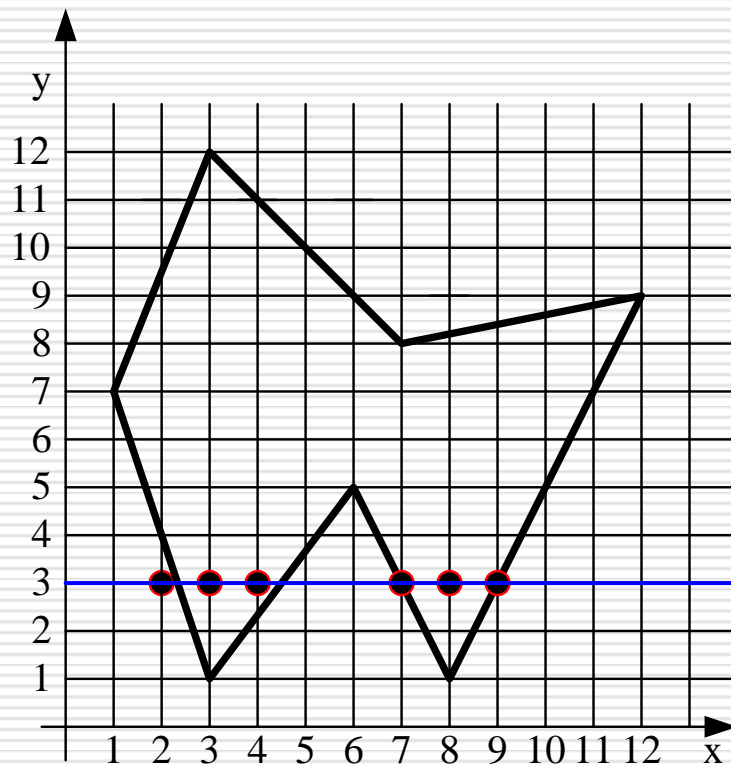


图3.2 x-扫描线算法填充多边形

X-扫描线算法——算法步骤

1. 确定多边形所占有的最大扫描线数，得到多边形顶点的最小和最大 y 值（ y_{\min} 和 y_{\max} ）。
2. 从 $y=y_{\min}$ 到 $y=y_{\max}$ ，每次用一条扫描线进行填充。
3. 对一条扫描线填充的过程可分为四个步骤：
求交；排序；交点配对；区间填色。

X-扫描线算法——取整规则

- 交点的取整规则：使生成的像素全部位于多边形之内。（用于直线等图元扫描转换的四舍五入原则可能导致部分像素位于多边形之外，从而不可用）。
- 假定非水平边与扫描线 $y=e$ 相交，交点的横坐标为 x ，规则如下：

□ **规则1**: X 为小数, 即交点落于扫描线上两个相邻像素之间时:

- 交点位于左边界之上, 向右取整;
- 交点位于右边界之上, 向左取整;

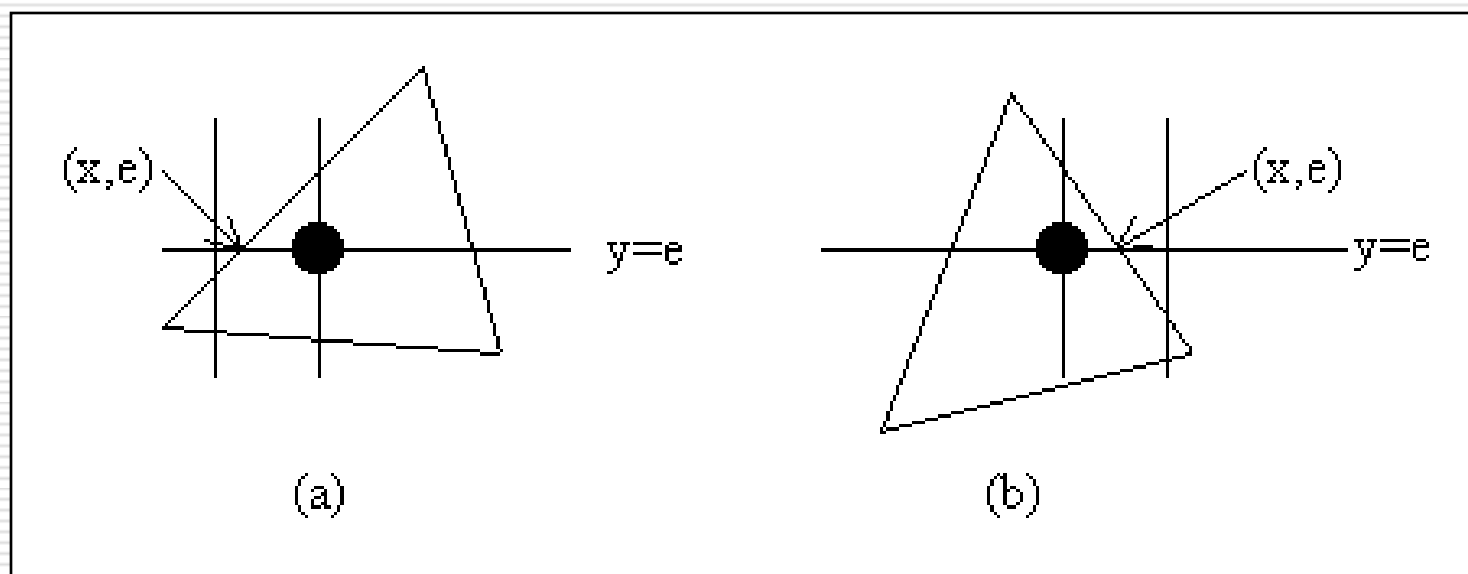


图3.3 取整规则1

X-扫描线算法——取整规则

□ **规则2**: 边界上像素的取舍问题, 避免填充扩大化。规定落在右边边界上的像素不予填充。(具体实现时, 只要对扫描线与多边形的相交区间左闭右开)

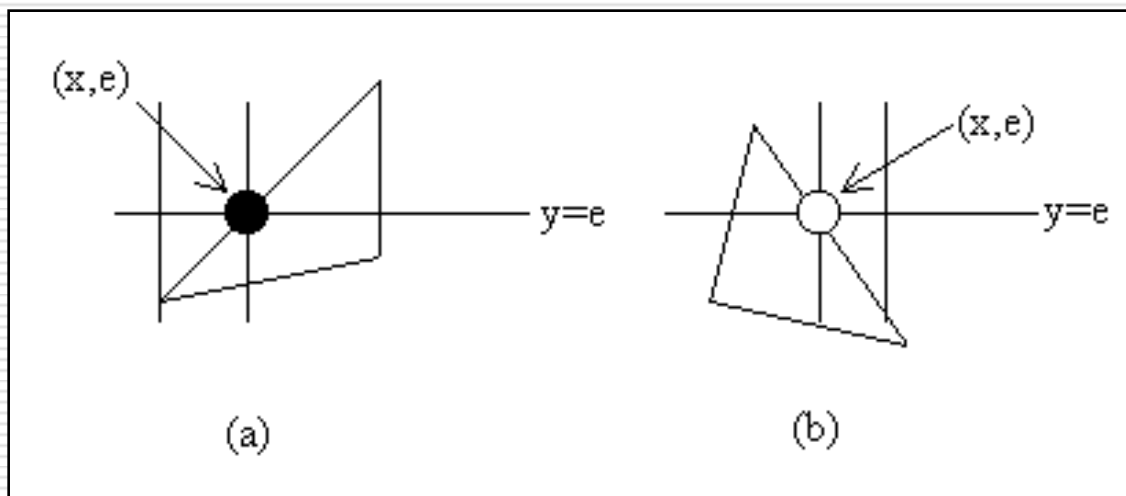


图3.4 取整规则2

X-扫描线算法——取整规则

□ **规则3**: 当扫描线与多边形顶点相交时, 交点的取舍, 保证交点正确配对。

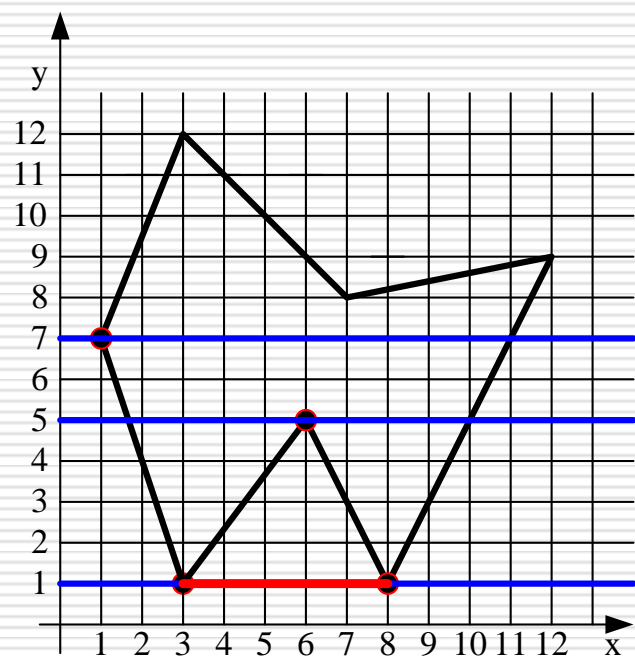


图3.5 取整规则3

X-扫描线算法——取整规则

解决方法:

当扫描线与多边形的顶点相交时,

- 若共享顶点的两条边分别落在扫描线的两边, 交点只算一个;
- 若共享顶点的两条边在扫描线的同一边, 这时交点作为零个或两个。

X-扫描线算法——取整规则

实际处理：只要检查顶点的两条边的另外两个端点的Y值，两个Y值中大于交点Y值的个数是0，1，2，来决定取0，1，2个交点。

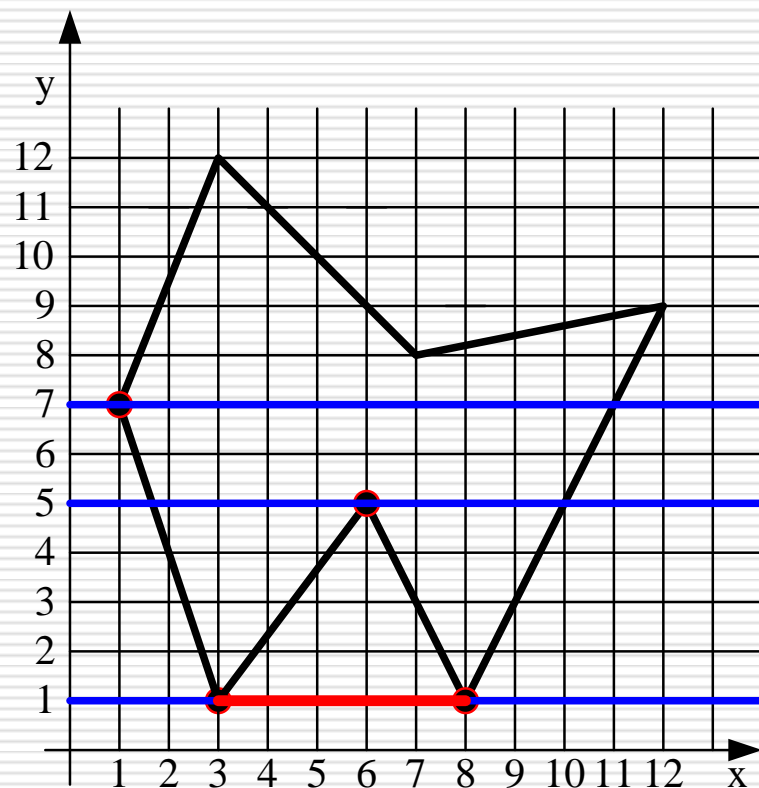


图3.6 取整规则3

X-扫描线算法——取整规则

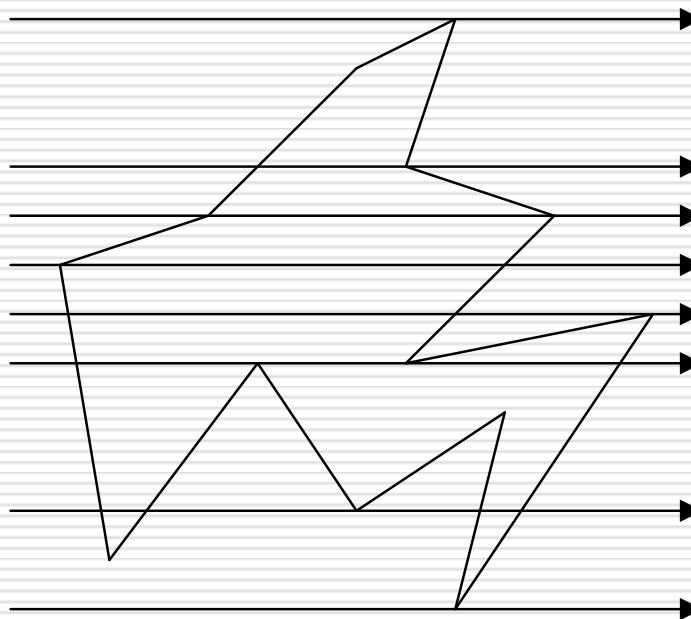


图5.21 与扫描线相交的多边形顶点的交点数

填充过程实例

改进的有效边表算法 (Y连贯性算法)

改进原理:

- 处理一条扫描线时，仅对有效边求交。
- 利用扫描线的连贯性。
- 利用多边形边的连贯性。

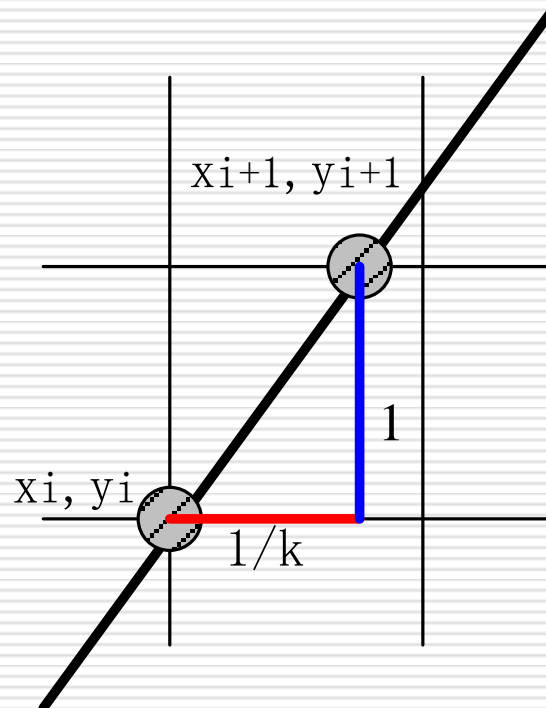


图3.7 与多边形边界相交的两条连续扫描线交点的相关性

改进的有效边表算法 (Y连贯性算法)

- 有效边 (Active Edge): 指与当前扫描线相交的多边形的边, 也称为活性边。
- 有效边表 (Active Edge Table, AET): 把有效边按与扫描线交点 x 坐标递增的顺序存放在一个链表中, 此链表称为有效边表。
- 有效边表的每个结点:

x y_{\max} $1/k$ next

改进的有效边表算法——构造边表

- 首先构造一个纵向链表，链表的长度为多边形所占有的最大扫描线数，链表的每个结点，称为一个桶，则对应多边形覆盖的每一条扫描线。
- 将每条边的信息链入与该边最小 y 坐标 (y_{\min}) 相对应的桶处。也就是说，若某边的较低端点为 y_{\min} ，则该边就放在相应的扫描线桶中。

改进的有效边表算法——构造边表

- 每条边的数据形成一个结点，内容包括：该扫描线与该边的初始交点 x （即较低端点的 x 值）， $1/k$ ，以及该边的最大 y 值 y_{\max} 。

$x|_{y_{\min}} \quad y_{\max} \quad 1/k \quad \text{NEXT}$

- 同一桶中若干条边按 $x|_{y_{\min}}$ 由小到大排序，若 $x|_{y_{\min}}$ 相等，则按照 $1/k$ 由小到大排序。

解决顶点交点计为1时的情形:

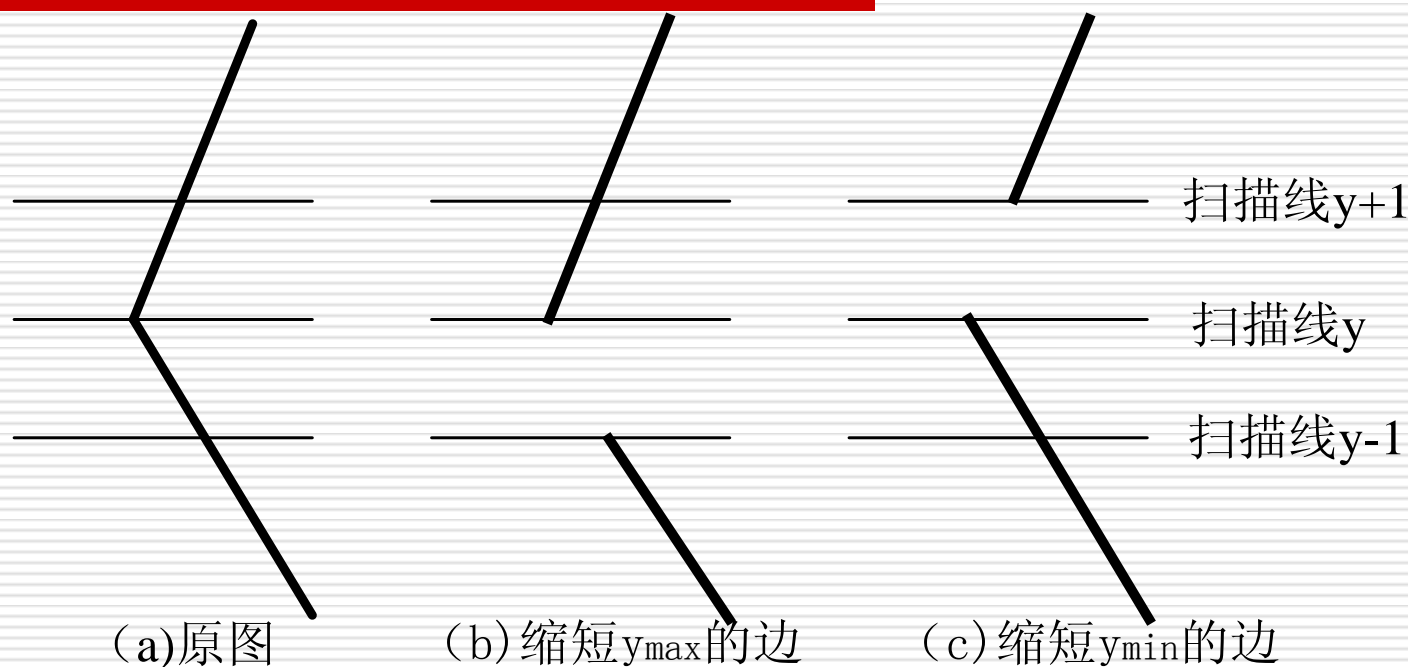


图3.8 将多边形的某些边缩短以分离那些应计为1个交点的顶点

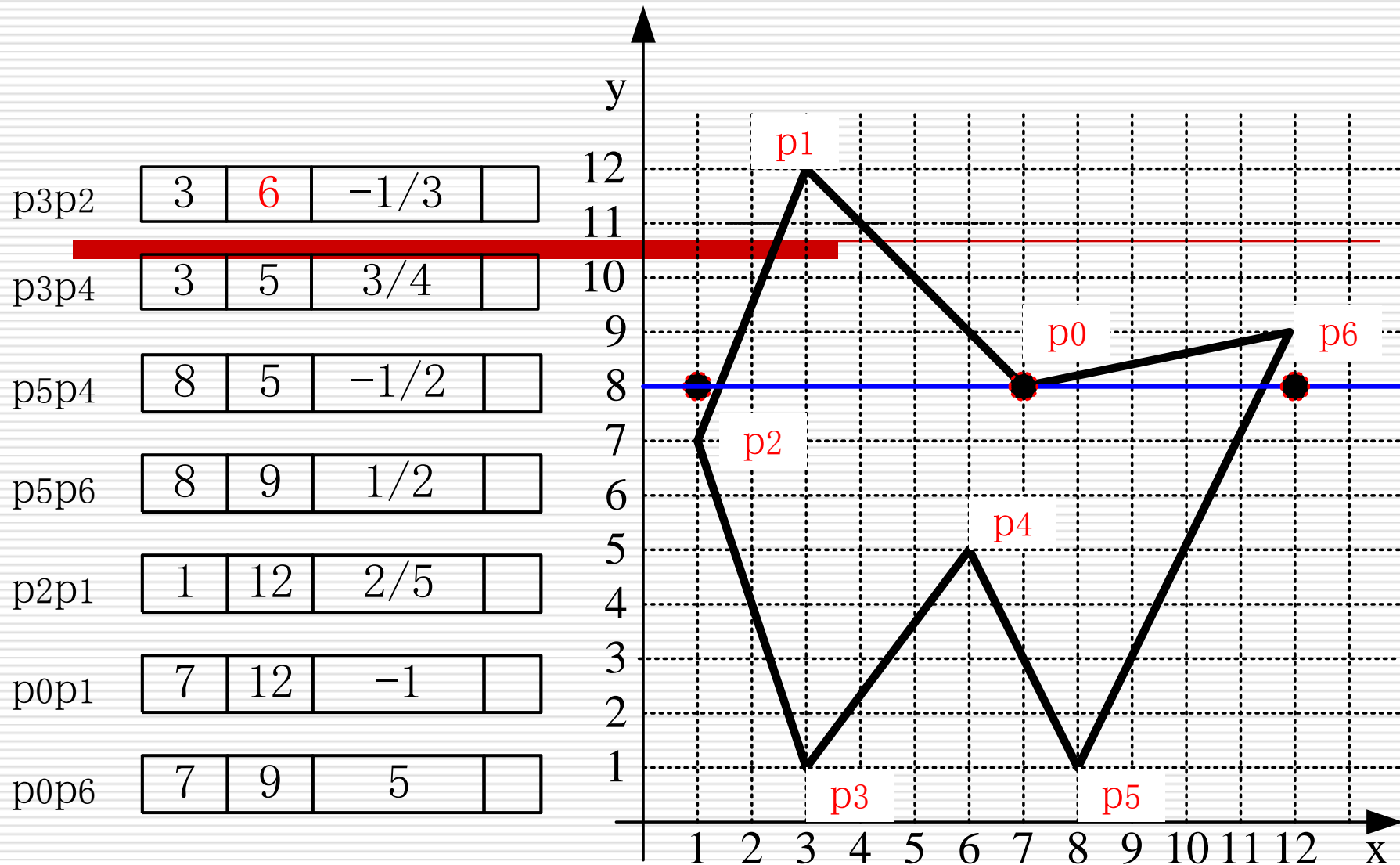


图3.9 多边形 $P_0P_1P_2P_3P_4P_5P_6$

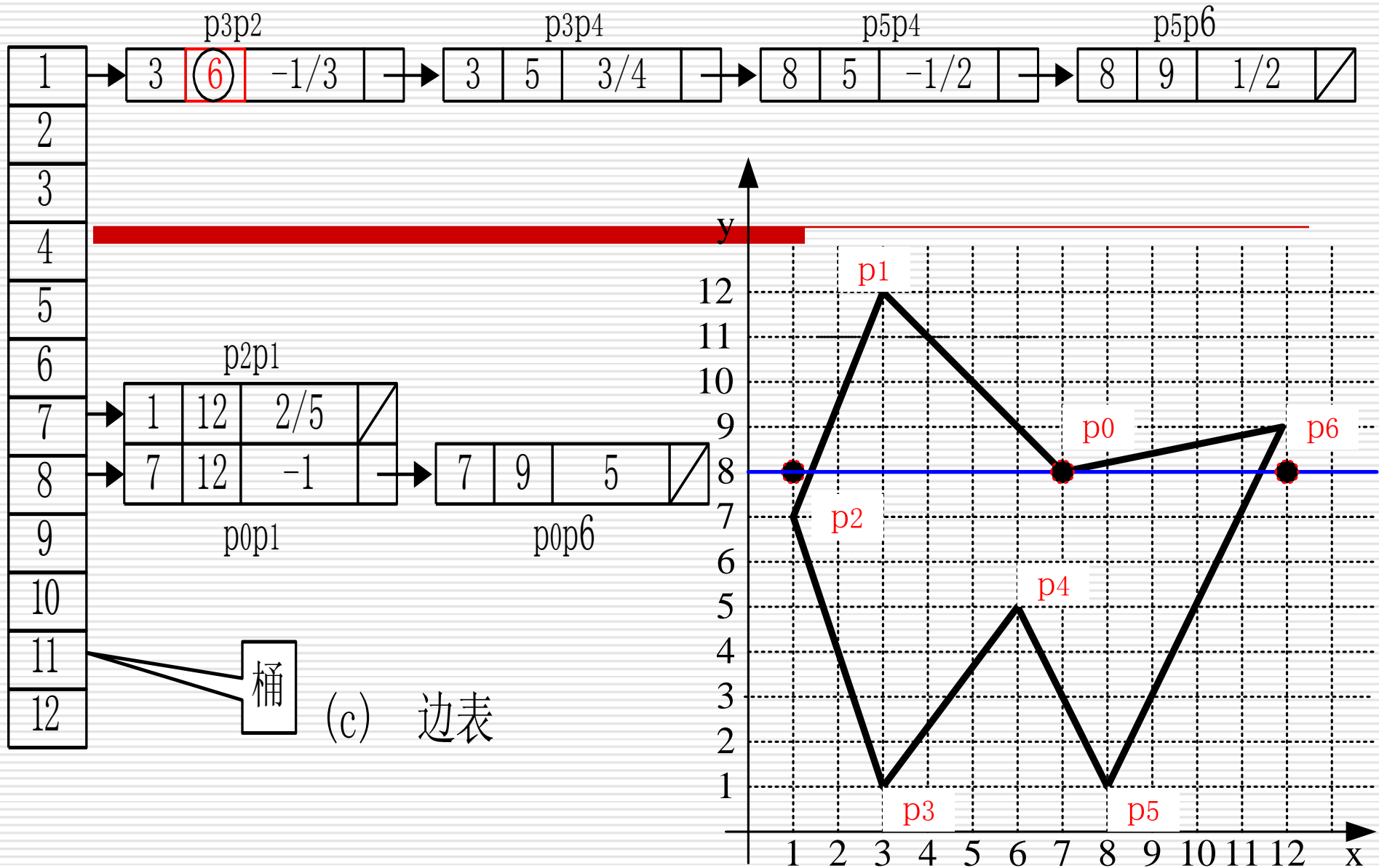


图3.10 多边形 $P_0P_1P_2P_3P_4P_5P_6$

改进的有效边表算法——算法步骤

- (1)初始化：构造边表，AET表置空；
- (2)将第一个不空的ET表中的边与AET表合并；
- (3)由AET表中取出交点对进行填充。填充之后删除 $y=y_{\max}$ 的边；
- (4) $y_{i+1}=y_i+1$,根据 $x_{i+1}=x_i+1/k$ 计算并修改AET表，同时合并ET表中 $y=y_{i+1}$ 桶中的边，按次序插入到AET表中，形成新的AET表；
- (5)AET表不为空则转(3)，否则结束。

边缘填充算法

□ 基本思想：按任意顺序处理多边形的每条边。

处理时，先求出该边与扫描线的交点，再对扫描线上交点右方的所有像素取反。

□ 算法简单，但对于复杂图型，每一像素可能被访问多次

栅栏填充算法

- ❑ 栅栏指的是一条过多边形顶点且与扫描线垂直的直线。它把多边形分为两半。
- ❑ 基本思想：按任意顺序处理多边形的每一条边，但处理每条边与扫描线的交点时，将交点与栅栏之间的像素取反。
- ❑ 这种算法尽管减少了被重复访问像素的数目，但仍有一些像素被重复访问。

边标志算法

- 基本思想：先用特殊的颜色在帧缓存中将多边形的边界勾画出来，然后将着色的象素点依 x 坐标递增的顺序配对，再把每一对象素构成的区间置为填充色。
- 分为两个步骤：打标记-多边形扫描转化；填充。
- 当用软件实现本算法时，速度与改进的有效边表算法相当，但本算法用硬件实现后速度会有很大提高。

区域填充（种子填充）

- 基本概念
- 区域的表示方法
- 区域的分类
- 区域填充算法

基本概念

- **区域填充**是指从区域内的某一个像素点（种子点）开始，由内向外将填充色扩展到整个区域内的过程。
- **区域**是指已经表示成点阵形式的填充图形，它是相互连通的一组像素的集合。

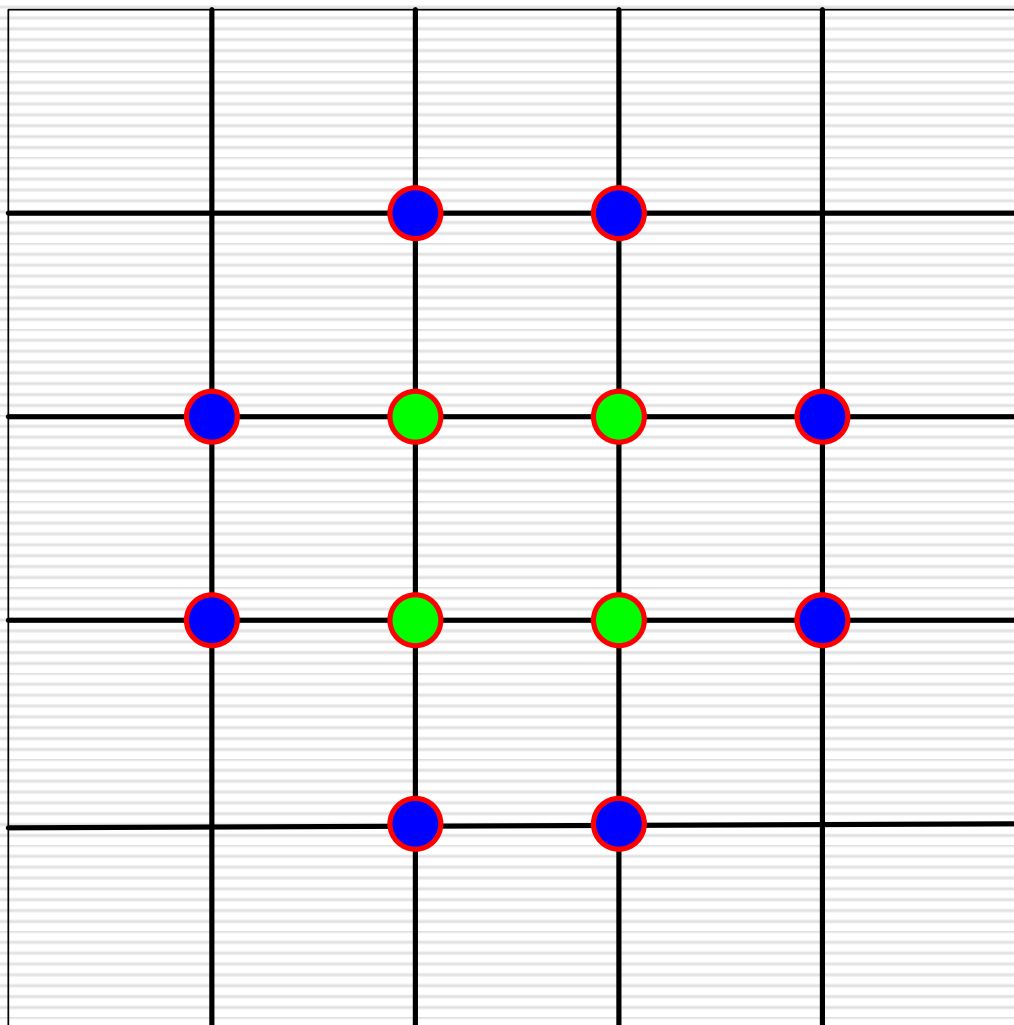


图3.12 区域的概念

区域的表示方法

- **边界表示法**：把位于给定区域的边界上的象素一一列举出来的方法。
- 边界表示法中，由于边界由特殊颜色指定，填充算法可以逐个象素地向外处理，直到遇到边界颜色为止，这种方法称为边界填充算法（Boundary-fill Algorithm）。

□ **内点表示法**：枚举出给定区域内所有象素的表示方法。以内点表示法为基础的区域填充算法称为**泛填充算法（Flood-fill Algorithm）**。

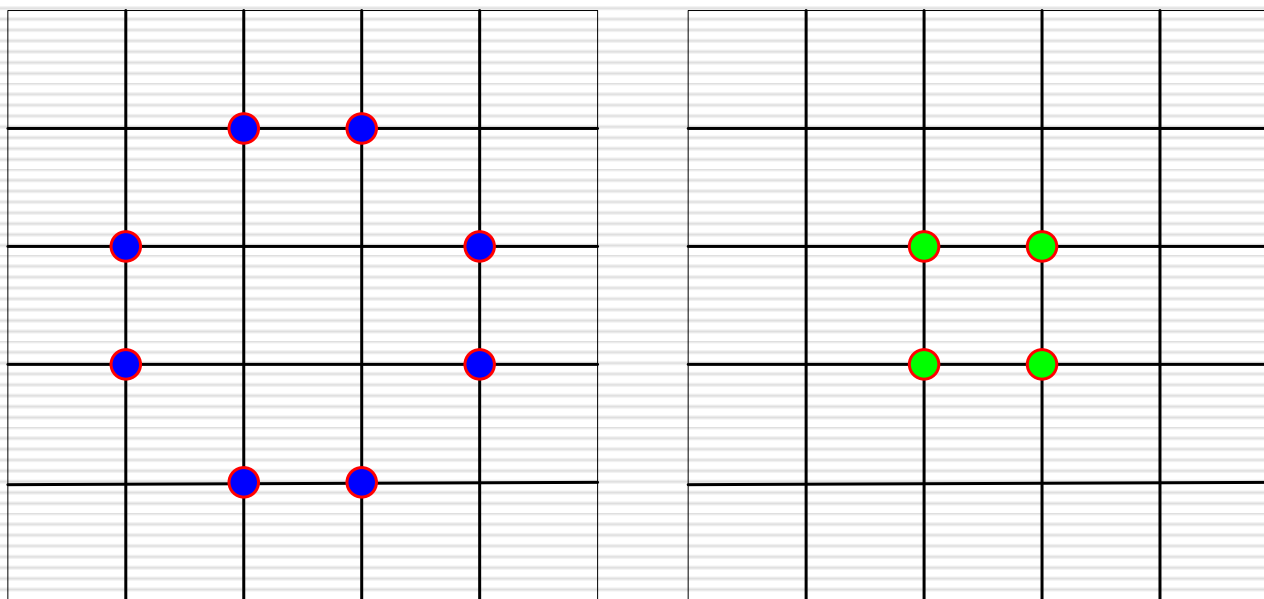
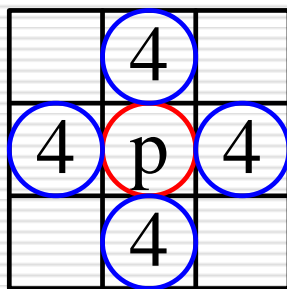


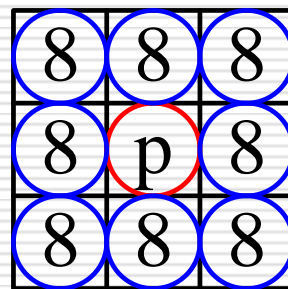
图3-13 区域的表示方法

区域的分类

4-连通区域，8-连通区域



(a) 4-邻接点



(b) 8-邻接点

图3-14 4-邻接点与8-邻接点

区域的分类

- **4-连通区域**: 从区域上的一点出发, 通过访问已知点的**4-邻接点**, 在不越出区域的前提下, 遍历区域内的所有象素点。
- **8-连通区域**: 从区域上的一点出发, 通过访问已知点的**8-邻接点**, 在不越出区域的前提下, 遍历区域内的所有象素点。

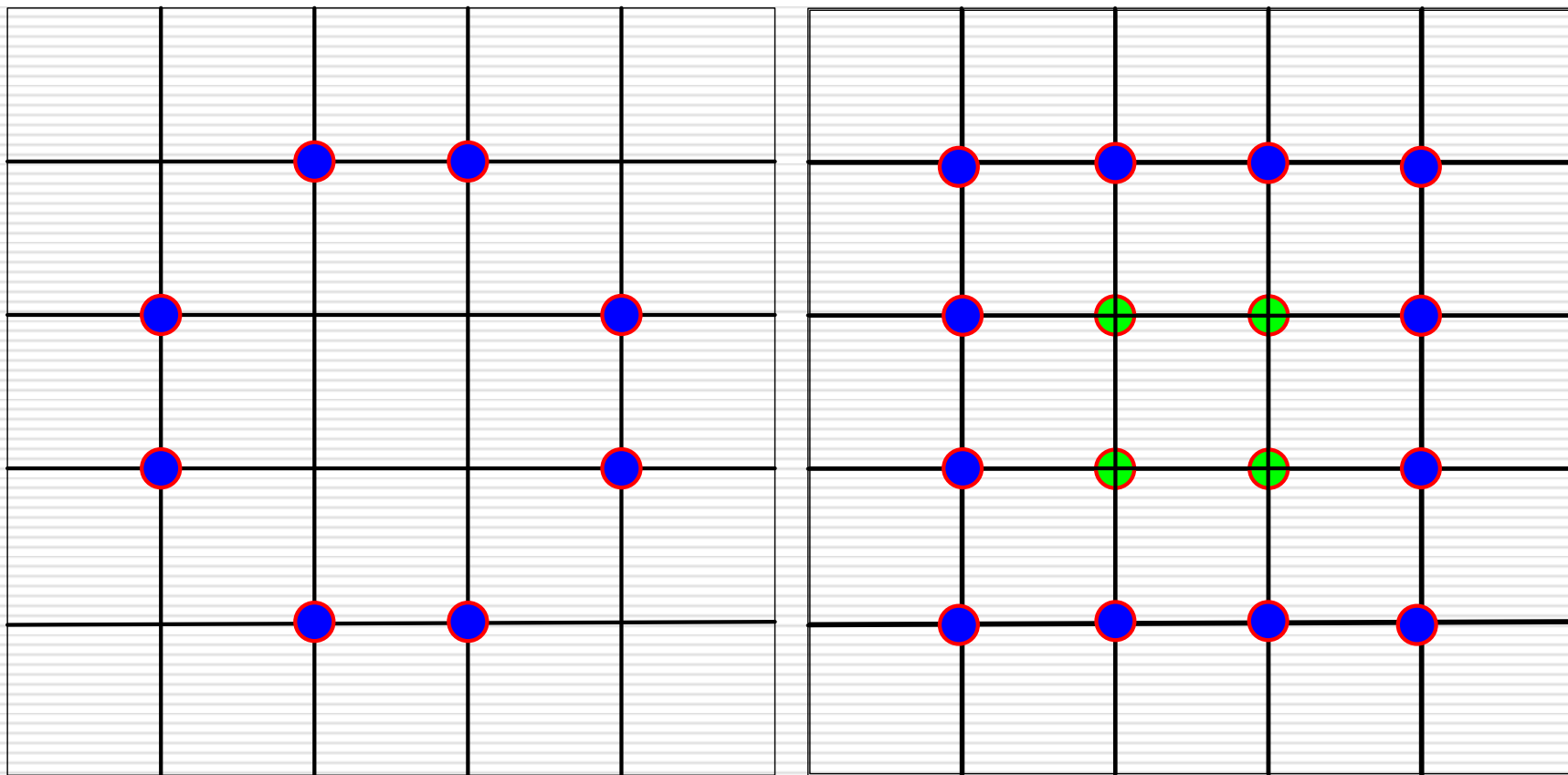


图3.15 4-连通与8-连通区域

4连通与8连通区域的区别

- 连通性： 4连通可看作8连通区域，但对边界有要求。
- 对边界的要求。

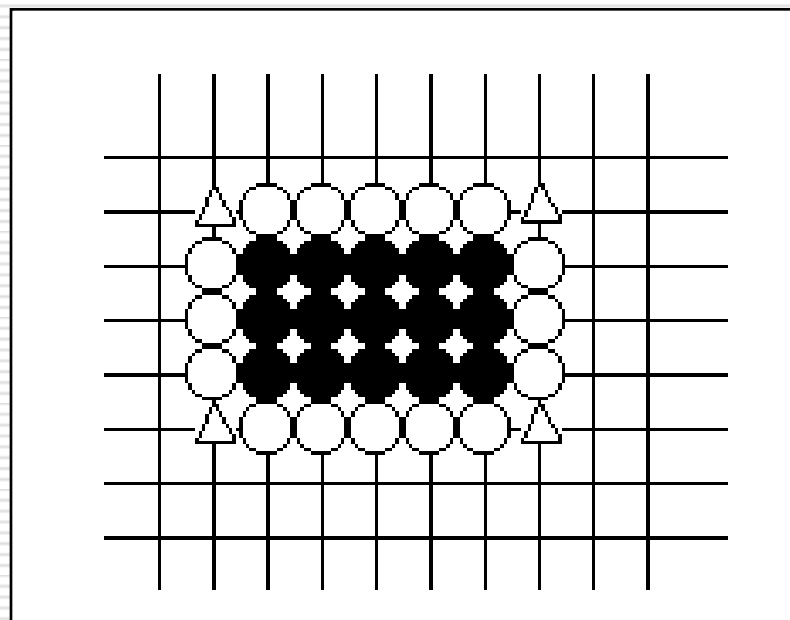


图3-164 - 连通与8 - 连通区域

区域填充算法

- 区域填充算法（边界填充算法和泛填充算法）
是根据区域内的一个已知像素点（种子点）出发，找到区域内其他像素点的过程，所以把这一类算法也成为种子填充算法。

区域填充算法——边界填充算法

- ❑ 算法的输入：种子点坐标 (x, y) ，填充色以及边界颜色。
- ❑ 利用堆栈实现简单的种子填充算法

算法从种子点开始检测相邻位置是否是边界颜色，若不是就用填充色着色，并检测该像素点的相邻位置，直到检测完区域边界颜色范围内的所有像素为止。

区域填充算法——边界填充算法

栈结构实现4-连通边界填充算法的算法步骤为：

种子像素入栈；当栈非空时重复执行如下三步操作：

(a)栈顶像素出栈；

(b)将出栈像素置成填充色；

(c)检查出栈像素的4-邻接点，若其中某个像素点不是边界色且未置成多边形色，则把该像素入栈。

区域填充算法——边界填充算法

栈结构实现8-连通边界填充算法的算法步骤为：

种子像素入栈；当栈非空时重复执行如下三步操作：

(a)栈顶像素出栈；

(b)将出栈像素置成填充色；

(c)检查出栈像素的8-邻接点，若其中某个像素点不是边界色且未置成多边形色，则把该像素入栈。

区域填充算法——边界填充算法

- ❑ 可以用于填充带有内孔的平面区域。
- ❑ 把太多的像素压入堆栈，降低了效率，同时需要较大的存储空间。
- ❑ 递归执行，算法简单，但效率不高，区域内每一像素都引起一次递归，进/出栈费时费内存。
- ❑ 通过沿扫描线填充水平像素段，来代替处理4-邻接点和8-邻接点。

区域填充算法——边界填充算法

- 扫描线种子填充算法：
扫描线通过在任意不间断扫描线区间中只取一个种子像素的方法使堆栈的尺寸极小化。不间断区间是指在一条扫描线上的一组相邻像素。

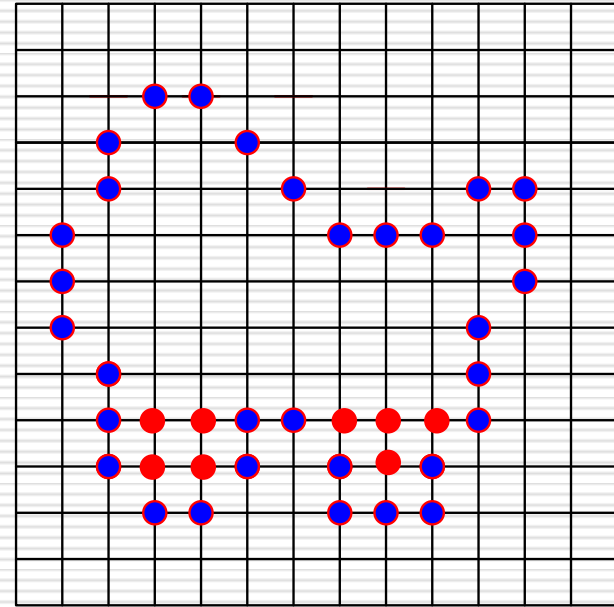


图3-17 扫描线种子填充算法

区域填充算法——边界填充算法

- 基本过程：当给定种子点时，首先填充种子点所在的扫描线上的位于给定区域的一个区段，然后确定与这一区段相通的上下两条扫描线上位于给定区域内的区段，并依次保存下来。反复这个过程，直到填充结束。

区域填充算法——边界填充算法

- 扫描线种子填充算法：我们可以在任意一个扫描线与多边形的相交区间中，只取一个种子像素，并将种子像素入栈，当栈非空时作如下四步操作：

- (1) 栈顶像素出栈;
- (2) 沿扫描线对出栈像素的左右像素进行填充, 直至遇到边界像素为止, 也就是对包含出栈像素的整个区间进行填充;
- (3) 上述区间内最左最右的像素分别记为 x_l 和 x_r ;
- (4) 在区间 $[x_l, x_r]$ 中检查与当前扫描线相邻的上下两条扫描线的有关像素是否全为边界像素或已填充的像素, 若存在非边界、非填充的像素, 则把每一区间的最右像素入栈。

上述改进之后的算法称之为扫描线种子填充算法，其伪C语言描述如下：

```
void scanline-seed-fill(polydef, color, x, y)
    多边形定义 polydef;
    int color, x, y; /* (x, y)为种子像素*/
{
    int x, y, x0, x1, xr
    push(seed(x, y)); /* 种子像素x栈 */
```

while (栈非空)

```
{ pop (pixel(x, y)); /* 栈顶像素出栈, 并置为
    (x, y) */
    putpixel(x, y, color);
    x0=x+1;
    while (getpixel(x0, y)的值不等于边界像素颜色值) /* 填充出栈像素的右方像素*/
        { putpixel(x0, y, color);
            x0=x0+1;
        }
    xr=x0-1; /*最右像素*/
    x0=x-1;
```

```
while(getpixel(x0, y) 的值不等于边界像素颜色值)
    /* 填充出栈像素的左方 像素*/
        { putpixel(x0, y, color);
          x0=x0-1;
        }
    x1=x0+1; /*最左像素*/
    /* 检查上一条扫描线, 若存在非边界且未填充
    的像素, 则将各连续区间的最右像素入栈 */
    x0=x1;  y=y+1;
```

```

while(x0<=xr)
{
    flag=0;
    while((getpixel(x0, y)的值不等于边界像素颜色值)&& (getpixel(x0, y)的值不等于多边形内像素颜色值)&& (x0<xr))
    {
        if (flag==0) flag=1;
        x0++;
    }
    if(flag==1)
    {
        if((x0==xr)&&(getpixel(x0, y)的值不等于边界像素颜色值)&& (getpixel(x0, y)的值不等于多边形内像素颜色值))    push((x0, y));
    }
}

```



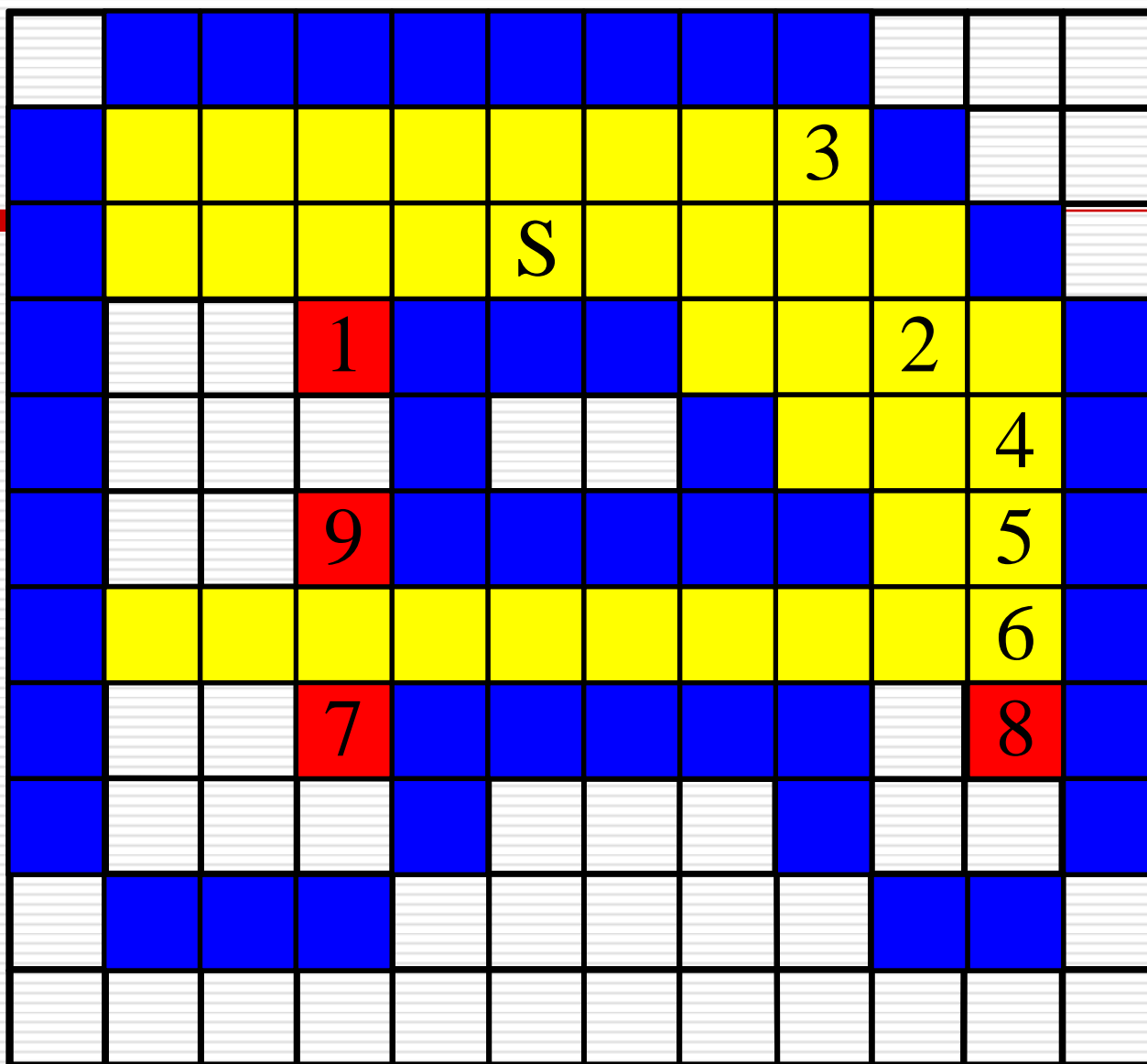
```

else    push((x0-1, y));
        flag=0;
    }
    xnextspan = x0;
    while(( pixel(x0, y1) 等于边界像素颜色值
) || (pixel(x0, y1) 等于多边形内像素颜色值
)&&(x0<=xr))
        x0++;
        if(xnextspan ==x0)      x0++;
} /* while(x0<=xr) */

```

/* 检查下一条扫描线，若存在非边界，未填充的像素，则将各连续区间的最右像素入栈；处理与前面一条扫描线的算法相同，只要把 $y+1$ 换为 $y-1$ 即可 */

```
    } /* while (栈非空) */  
}
```



相关概念——内外测试

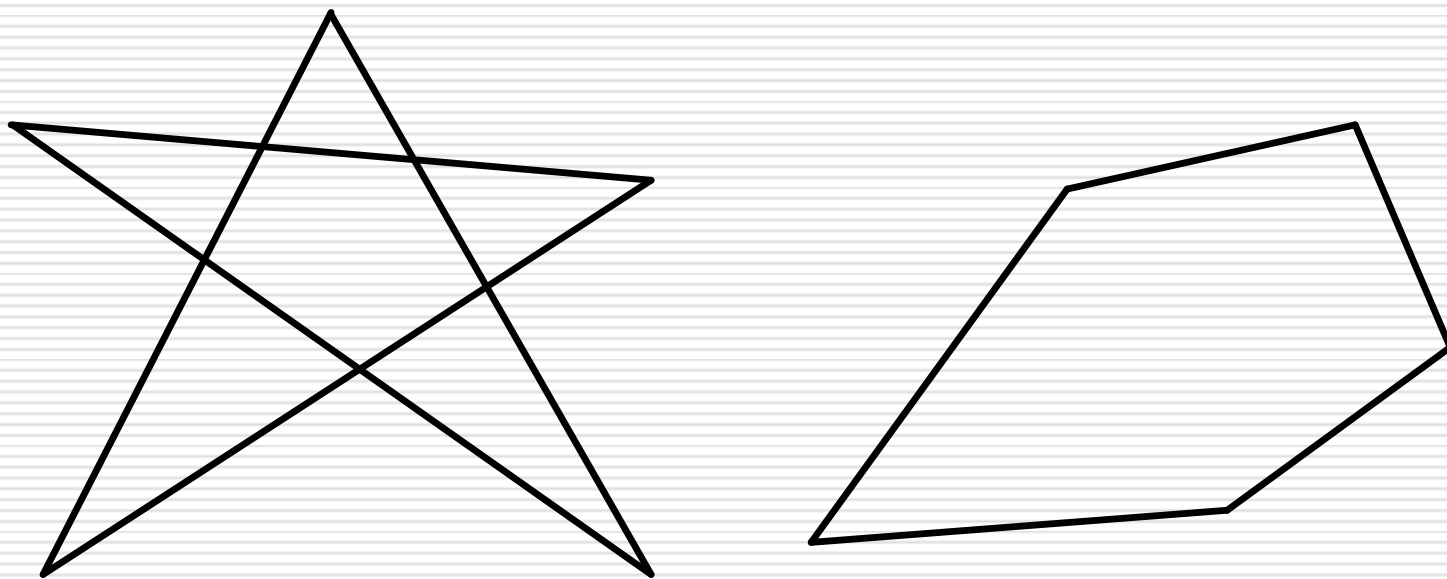
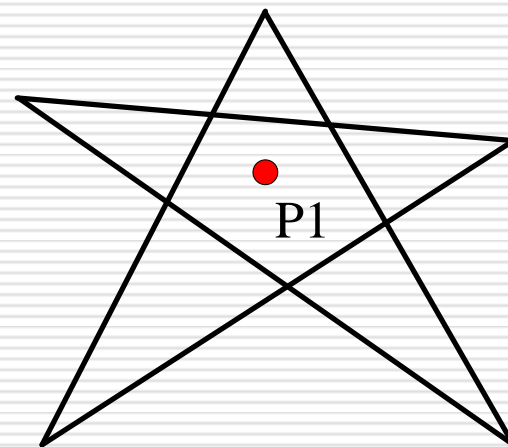


图3.19 不自交的多边形与自相交的多边形

相关概念——内外测试

□ 奇-偶规则 (Odd-even Rule)

从任意位置 p 作一条射线，若与该射线相交的多边形边的数目为奇数，则 p 是多边形内部点，否则是外部点。



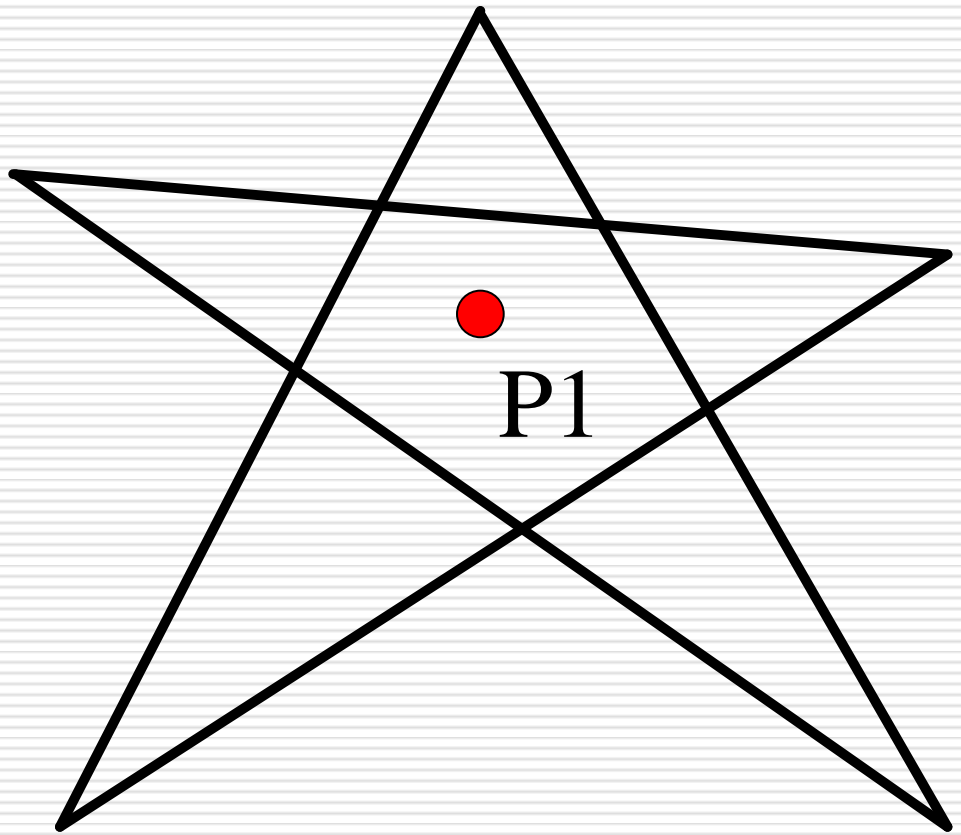
相关概念——内外测试

非零环绕数规则 (Nonzero Winding Number Rule)

- 首先使多边形的边变为矢量。
- 将环绕数初始化为零。
- 再从任意位置 p 作一条射线。当从 p 点沿射线方向移动时，对在每个方向上穿过射线的边计数，每当多边形的边从右到左穿过射线时，环绕数加1，从左到右时，环绕数减1。
- 处理完多边形的所有相关边之后，若环绕数为非零，则 p 为内部点，否则， p 是外部点。

相关概念——内外测试

□ 两种规则的比较



相关概念——曲线边界区域填充

□ 相交计算中包含了非线性边界。

□ 对于简单曲线：

（1）计算曲线相对两侧的两个扫描线交点

（2）简单填充在曲线两侧上的边界点间的
水平像素区间。

3.2 字符处理

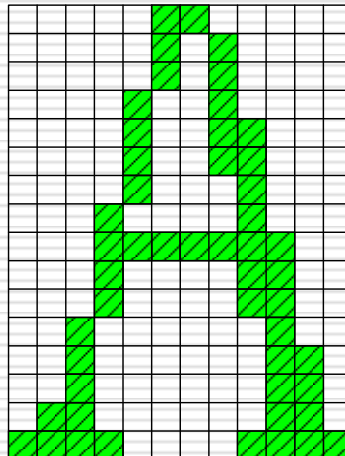
- ❑ **ASCII 码**：“美国信息交换用标准代码集”（**American Standard Code for Information Interchange**），简称**ASCII**码。
- ❑ **国际码**：“中华人民共和国国家标准信息交换编码，简称为国际码，代号**GB2312-80**。
- ❑ **字库**：字库中储存了每个字符的图形信息。
- ❑ **矢量字库和点阵字库**。

字符处理——点阵字符

- 在点阵表示中，每个字符由一个点阵位图来表示。
- 显示时：形成字符的像素图案。

0	0	0	0	0	1	1	0	0	0	0	0
0	0	0	0	0	1	0	1	0	0	0	0
0	0	0	0	0	1	0	1	0	0	0	0
0	0	0	0	1	0	0	1	0	0	0	0
0	0	0	0	1	0	0	1	1	0	0	0
0	0	0	0	1	0	0	1	1	0	0	0
0	0	0	0	1	0	0	0	1	0	0	0
0	0	0	1	0	0	0	0	1	0	0	0
0	0	0	1	1	1	1	1	1	1	0	0
0	0	0	1	0	0	0	0	1	1	0	0
0	0	0	1	0	0	0	0	1	1	0	0
0	0	1	0	0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	0	1	1	0
0	0	1	0	0	0	0	0	0	1	1	0
0	1	1	0	0	0	0	0	0	1	1	0
1	1	1	1	0	0	0	0	1	1	1	1

(a)字符A的点阵位图



(a)字符A的像素图案

图3.20 字符A的点阵表示

字符处理——点阵字符

- 定义和显示直接、简单。
- 存储需要耗费大量空间。
 - 从一组点阵字符生成不同尺寸和不同字体的其他字符。
 - 采用压缩技术。

字符处理——矢量字符

- 矢量字符采用直线和曲线段来描述字符形状，矢量字符库中记录的是笔划信息。
- 显示时：解释字符的每个笔划信息。

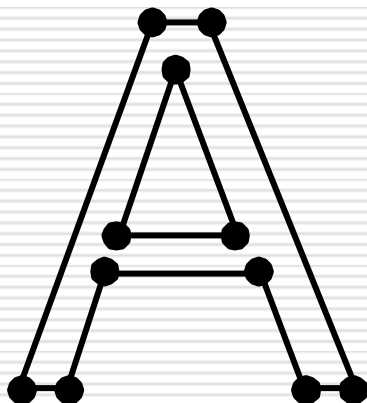


图3.21 字符A的矢量表示

字符处理——矢量字符

- 轮廓字型法采用直线或二、三次曲线的集合来描述一个字符的轮廓线，轮廓线构成了一个或若干个封闭区域，显示时只要填充这些区域就可产生相应的字符。
- 显示时可以“无极放缩”。
- 占用空间少，美观，变换方便。

3.3 属性处理

- 图素或图段的外观由其属性决定。
- 图形软件中实现属性选择的方法是提供一张系统当前属性值表，通过调用标准函数提供属性值表的设置和修改。进行扫描转换时，系统使用属性值表中的当前属性值进行显示和输出。

属性处理

- 线型与线宽
- 区域填充属性

线型与线宽——线型

- 线型的显示可用象素段方法实现：针对不同的线型，画线程序沿路径输出一些连续象素段，在每两个实心段之间有一段中间空白段，他们的长度（象素数目）可用象素模板(pixel mask)指定。
- 存在问题：如何保持任何方向的划线长度近似地相等。

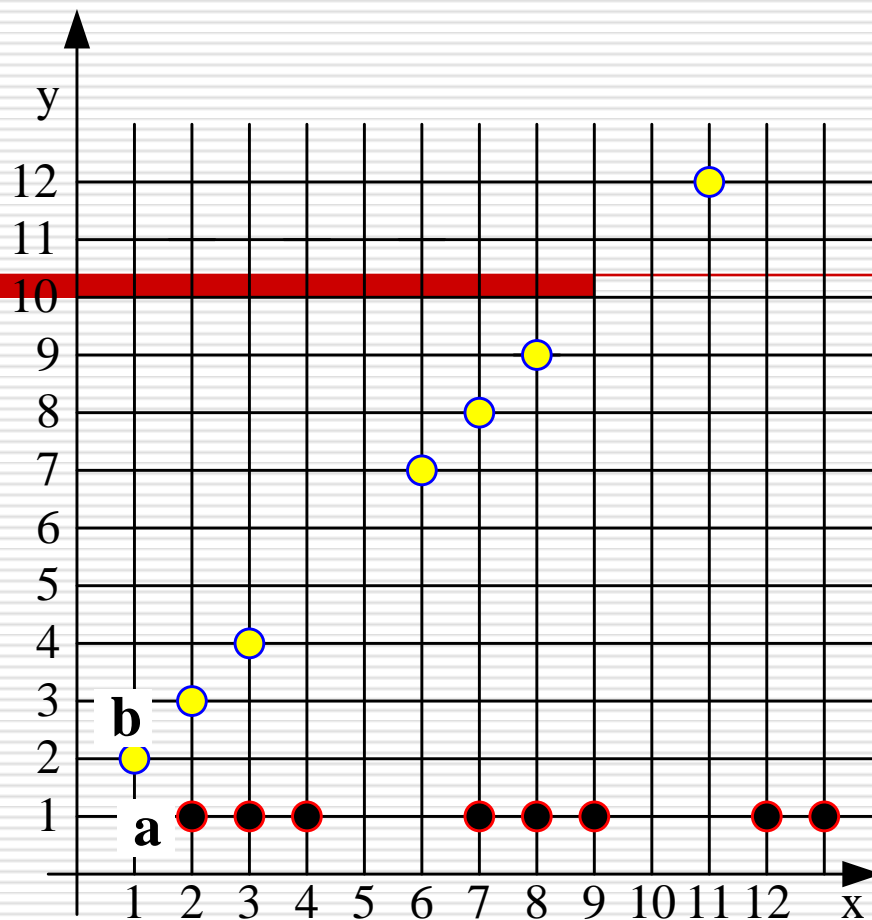


图3.22 相同数目象素显示的不等长划线

□ 可根据线的斜率来调整实心段和中间空白段的像素数目。

线型与线宽——线宽

□ 线刷子：垂直刷子、水平刷子

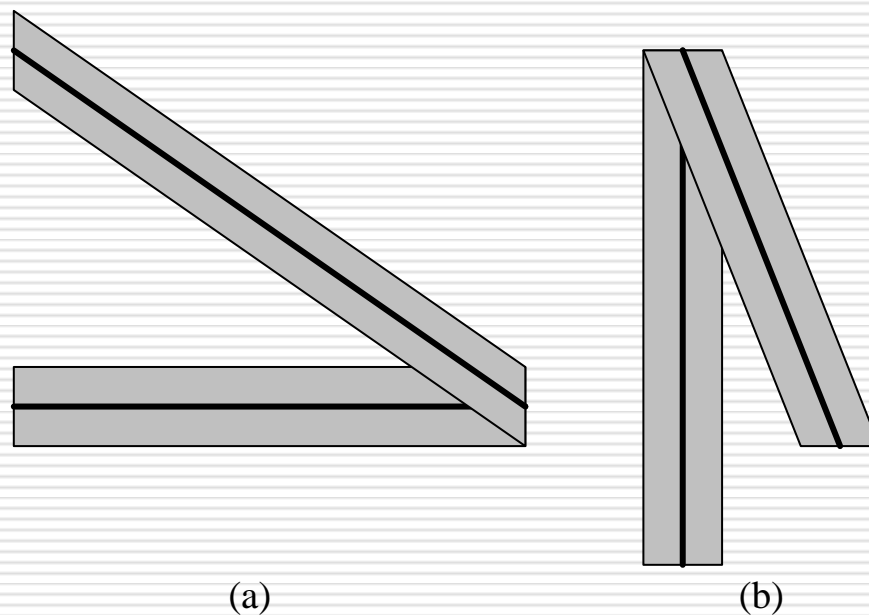


图3.23 线刷子

线型与线宽——线型

- ❑ 实现简单、效率高。
- ❑ 斜线与水平(或垂直)线不一样粗。
- ❑ 当线宽为偶数个像素时，线的中心将偏移半个像素。
- ❑ 利用线刷子生成线的始末端总是水平或垂直的，看起来不太自然。

解决：添加“线帽（line cap）”

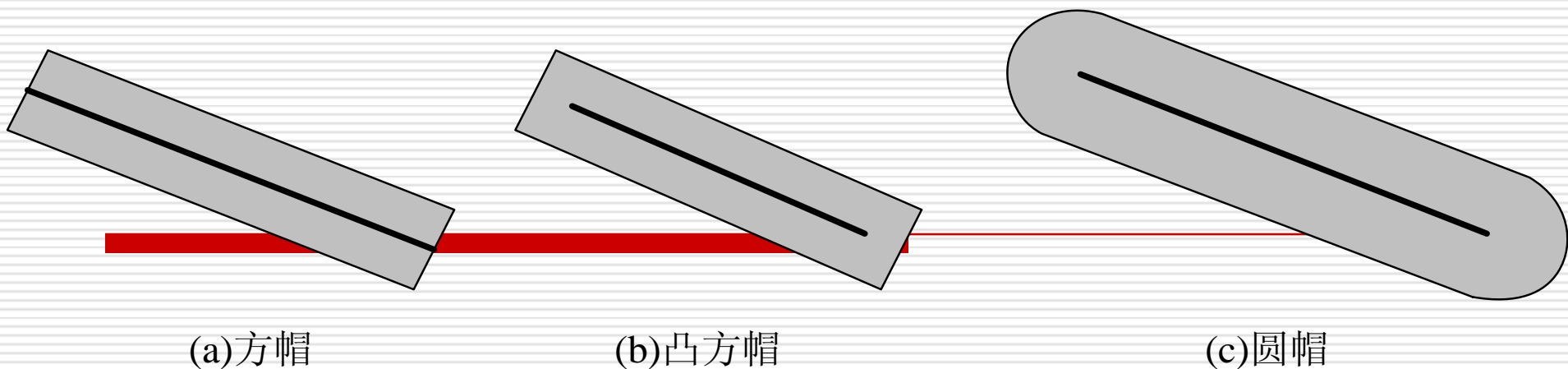


图3.24 线帽

- 方帽：调整端点位置，使粗线的显示具有垂直于线段路径的正方形端点。
- 凸方帽：简单将线向两头延伸一半线宽并添加方帽。
- 圆帽：通过对每个方帽添加一个填充的半圆得到。

线型与线宽——线型

- 当比较接近水平的线与比较接近垂直的线汇合时，汇合处外角将有缺口。

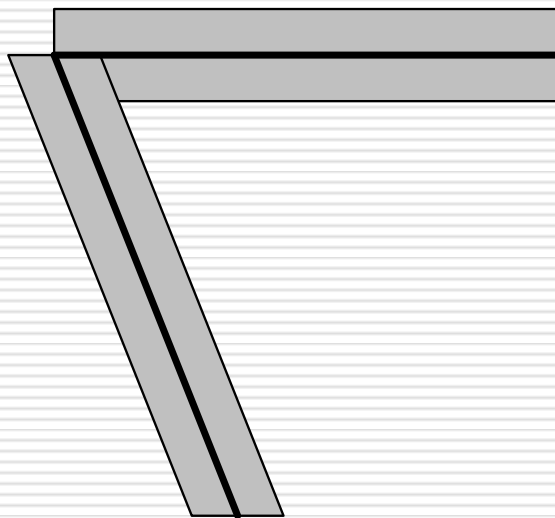
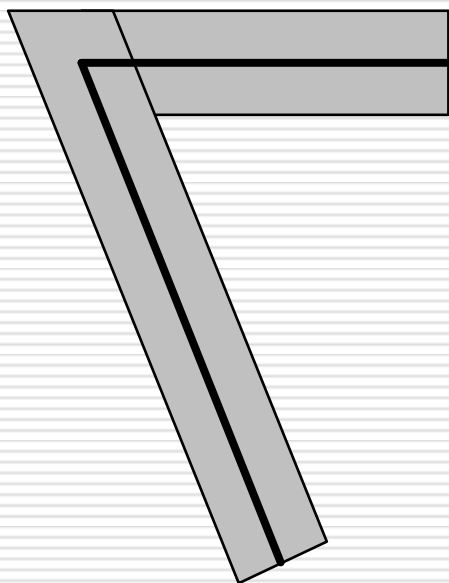
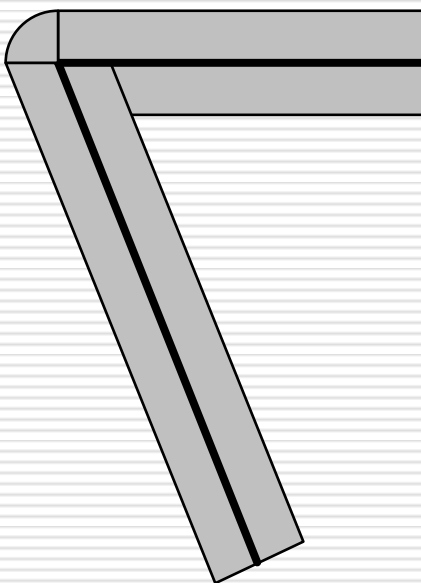


图3.25 线刷子产生的缺口

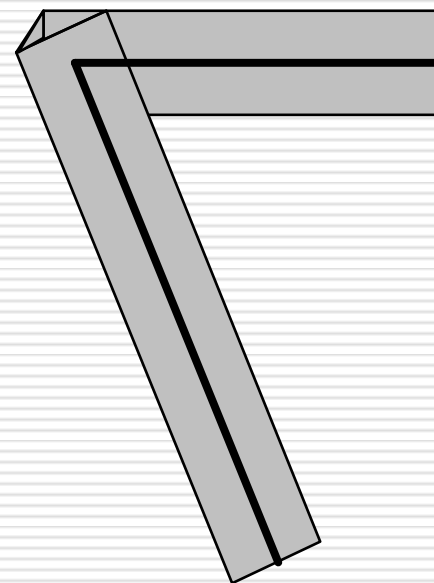
□ 解决：斜角连接 (miter join)、圆连接
~~(round join)、斜切连接 (bevel join)~~



(a)斜角连接



(b)圆连接



(c)斜切连接

图3.26 线刷子产生的缺口

方刷子

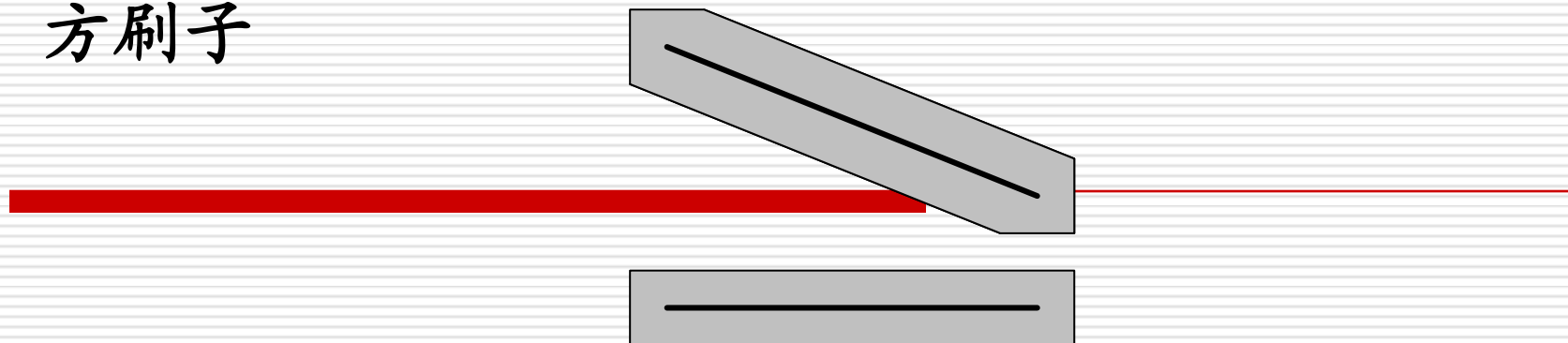


图3.27 方刷子

特点:

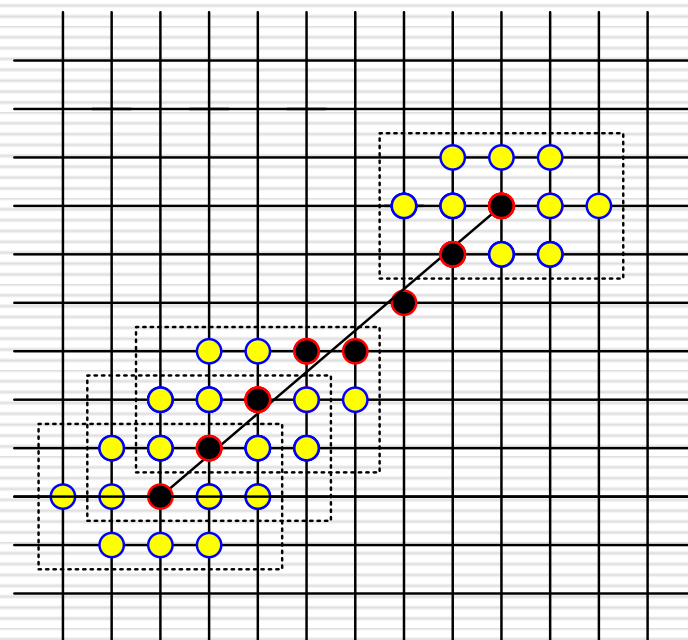
- ❑ 方刷子绘制的线条（斜线）比用线刷子所绘制的线条要粗一些。
- ❑ 方刷子绘制的斜线与水平（或垂直）线不一样粗。
- ❑ 方刷子绘制的线条自然地带有一个“方线帽”。

线型与线宽——线型

- 区域填充
- 改变刷子形状

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

(a) 像素模板



(b) 用该模板进行线宽处理

图3.28 利用像素模板进行线宽处理

线型与线宽——曲线的线型和线宽

□ 线型：可采用像素模板的方法。

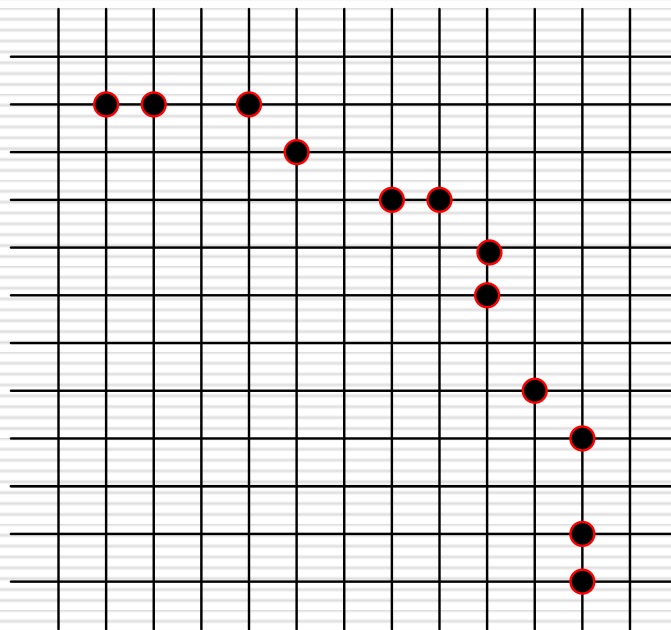


图3.29 利用模板110进行圆的线型处理

线型与线宽——曲线的线型和线宽

- 从一个八分象限转入下一个八分象限时要交换像素的位置，以保持划线长度近似相等。
- 在沿圆周移动时调整画每根划线的像素数目以保证划线长度近似相等。
- 改进：可以采用沿等角弧画像素代替用等长区间的像素模板来生成等长划线。

线型与线宽——曲线的线型和线宽

线宽:

□ 线刷子: 经过曲线斜率为1和-1处, 必须切换刷子。

曲线接近水平与垂直的地方, 线条更粗。

□ 方刷子: 接近水平垂直的地方, 线条更细

要显示一致的曲线宽度可通过旋转刷子方向以使其在沿曲线移动时与斜率方向一致

□ 圆弧刷子

□ 采用填充的办法。

区域填充属性

□ 区域填充属性选择包括颜色、图案和透明度。

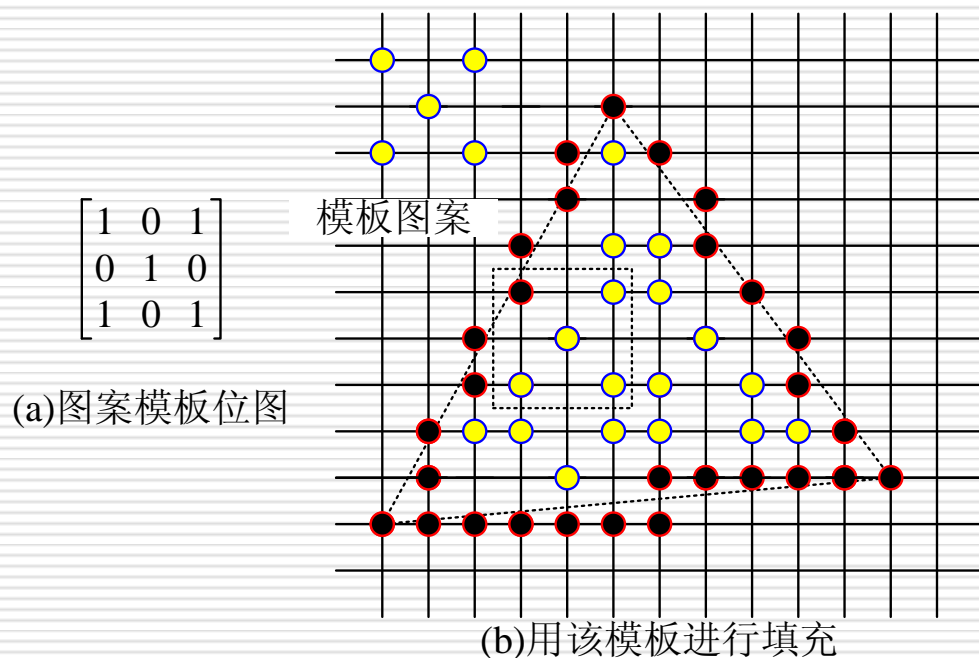


图3.30 利用图案模板进行三角形的填充

根据图案和透明度属性来填充平面区域的基本思想是：

- 首先用模板定义各种图案。
- 然后，修改填充的扫描转换算法：在确定了区域内一像素之后，不是马上往该像素填色而是先查询模板位图的对应位置。若是以透明方式填充图案，则当模板位图的对应位置为1时，用前景色写像素，否则，不改变该像素的值。若是以不透明方式填充图案，则视模板位图对应位置为1或0来决定是用前景色还是背景色去写像素。

区域填充属性

□ 确定区域与模板之间的位置关系（对齐方式）

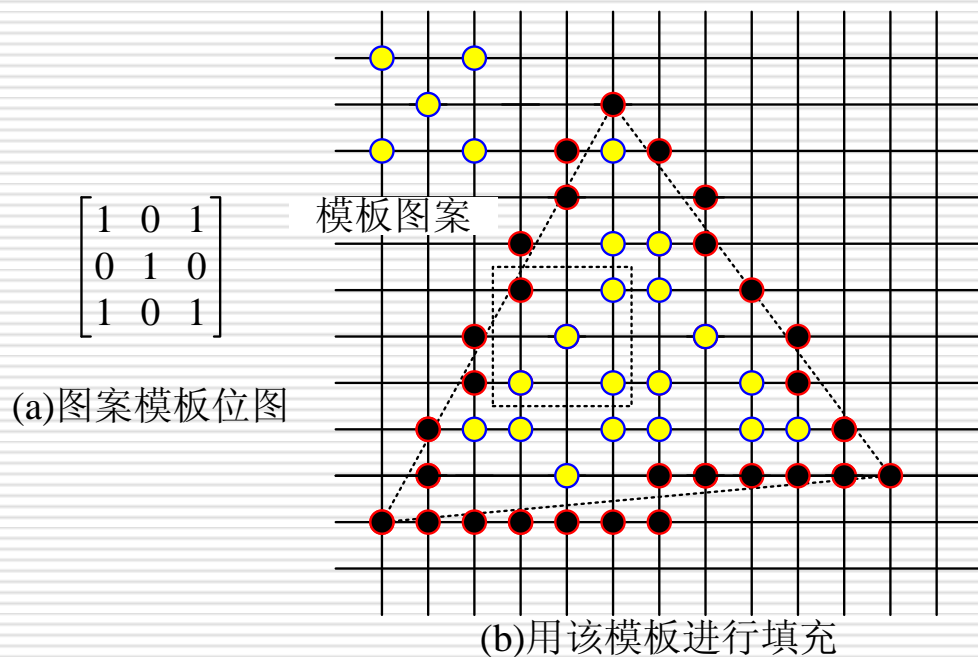


图3.31 利用图案模板进行三角形的填充

3.4 反走样

□ 用离散量表示连续量引起的失真，就叫做走样（Aliasing）。

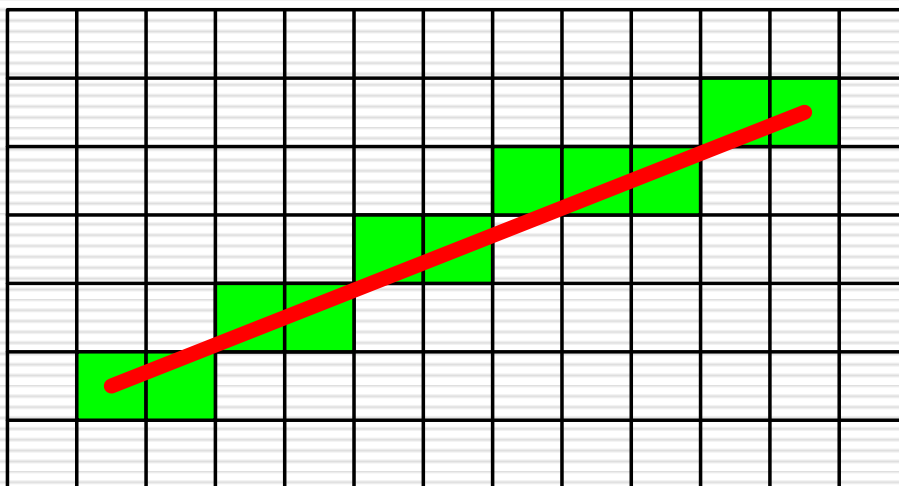


图3.32 绘制直线时的走样现象

反走样

产生原因:

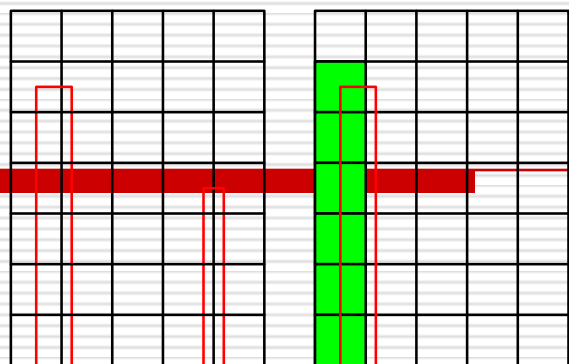
数学意义上的图形是由无限多个连续的、面积为零的点构成；但在光栅显示器上，用有限多个离散的，具有一定面积的像素来近似地表示他们。

反走样

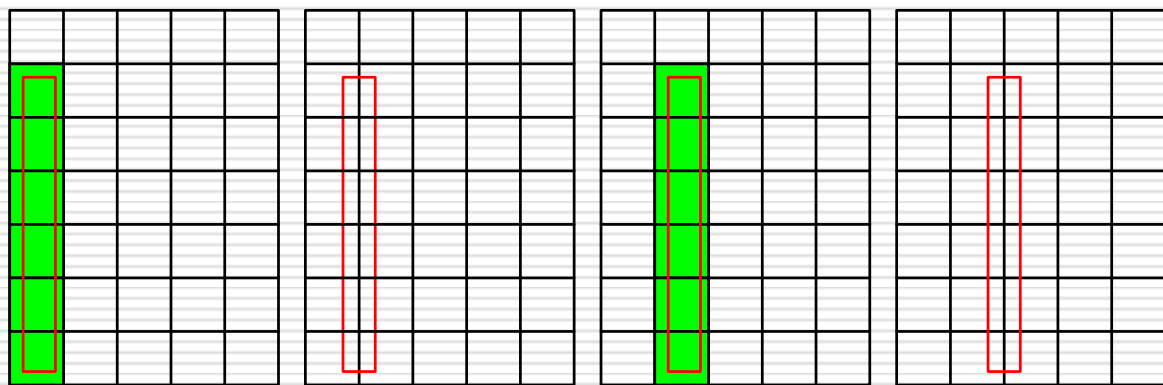
走样现象:

- 一是光栅图形产生的阶梯形。
- 一是图形中包含相对微小的物体时，这些物体在静态图形中容易被丢弃或忽略，在动画序列中时隐时现，产生闪烁。

例子



(a)需显示的矩形 (b)显示结果



(a)显示 (b)不显示 (c)显示 (d)不显示

图3.33 丢失细节与运动图形的闪烁

□ 用于减少或消除这种效果的技术，称为反走样（**antialiasing**，图形保真）。

□ 一种简单方法:

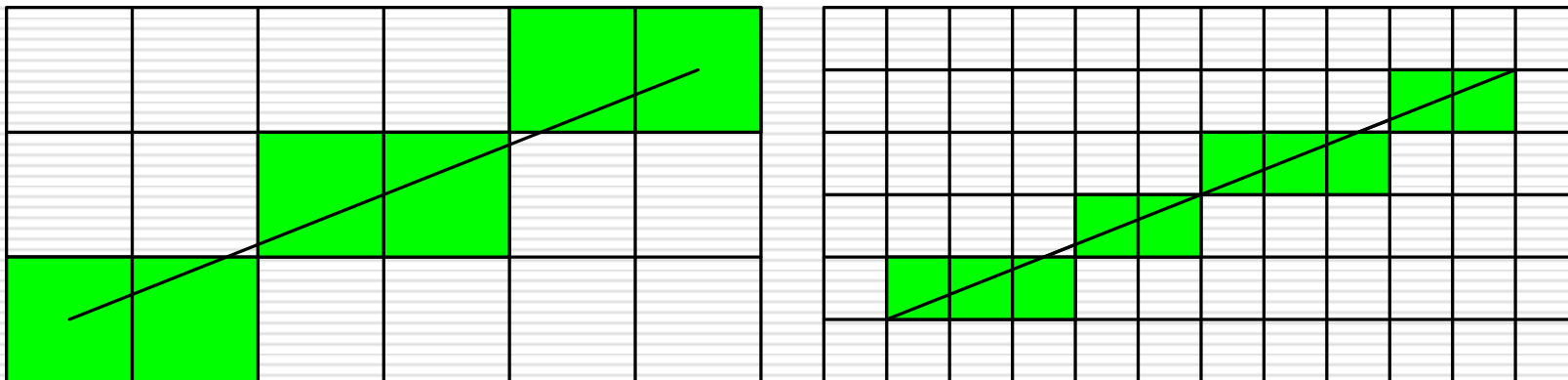


图3.34 分辨率提高一倍，阶梯程度减小一倍

□ 过取样（**supersampling**），或后滤波

□ 区域取样（**area sampling**），或前滤波

反走样——过取样 (super sampling)

- **过取样**: 在高于显示分辨率的较高分辨率下用点取样方法计算, 然后对几个象素的属性进行平均得到较低分辨率下的象素属性。

简单过取样

在x, y方向把分辨率都提高一倍, 使每个象素对应4个子象素, 然后扫描转换求得各子象素的颜色亮度, 再对4个象素的颜色亮度进行平均, 得到较低分辨率下的象素颜色亮度。

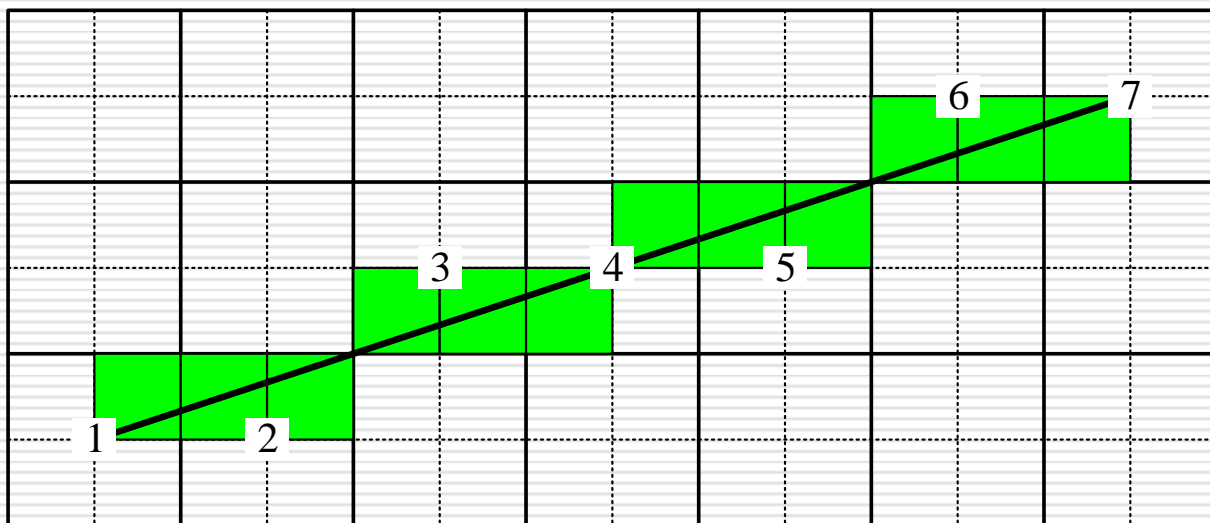
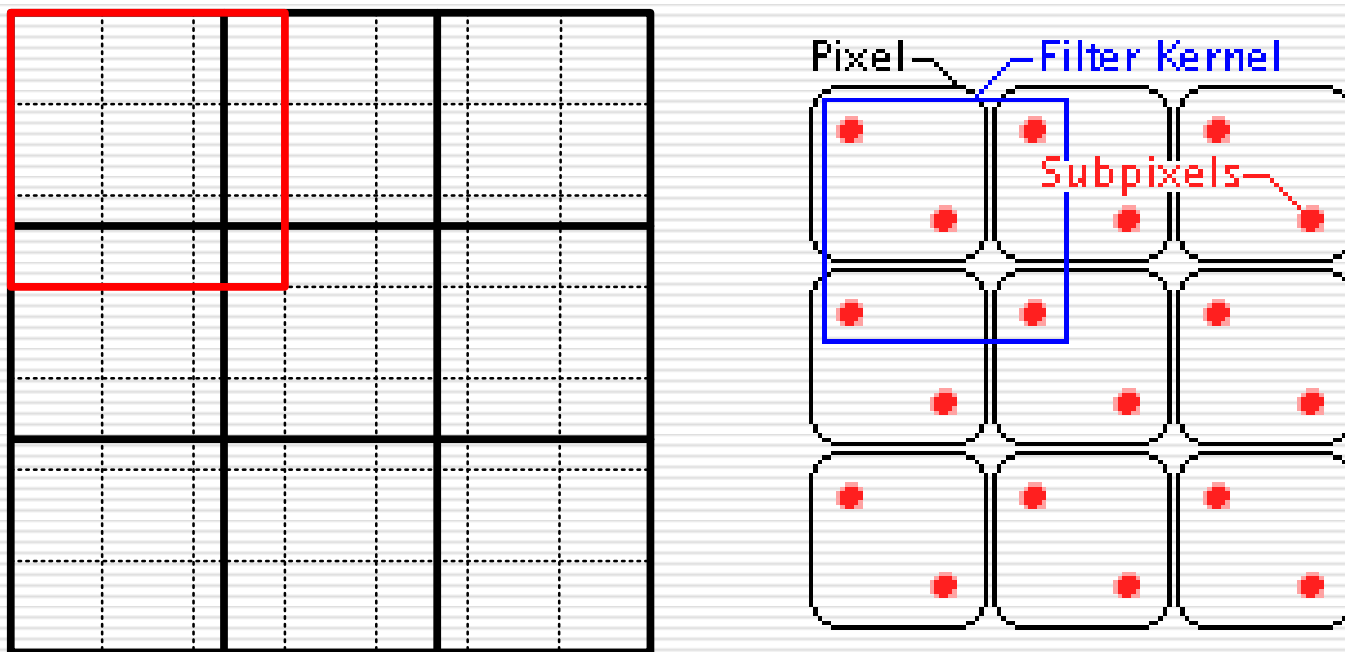


图3.35 简单的过取样方式

□ **重叠过取样**: 为了得到更好的效果, 在对一个像素点进行着色处理时, 不仅仅只对其本身的子像素进行采样, 同时对其周围的多个像素的子像素进行采样, 来计算该点的颜色属性。



□ 基于加权模板的过取样：前面在确定像素的亮度时，仅仅是对所有子像素的亮度进行简单的平均。更常见的做法是给接近像素中心的子像素赋予较大的权值，即对所有子像素的亮度进行加权平均。

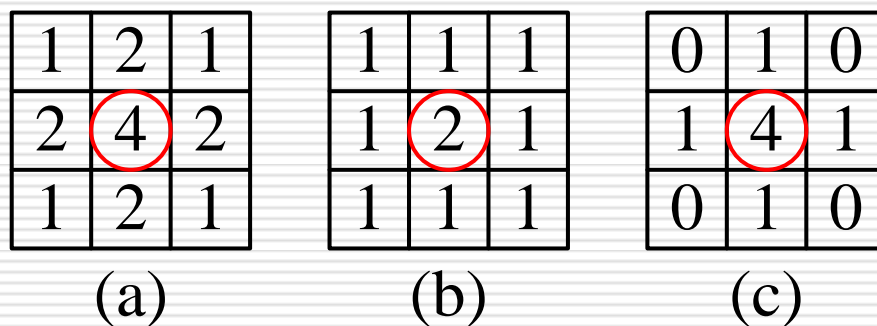


图3.37 常用的加权模板

反走样——简单的区域取样

- 在整个像素区域内进行采样，这种技术称为区域取样。又由于像素的亮度是作为一个整体被确定的，不需要划分子像素，故也被称为前置滤波。

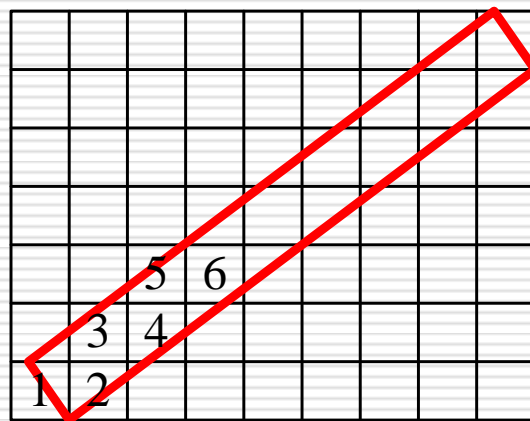


图3.38 有宽度的直线段

反走样——简单的区域取样

如何计算直线段与像素相交区域的面积？

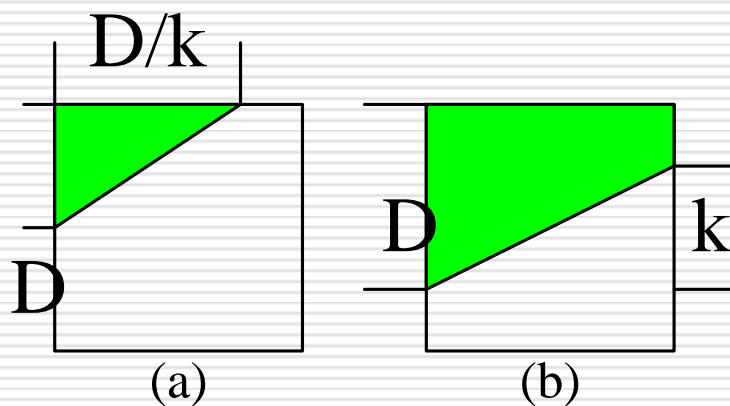


图3.39 重叠区域面积的计算

反走样——简单的区域取样

- 可以利用一种求相交区域的近似面积的离散计算方法：
 - (1)将屏幕像素分割成 n 个更小的子像素,
 - (2)计算中心落在直线段内的子像素的个数 m ,
 - (3) m/n 为线段与像素相交区域面积的近似值。
- 直线段对一个像素亮度的贡献与两者重叠区域的面积成正比。
- 相同面积的重叠区域对像素的贡献相同。

反走样——加权区域取样

- 过取样中，我们对所有子像素的亮度进行简单平均或加权平均来确定像素的亮度。
- 在区域取样中，我们使用覆盖像素的连续的加权函数（**Weighting Function**）或滤波函数（**Filtering Function**）来确定像素的亮度。

加权区域取样原理

加权函数 $W(x,y)$ 是定义在二维显示平面上的函数。对于位置为 (x,y) 的小区域 dA 来说，函数值 $W(x,y)$ （也称为在 (x,y) 处的高度）表示小区域 dA 的权值。将加权函数在整个二维显示图形上积分，得到具有一定体积的滤波器（**Filter**），该滤波器的体积为1。将加权函数在显示图形上进行积分，得到滤波器的一个子体，该子体的体积介于0到1之间。用它来表示像素的亮度。

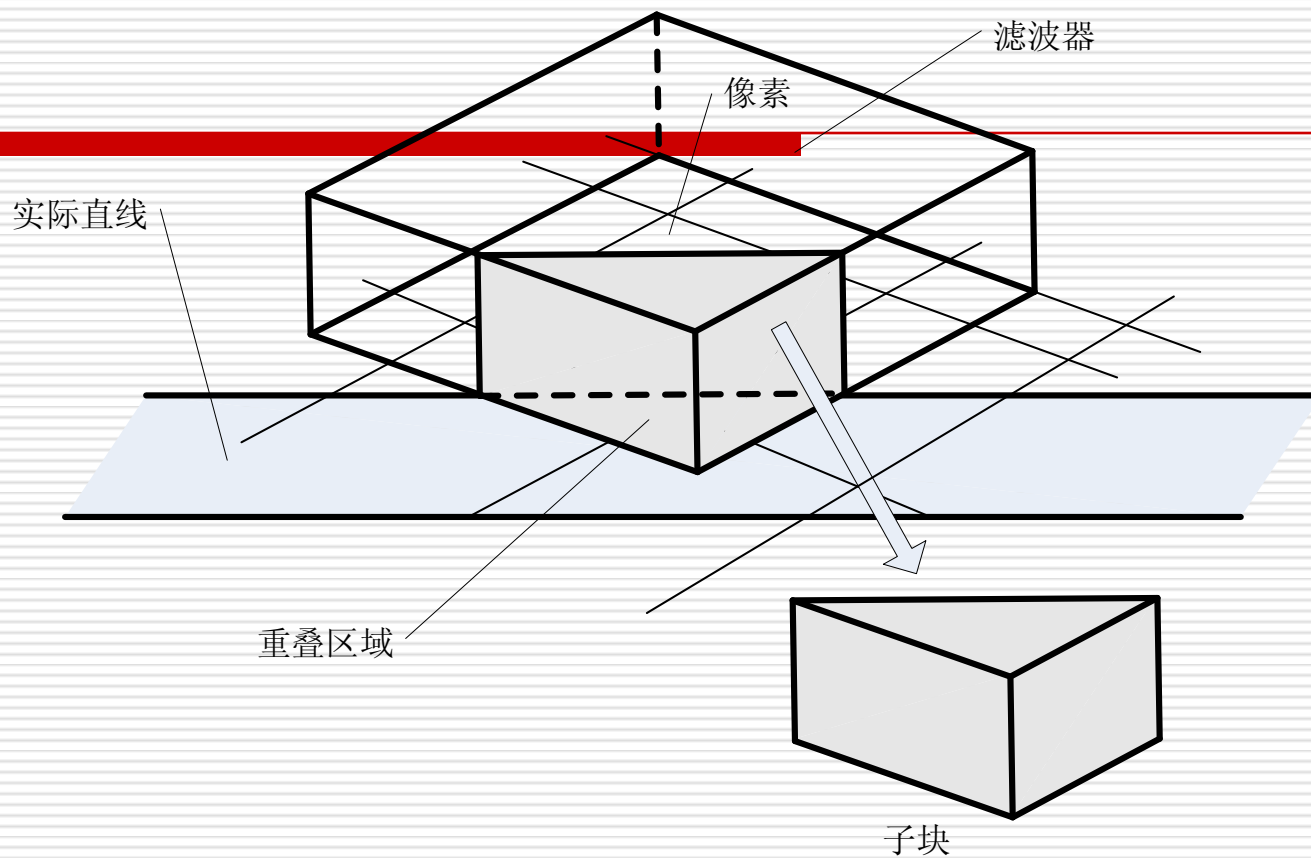


图3.40 盒式滤波器的加权区域取样

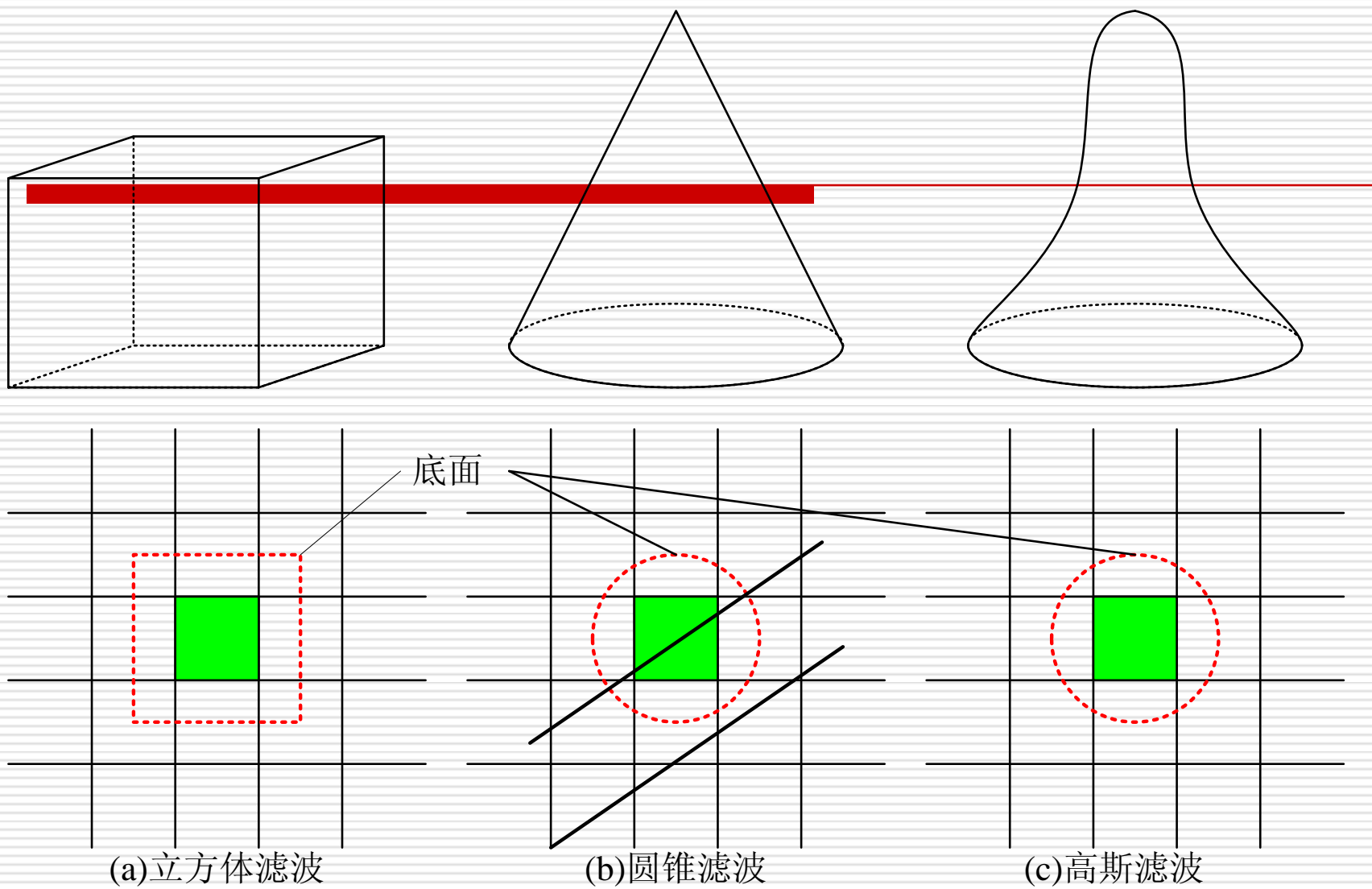


图3.41 常用的滤波函数

反走样——加权区域取样

特点:

- 接近理想直线的象素将被分配更多的灰度值;
- 相邻两个象素的滤波器相交, 有利于缩小直线条上相邻象素的灰度差。

3.5 在OpenGL中绘图

- 点的绘制
- 直线的绘制
- 多边形面的绘制
- **OpenGL中的字符函数**
- **OpenGL中的反走样**

点的绘制

□ 点的绘制

```
glBegin(GL_POINTS);  
    glVertex3f(0.0f, 0.0f, 0.0f);  
    glVertex3f(10.0f, 0.0f, 0.0f);  
glEnd();
```

□ 点的属性（大小）

```
void glPointSize(GLfloat size);
```

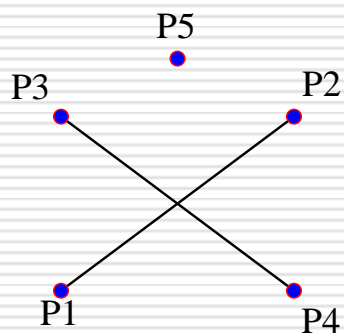
直线的绘制

□ 直线的绘制模式

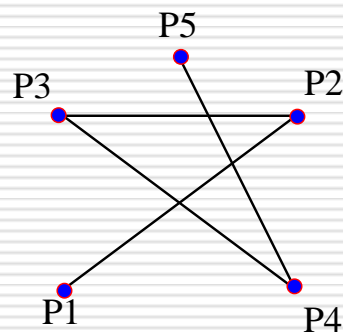
■ GL_LINES

■ GL_LINE_STRIP

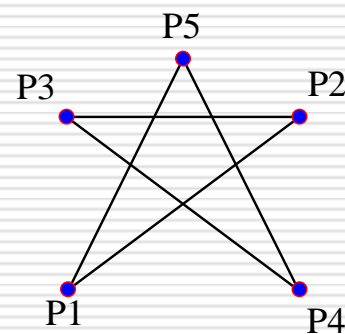
■ GL_LINE_LOOP



(a) GL_LINES画线模式



(b) GL_LINE_LOOP画线模式



(c) GL_LINE_STRIP画线模式

图3.42 OpenGL画线模式

直线的绘制

□ 直线的属性

■ 线宽

void glLineWidth(GLfloat width)

■ 线型

glEnable(GL_LINE_STIPPLE);
glLineStipple(GLint factor, GLushort pattern);

直线的绘制

模式：0X00FF = 255

二进制表示：0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

画线模式：



线：



图3.43 画线模式用于构造线段

多边形面的绘制

□ 三角形面的绘制

- **GL_TRIANGLES**

- **GL_TRIANGLE_STRIP**

- **GL_TRIANGLE_FAN**

□ 四边形面的绘制

- **GL_QUADS**

- **GL_QUADS_STRIP**

□ 多边形面的绘制 (**GL_POLYGON**)

多边形面的绘制

□ 多边形面的绘制规则

- 所有多边形都必须是平面的。

- 多边形的边缘决不能相交，而且多边形必须是凸的。

□ 解决：对于非凸多边形，可以把它分割成几个凸多边形（通常是三角形），再将它绘制出来。

多边形面的绘制

- 问题：轮廓图形状状态会看到组成大表面的所有小三角形。处理OpenGL提供了一个特殊标记来处理这些边缘，称为边缘标记。

glEdgeFlag (True)

glEdgeFlag (False)

多边形面的属性

□ 多边形面的正反属性（绕法）

指定顶点时顺序和方向的组合称为“绕法”。绕法是一切多边形图元的一个重要特性。一般默认情况下，OpenGL认为逆时针绕法的多边形是正对着的。

glFrontFace(GL_CW);

多边形面的属性

□ 多边形面的颜色

- **glShadeModel(GL_FLAT)** 用指定多边形最后一个顶点时的当前颜色作为填充多边形的纯色，唯一例外是**GL_POLYGON**图元，它采用的是第一个顶点的颜色。
- **glShadeModel(GL_SMOOTH)** 从各个顶点给三角形投上光滑的阴影，为各个顶点指定的颜色之间进行插值。

多边形面的属性

□ 多边形面的显示模式

```
glPolygonMode(GLenum face,  
GLenum mode);
```

- 参数`face`用于指定多边形的哪一个面受到模式改变的影响。
- 参数`mode`用于指定新的绘图模式。

多边形面的属性

□ 多边形面的填充

多边形面既可以用纯色填充，也可以用
 32×32 的模板位图来填充。

```
void glPolygonStipple(const GLubyte  
*mask);
```

```
glEnable(GL_POLYGON_STIPPLE);
```

多边形面的属性

□ 多边形面的法向量

- 法向量是垂直于面的方向上点的向量，它确定了几何对象在空间中的方向。
- 在OpenGL中，可以为每个顶点指定法向量。

```
void glNormal3{bsidf} ( TYPE nx, TYPE  
ny, TYPE nz);
```

```
void glNormal3{bsidf}v (const TYPE* v);
```

OpenGL中的字符函数

□ GLUT位图字符

```
void glutBitmapCharacter(void *font,  
int character);
```

□ GLUT矢量字符

```
void glutStrokeCharacter(void *font,  
int character);
```

OpenGL中的反走样

□ 启用反走样

`glEnable(primitiveType);`

□ 启用OpenGL颜色混和并指定颜色混合函数

`glEnable(GL_BLEND);`

`glBlendFunc(GL_SRC_ALPHA,
GL_ONE_MINUS_SRC_ALPHA);`

OpenGL中的反走样

- 颜色混和函数用于计算两个相互重叠的对象的颜色。
- 在RGBA颜色模式（A表示透明度）中，已知源像素的颜色值为 (S_r, S_g, S_b, S_a) ，目标像素的颜色值为 (D_r, D_g, D_b, D_a) ，颜色混合后像素的颜色为：

$$(R_S.S_r + R_D.D_r, G_S.S_g + G_D.D_g, B_S.S_b + B_D.D_b, A_S.S_a + A_D.D_a)$$

OpenGL中的反走样

□ 定义混合因子

```
void glBlendFunc(GLenum srcfactor,  
GLenum destfactor);
```

表5-1 源混和因子和目标混合因子

常量	RGB混合因子	Alpha混合因子
GL_ZERO	(0, 0, 0)	0
GL_ONE	(1, 1, 1)	1
GL_SRC_COLOR	(R_S, G_S, B_S)	A_S
GL_ONE_MINUS_SRC_COLOR	$(1, 1, 1) - (R_S, G_S, B_S)$	$1 - A_S$
GL_DST_COLOR	(R_D, G_D, B_D)	A_D
GL_ONE_MINUS_DST_COLOR	$(1, 1, 1) - (R_D, G_D, B_D)$	$1 - A_D$
GL_SRC_ALPHA	(A_S, A_S, A_S)	A_S
GL_ONE_MINUS_SRC_ALPHA	$(1, 1, 1) - (A_S, A_S, A_S)$	$1 - A_S$
GL_DST_ALPHA	(A_D, A_D, A_D)	A_D
GL_ONE_MINUS_DST_ALPHA	$(1, 1, 1) - (A_D, A_D, A_D)$	$1 - A_D$



直线光栅化算法

- DDA算法
- Bresenham算法

圆光栅化算法

- 中点算法
- 中点整数算法
- 中点整数优化算法





2.1 直线光栅化法

DDA 算法 (Digital Differential Analyzer)

- David F. Rogers 的描述 (适用于所有象限)
- James D. Foley 的描述 (只适用于第一象限, 且 $K < 1$)
- 本教程的描述 (适用于所有象限及任何端点)

Bresenham 算法

- 基本原理
- Bresenham 算法
- 整数 Bresenham 算法
- 一般整数 Bresenham 算法





2.1.1 DDA算法算法

1) David F. Rogers 描述描述

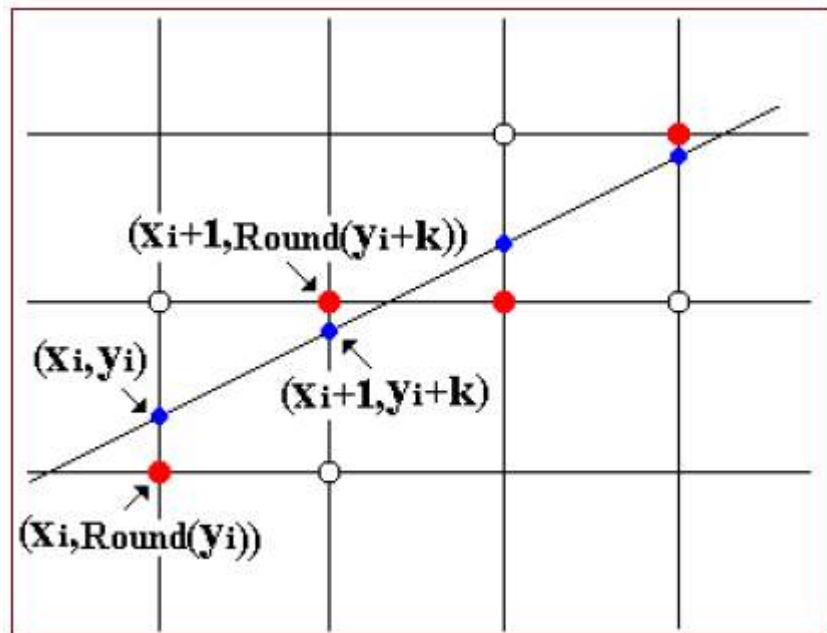
直线的基本微分方程是：

$$\frac{dy}{dx} = \text{常数 } (k)$$

设直线通过点 $P1(x1,y1)$ 和 $P2(x2,y2)$,

则直线方程可表示为：

$$\frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1} = k$$





1) David F. Rogers 描述描述

- ◆ 如果已知第 i 点的坐标，可用步长 StepX 和 StepY 得到

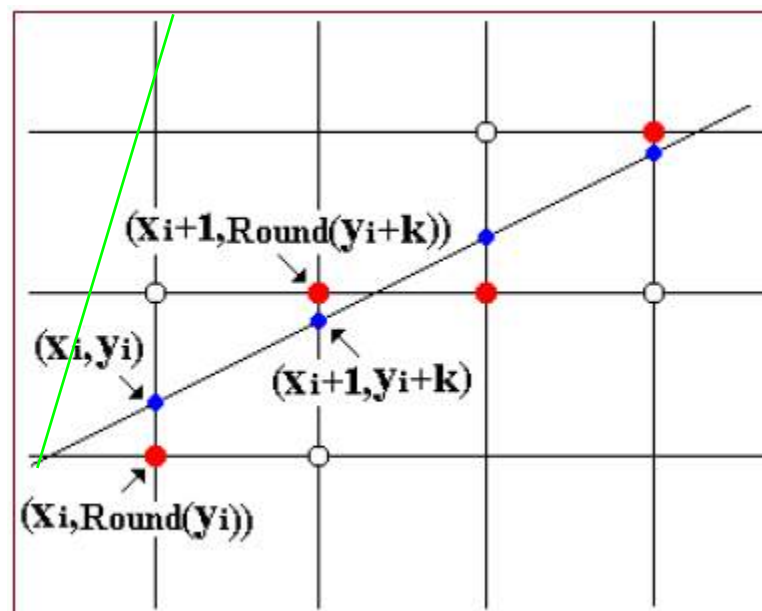
第 $i+1$ 点的坐标为：

- $x_{i+1} = x_i + \text{StepX}$
- $y_{i+1} = y_i + \text{StepY}$ 或 $y_{i+1} = y_i + k * \text{StepX}$

- ◆ 例图中

- $k < 1$
- $\text{StepX} = 1$
- $\text{StepY} = k$

- ◆ 将算得的直线上每个点的当前坐标，按四舍五入得到光栅点的位置





1) David F. Rogers 描述

// Digital Differential Analyzer (DDA) routine for rasterizing a line

// The line end points are (xs,ys) and (xe,ye) assumed not equal.

// Round is the function. Note: Many Round functions are floor functions, i.e Round (-8.5)=-9 rather than -8. The algorithm assumes this is the case.

// Approximate the line length

If $(|xs - xe| \geq |ys - ye|)$ then //插补长度

Length $\leftarrow |xs - xe|$;

else

Length $\leftarrow |ys - ye|$;

end if





1) David F. Rogers 描述描述

// Select the larger of Δx or Δy to be one raster unit.

StepX = ($x_e - x_s$) / Length;

StepY = ($y_e - y_s$) / Length;

x = x_s ; //首点

y = y_s ;

i = 1; // Begin main loop

while (i ≤ Length)

 WritePixel (Round(x), Round(y) ,value));

 x = x + StepX;

 y = y + StepY;

 i++;

end while





2) James D.Foley 描述描述

□ 令
$$k = \frac{y_2 - y_1}{x_2 - x_1}$$

有：

$$y_{i+1} = y_i + k * \text{StepX}$$

□ 若 $0 < k < 1$ ，即 $\Delta x > \Delta y$

□ 因光栅单位为 1，

□ 可以采用每次 x 方向增加 1，

□ 而 y 方向增加 k 的办法得到下一个直线点。





2) James D.Foley 描述描述

```
void Line ( //设  $0 \leq k \leq 1, x_s < x_e$ 
```

```
    int xs,ys; //左端点
```

```
    int xe,ye; //右端点
```

```
    int value) //赋给线上的象数值
```

```
{
```

```
    int x; //x以步长为单位从 xs增长到 xe
```

```
    double dx =xe-xs;
```

```
    double dy =ye-ys;
```

```
    double k =dy/dx; // 直线之斜率 k
```

```
    double y =ys;
```

```
    for (x=xs; x<=xe; x++) {
```

```
        WritePixel(x,Round(y),value); //置象数值为 value
```

```
        y+=k; // y移动步长是斜率 k
```

```
    } // End of for
```

```
} // Line
```





3) 已有算法描述分析

Rogers 描述 :

- ◆ 采用 $x = x + \text{StepX}$, $y = y + \text{StepY}$,
- ◆ 逼近点并不是直线的一个最好的逼近 ;

D.Foley描述 : 可能引起积累误差

- ◆ 未分析直线端点不在象素点上的情况 ;
- ◆ 只给出 $0 - 45^\circ$ 第一个八卦限的描述 。

为避免引起积累误差 , **D.Foley**描述中采用

- ◆ `double dx =xe-xs;`
- ◆ `double dy =ye-ys;`
- ◆ `double k =dy/dx; // 直线之斜率 k`





4)本教程描述——任意方向直线插补算法

```
void DDALine (  
    float xs, ys; //起点  
    float xe, ye; //终点  
    int value) //赋给线上的象数值  
{  
    int n, ix, iy, idx, idy ;  
    int Flag; //插补方向标记  
    int Length; //插补长度  
    float x, y, dx, dy;
```





第 2 章 基本图形生成算法 (I)

```
dx=xe-xs;          dy=ye-ys;
if (fabs(dy)<fabs(dx)) { //X方向长 , 斜率 <=1
    Length=abs(Round(xe)-Round(xs));
    Flag=1; //最大的插补长度和方向标记
    ix= Round(xs); //初始 X点
    idx=isign(dx); //X方向单位增量
    y= ys+dy/dx*((float)(ix)-xs); //初始 Y点修正
    dy=dy/fabs(dx); //Y方向斜率增量
}
else { // Y方向长 , 斜率 >1
    Length=abs(Round(ye)-Round(ys));
    Flag=0;
    iy= Round(ys); //初始 Y点
    idy=isign(dy); //Y方向单位增量
    x= xs+dx/dy*((float)(iy)-ys); //初始 X点修正
    dx=dx/fabs(dy); //X方向斜率增量
}
```





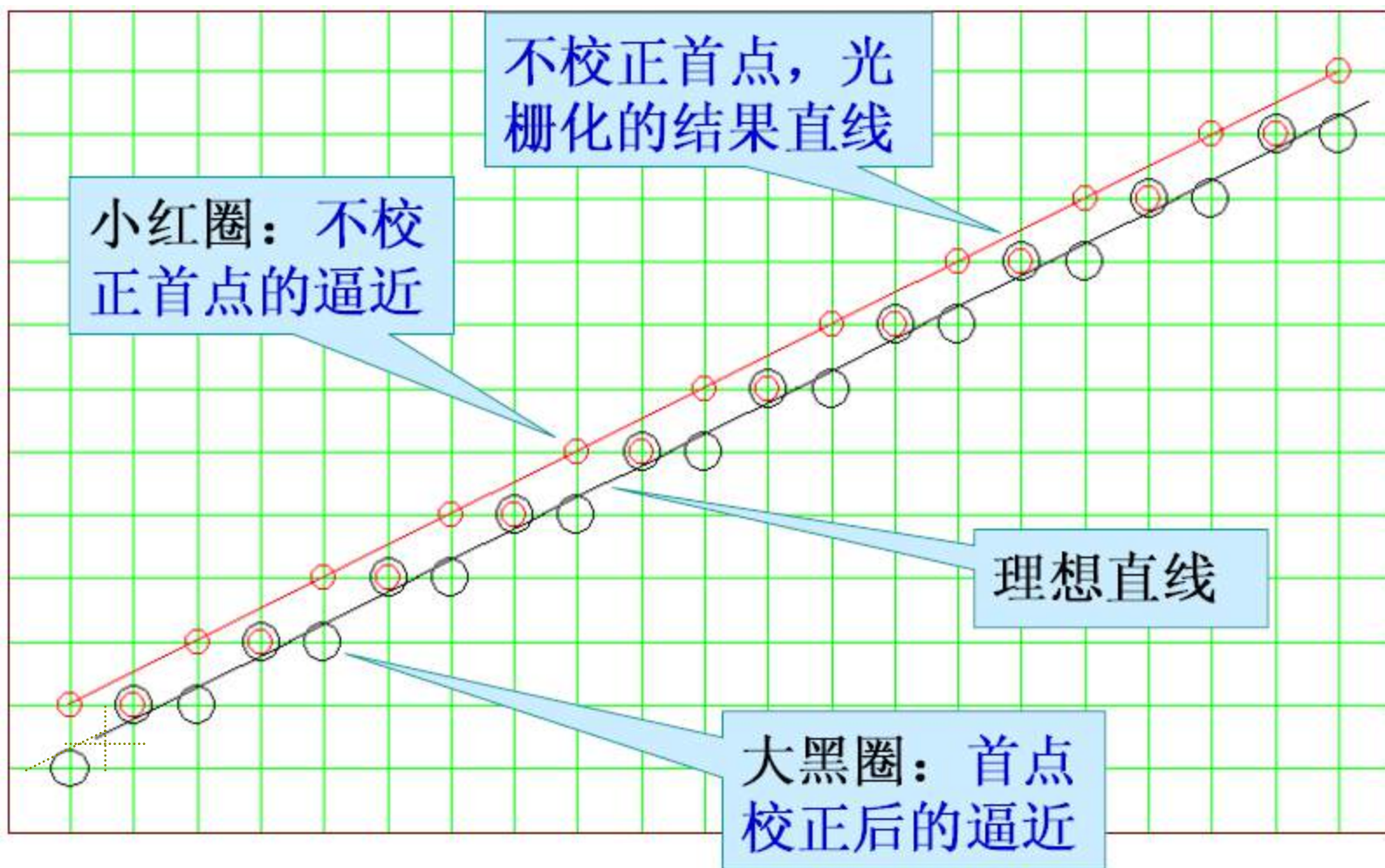
第 2 章 基本图形生成算法 (I)

```
if (Flag) { //X方向单位增量
    for (n=0; n<= Length; n++) { //X方向插补过程
        WritePixel(ix, Round(y), value);
        ix+=idx;
        y+=dy;
    } //End of for
} //End of if
else { //Y方向斜率增量
    for (n=0; n<= Length; n++) { //Y方向插补过程
        WritePixel (Round(x), iy, value);
        iy+=idy;
        x+=dx;
    } //End of for
} //End of else
} //Finish
```





5) 本教程描述 —— 首点校正对逼近的影响





2.1.2 Bresenham算法

- Bresenham算法是计算机图形学典型的直线光栅化算法 。
- 从另一个角度看直线光栅化显示算法的原理 ：

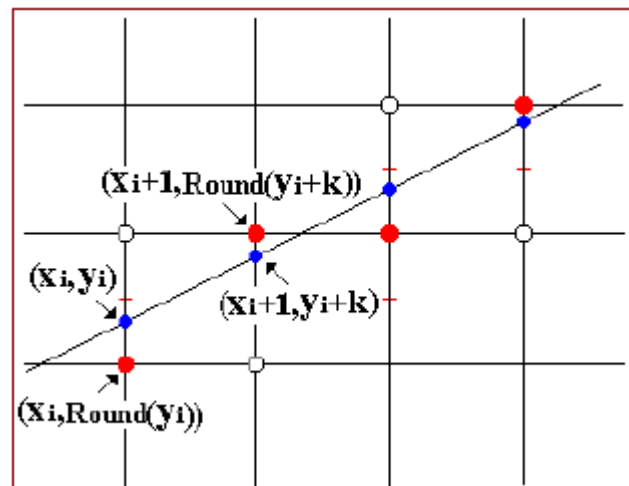
由直线的斜率确定选择在 x 方向或 y 方向上每次递增（减）1个单位，另一变量的递增（减）量为 0 或 1，它取决于实际直线与最近光栅网格点的距离，这个距离的最大误差为 0.5。





1) Bresenham的基本原理

- 假定直线斜率 k 在 $0 \sim 1$ 之间。此时，只需考虑 x 方向每次递增 1 个单位，决定 y 方向每次递增 0 或 1。
- 设直线的
当前点为 (x_i, y_i)
当前光栅点为 (x_i, y_i)
- 下一个
直线的点应为 $(x_{i+1}, y_i + k)$
直线的光栅点
 - 或为右光栅点 (x_{i+1}, y_i) (y 方向递增量 0)
 - 或为右上光栅点 $(x_{i+1}, y_i + 1)$ (y 方向递增量 1)





第 2 章 基本图形生成算法 (I)

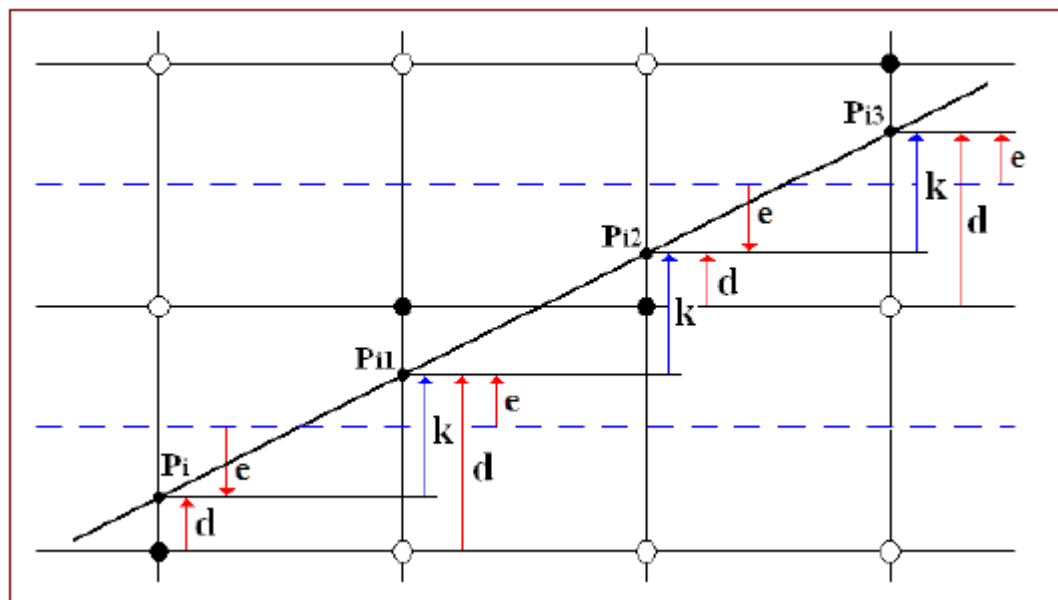
□ 记直线与它垂直方向最近的下光栅点的误差为 d ,

有： $d = (y + k) - y_i$ ，且

□ $0 \leq d \leq 1$

□ 当 $d < 0.5$ ：下一个象素应取右光栅点 (x_{i+1}, y_i)

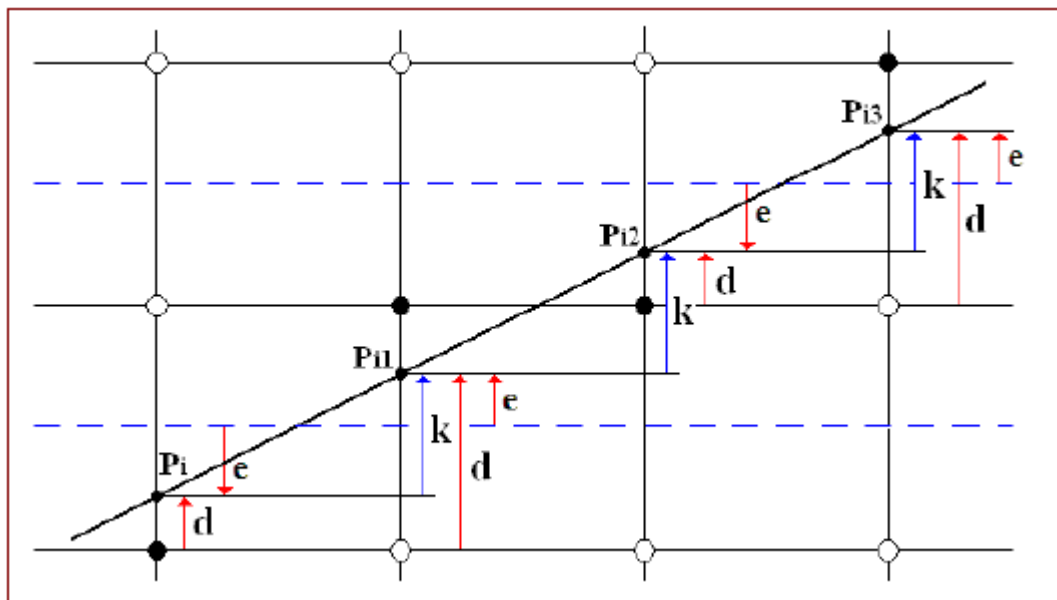
□ 当 $d \geq 0.5$ ：下一个象素应取右上光栅点 (x_{i+1}, y_{i+1})





1) Bresenham的基本原理

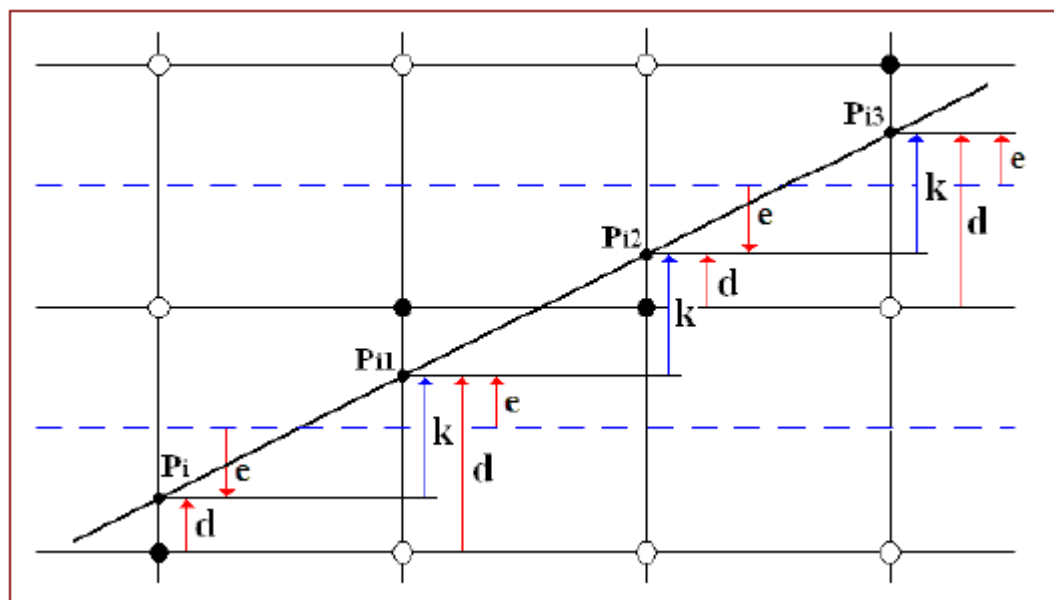
- 如果直线的（起）端点在整数点上，误差项 d 的初值： $d_0 = 0$
- x 坐标每增加 1， d 的值相应递增直线的斜率值 k ，
即： $d = d + k$
- 一旦 $d \geq 1$ ，就把它减去 1，保证 d 的相对性，且在 0-1 之间。





第 2 章 基本图形生成算法 (I)

- 令 $e=d-0.5$ ，关于 d 的判别式和初值可简化成：
 - e 的初值 $e_0 = -0.5$ ，增量亦为 k ;
 - $e < 0$ 时，取当前像素 (x_i, y_i) 的右方像素 (x_{i+1}, y_i) ;
 - $e > 0$ 时，取当前像素 (x_i, y_i) 的右上方像素 (x_{i+1}, y_{i+1}) ;
 - $e = 0$ 时，可任取上、下光栅点显示。





1) Bresenham的基本原理

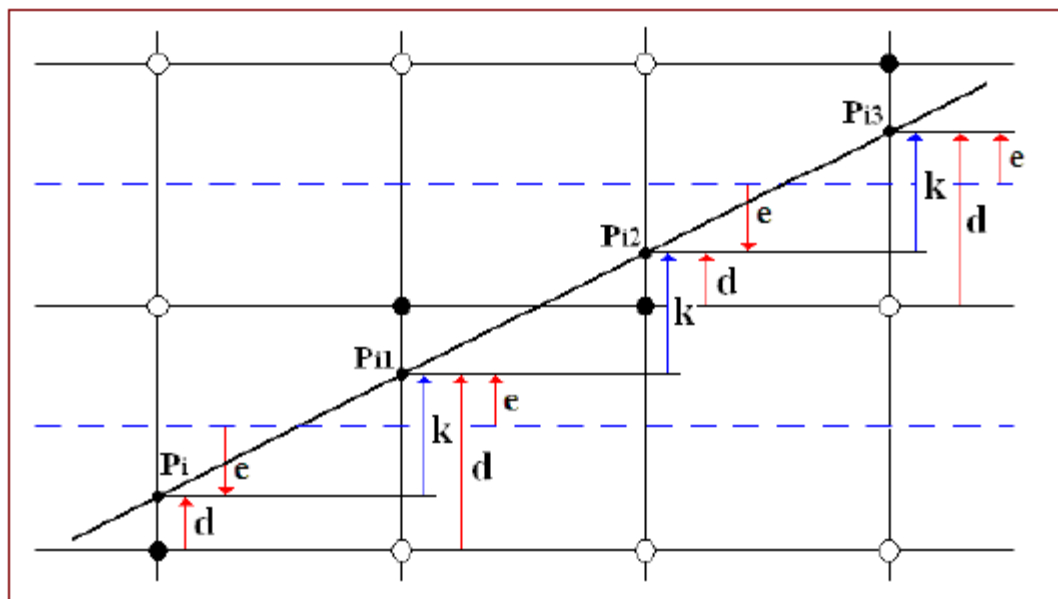
- Bresenham算法的构思巧妙：它引入动态误差 e ，当 x 方向每次递增 1 个单位，可根据 e 的符号决定 y 方向每次递增 0 或 1。
 - $e < 0$ ， y 方向不递增
 - $e > 0$ ， y 方向递增 1
 - x 方向每次递增 1 个单位， $e = e + k$





1) Bresenham的基本原理

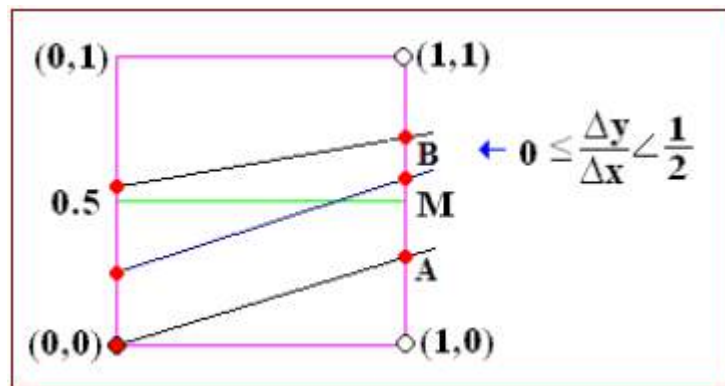
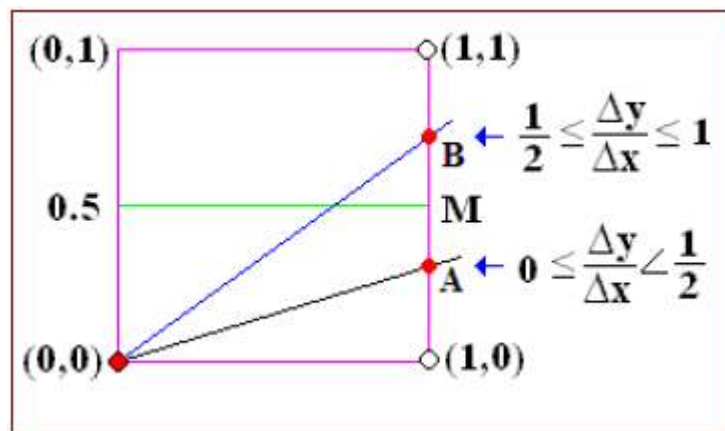
□ 因为 e 是相对量，所以当 $e > 0$ 时，表明 e 的计值将进入下一个参考点（上升一个光栅点），此时须： $e = e - 1$





2) Bresenham算法的实施——Rogers 版

- 通过 (0,0) 的所求直线的斜率大于 0.5，它与 $x=1$ 直线的交点离 $y=1$ 直线较近，离 $y=0$ 直线较远，因此取光栅点 (1,1) 比 (1,0) 更逼近直线；
- 如果斜率小于 0.5，则反之；
- 当斜率等于 0.5，没有确定的选择标准，但本算法选择 (1,1)。





2) Bresenham算法的实施 ——Rogers 版

//Bresenham's line rasterization algorithm for the first octal.

//The line end points are (xs,ys) and (xe,ye) assumed not equal.

// Round is the integer function.

// x,y, Δx , Δy are the integer, Error is the real.

//initialize variables

x=xs

y=ys

$\Delta x = x_e - x_s$

$\Delta y = y_e - y_s$

//initialize e to compensate for a nonzero intercept

Error = $\Delta y / \Delta x - 0.5$





2) Bresenham 算法的实施 —— Rogers 版

//begin the main loop

for i=1 to Δx

WritePixel (x, y, value)

if (Error ≥ 0) then

y=y+1

Error = Error -1 提问学生 why ?

end if

x=x+1

Error = Error + $\Delta y / \Delta x$

next i

finish





3) 整数 Bresenham 算法

- 上述 Bresenham 算法在计算直线斜率和误差项时要用到浮点运算和除法，采用整数算术运算和避免除法可以加快算法的速度。
- 由于上述 Bresenham 算法中只用到误差项（初值 $\text{Error} = \Delta y / \Delta x - 0.5$ ）的符号
- 因此只需作如下的简单变换：
$$\text{NError} = 2 * \text{Error} * \Delta x$$
- 即可得到整数算法，这使本算法便于硬件（固件）实现





3)整数 Bresenham算法

//Bresenham's integer line rasterization algorithm
for the first octal.

//The line end points are (xs,ys) and (xe,ye) assumed
not equal. All variables are assumed integer.

//initialize variables

$x = x_s$

$y = y_s$

$\Delta x = x_e - x_s$

$\Delta y = y_e - y_s$

//initialize e to compensate for a nonzero intercept

$NError = 2 * \Delta y - \Delta x$ (**Error** = $\Delta y / \Delta x - 0.5$)





3) 整数 Bresenham 算法

//begin the main loop

for i=1 to Δx

WritePixel (x, y)

if (NError ≥ 0) then

y=y+1

NError = NError - 2* Δx (Error = Error -1)

end if

x=x+1

NError = NError + 2* Δy (Error = Error + $\Delta y / \Delta x$)

next i

finish





4)一般 Bresenham算法算法

- 要使第一个八卦的 Bresenham算法适用于一般直线，只需对以下 2点作出改造：
 - 当直线的斜率 $|k|>1$ 时，改成 y 的增量总是 1，再用 Bresenham误差判别式确定 x 变量是否需要增加 1；
 - x 或 y 的增量可能是 “+1”或 “-1”，视直线所在的象限决定。





第 2 章 基本图形生成算法 (I)

//Bresenham's integer line rasterization algorithm for all quadrants

//The line end points are (xs,ys) and (xe,ye) assumed not equal. All variables are assumed integer.

//initialize variables

x=xs

y=ys

$\Delta x = \text{abs}(xe - xs)$

$$\Delta x = xe - xs$$

$\Delta y = \text{abs}(ye - ys)$

$$\Delta y = ye - ys$$

sx = isign(xe - xs)

sy = isign(ye - ys)

//Swap Δx and Δy depending on the slope of the line.

if $\Delta y > \Delta x$ then

 Swap($\Delta x, \Delta y$)

 Flag=1

else

 Flag=0

end if





第 2 章 基本图形生成算法 (I)

//initialize the error term to compensate for a nonzero

intercept

$NError = 2 * \Delta y - \Delta x$

//begin the main loop

for $i=1$ to Δx

WritePixel(x, y , value)

if ($NError \geq 0$) then

if (Flag) then // $\Delta y > \Delta x$

$x = x + sx$

else

$Y = Y + 1$

$y = y + sy$

end if // End of Flag

$NError = NError - 2 * \Delta x$

end if // End of NError





4)一般 Bresenham算法算法

if (Flag) then $\Delta y > \Delta x$

$y = y + sy$

else

$X = X + 1$

$x = x + sx$

end if

$NError = NError + 2 * \Delta y$

next i

finish





2.2 圆光栅化算法

2.2.1 利用圆的八方对称性画圆

□ 对圆的分析均假定圆心在坐标原点， 因为即使圆心不在原点， 可以通过一个简单的平移即可， 而对原理的叙述却方便了许多

□ 即考虑圆的方程为： $x^2+y^2=R^2$

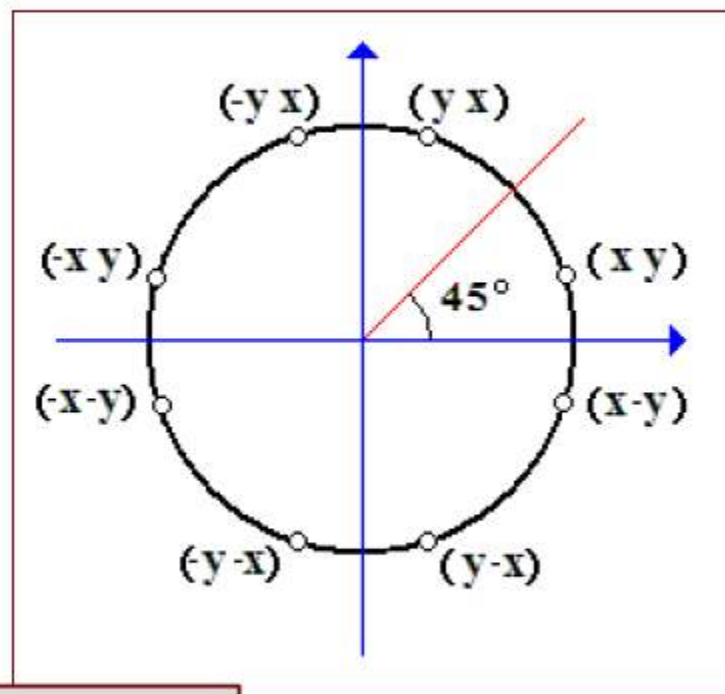




2.2.1 利用圆的八方对称性画圆

***void* CirclePoints (*int* x,*int* y, *int* value)**

```
{  
    WritePixel (x, y, value);  
    WritePixel (-x, y, value);  
    WritePixel (-x, -y, value);  
    WritePixel (x, -y, value);  
    WritePixel (y, x, value);  
    WritePixel (-y,x, value);  
    WritePixel (-y, -x, value);  
    WritePixel (y, -x, value);  
}
```



显然，当 $x=0$ 或 $x=y$ 或 $y=0$ 时，圆上的对称点只有4个，因此，CirclePoints()需要修正。



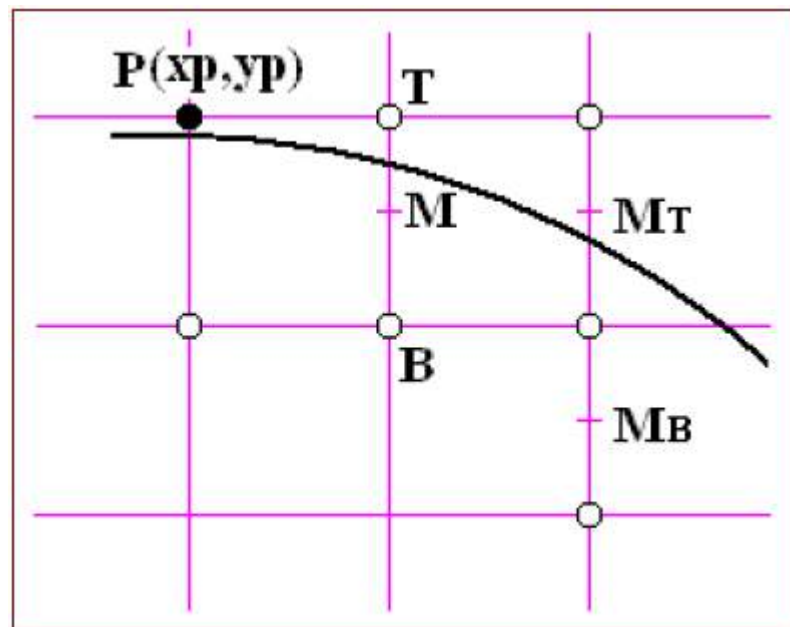


2.2.2 中点圆算法 —— 原理

□ 设 d 是点 $p(x,y)$ 到圆心的距离，有：

$$d = F(x,y) = x^2 + y^2 - R^2$$

□ 按照 Bresenham 算法符号变量的思想，以圆的下 2 个可选像素中点的函数值 d 的符号 决定选择 2 个可选像素 T 和 B 中哪一个更接近圆而作为圆的显示点？

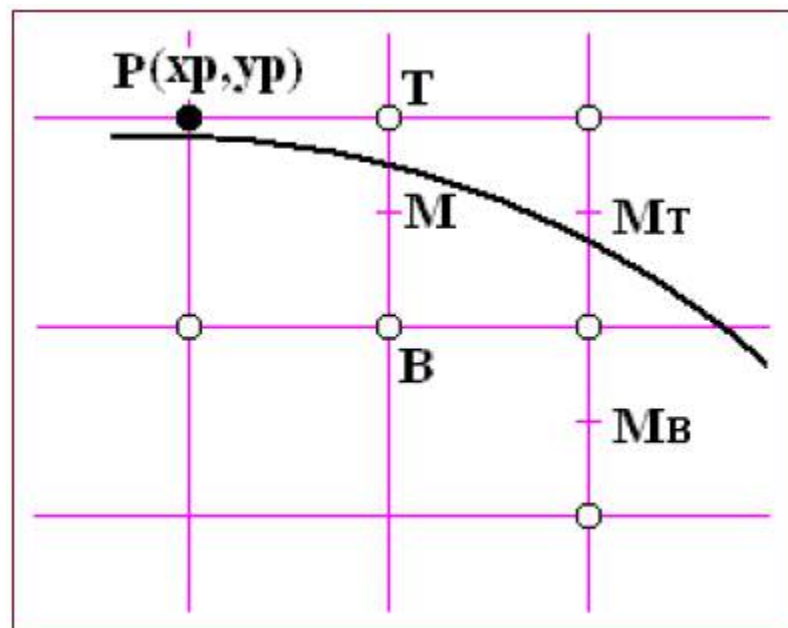


□ $d_M = F(x_M, y_M) = F(x_p + 1, y_p - 0.5) = (x_p + 1)^2 + (y_p - 0.5)^2 - R^2$

□ $d_{MT} = F(x_{MT}, y_{MT}) = F(x_p + 2, y_p - 0.5) = (x_p + 2)^2 + (y_p - 0.5)^2 - R^2$

□ $\Delta d_{MT} = d_{MT} - d_M = 2x_p + 3$

注意: $x_p^2+y_p^2-R^2$ 并不等于零





第 2 章 基本图形生成算法 (I)

如果 $d_M > 0$ ，表示下一中点 M 在圆外，用 B 点逼近，得

□ $d_{MB} = F(x_{MB}, y_{MB}) = F(x_p + 2, y_p - 1.5) = (x_p + 2)^2 + (y_p - 1.5)^2 - R^2$

□ $\Delta d_{MB} = d_{MB} - d_M = 2x_p - 2y_p + 5$

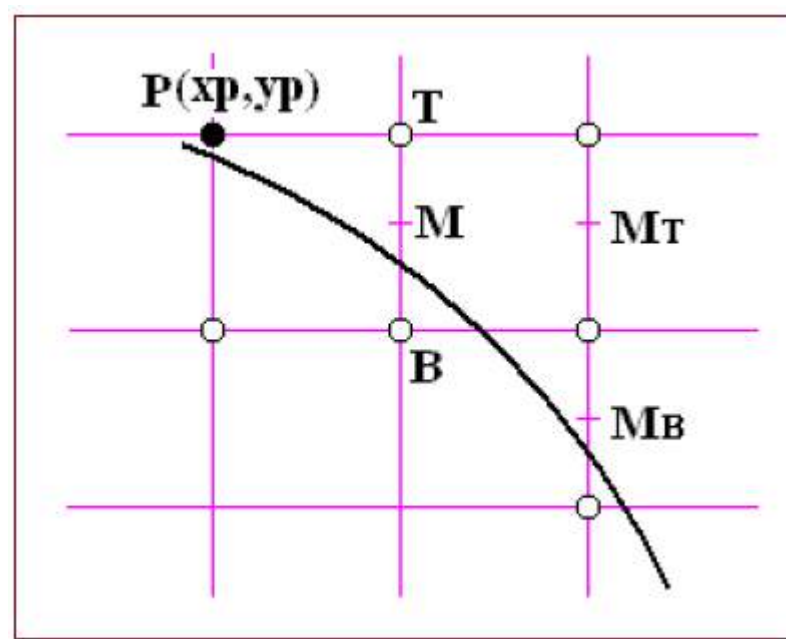
□ 结论：

□ 根据中点 d 的值，决定

□ 显示的光栅点（ T 或 B ）

□ 新的 Δd （ Δd_{MT} 或 Δd_{MB} ）

□ 更新 d

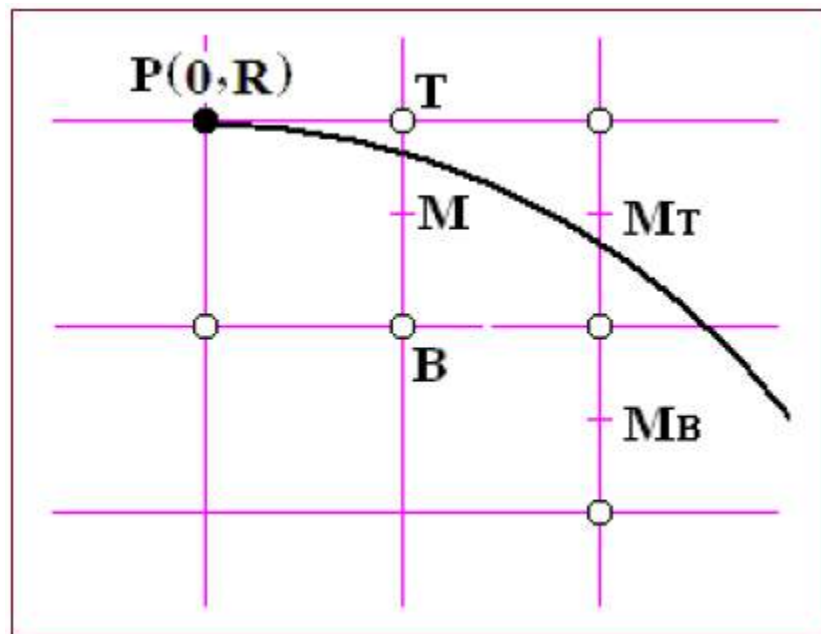




2.2.2 中点圆算法 —— 原理

初值

- 由 $x_0=0$, $y_0=R$
- 得 $x_{M0}=0+1$, $y_{M0}=R-0.5$
- $d_{M0}=F(x_{M0}, y_{M0})=F(1, R-0.5)=1^2+(R-0.5)^2-R^2=1.25-R$





2.2.3 中点圆算法 —— 实施

//中点圆算法 (假设圆的中心在原点)

```
void MidPointCircle(int radius,int value)
```

```
{
```

```
    int x=0;
```

```
    int y= radius;
```

```
    double d=1.25- radius;
```





2.2.3 中点圆算法 —— 实施

```
While (y>x) {  
    CirclePoints(x, y,value);  
    if (d<0) //选择 T  
        d+=2.0*x+3.0;  
    else { //选择 B  
        d+=2.0*(x-y)+5.0;  
        y--;  
    }  
    x++;  
} //End of while  
}
```





2.2.4 中点圆整数算法 —— 原理

- 中点圆算法 的半径是整数，而用于该算法
符号判别的变量 d （初值 $d=1.25-\text{radius}$ ）
采用浮点运算，会花费较多的时间。
- 为了将其改造成整数计算，定义新变量：
 $D = d - 0.25$
- 那么判别式 $d < 0$ 等价于 $D < -0.25$ 。
- 在 D 为整数情况下， $D < -0.25$ 和 $D < 0$ 等价
- 仍将 D 写成 d （新的初值 $d=1-\text{radius}$ ），可
得到 中点圆整数算法。





2.2.4 中点圆整数算法 —— 实施

//中点圆算法 (假设圆的中心在原点)

```
void MidPointCircle(int radius,int value)
```

```
{
```

```
    int x=0;
```

```
    int y= radius;
```

```
    int d=1- radius;    d=1.25-
```

```
    //CirclePoints(x, y, value);    radius
```





2.2.4 中点圆整数算法 —— 实施

While (y>x) {

 CirclePoints(x, y,value);

 if (d<0) //选择 T

 d+=2*x+3;

d+=2.0*x+3.0

 else { //选择 B

 d+=2*(x-y)+5;

d+=2.0*(x-y)+5.0

 y--;

 } //End of else

 x++;

 //CirclePoints(x, y,value);

} //End of while

}





2.2.5 中点圆整数优化算法 — 原理

□ 用 Δd 修正 d

□ 1) 选择 T 点 ($x_p \leftarrow x_p + 1$) :

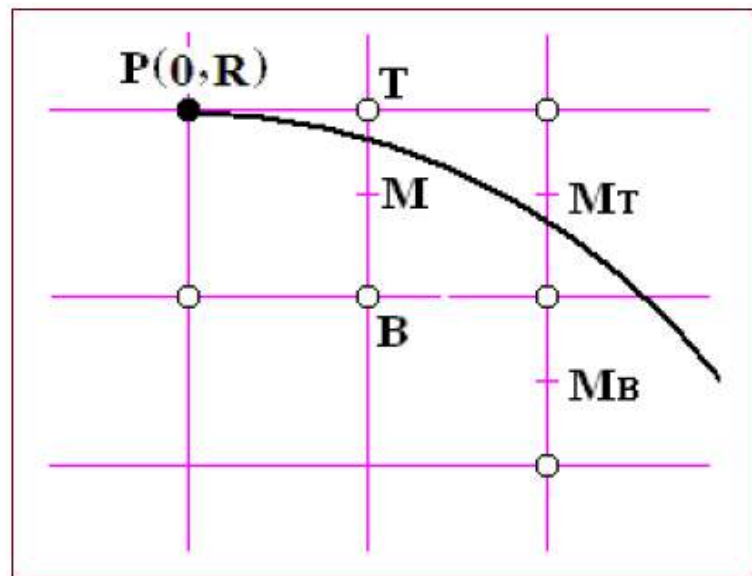
□ d 的增量 (一次差分) :

$$\triangleright \Delta d_T = 2x_p + 3$$

□ Δd 的增量 (二次差分) :

$$\triangleright \Delta^2 d_T = 2(x_p + 1) + 3 - (2x_p + 3) = 2$$

$$\triangleright \Delta^2 d_B = 2(x_p + 1) - 2y_p + 5 - (2x_p - 2y_p + 5) = 2$$





2.2.5 中点圆整数优化算法 — 原理

□ 2) 选择 B 点 ($x_p \leftarrow x_p + 1$, $y_p \leftarrow y_p - 1$) :

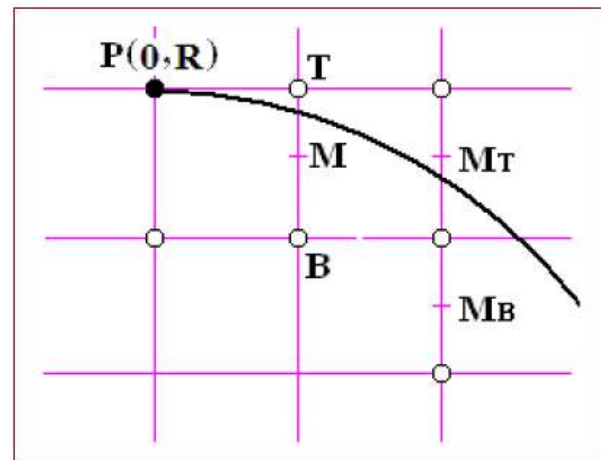
□ d 的增量 (一次差分) :

$$\triangleright \Delta d_B = 2x_p - 2y_p + 5$$

□ Δd 的增量 (二次差分) :

$$\triangleright \Delta^2 d_T = 2(x_p + 1) + 3 - (2x_p + 3) = 2$$

$$\triangleright \Delta^2 d_B = 2(x_p + 1) - 2(y_p - 1) + 5 - (2x_p - 2y_p + 5) = 4$$





2.2.5 中点圆整数优化算法 — 实施

//中点圆整数优化算法 （ 假设圆的中心在原点 ）

```
void MidPointCircleInt(int radius,int value)
```

```
{
```

```
    int x=0;
```

```
    int y= radius;
```

```
    int d=1- radius;
```

```
    int dt=3;
```

```
    int db= -2*radius+5;
```





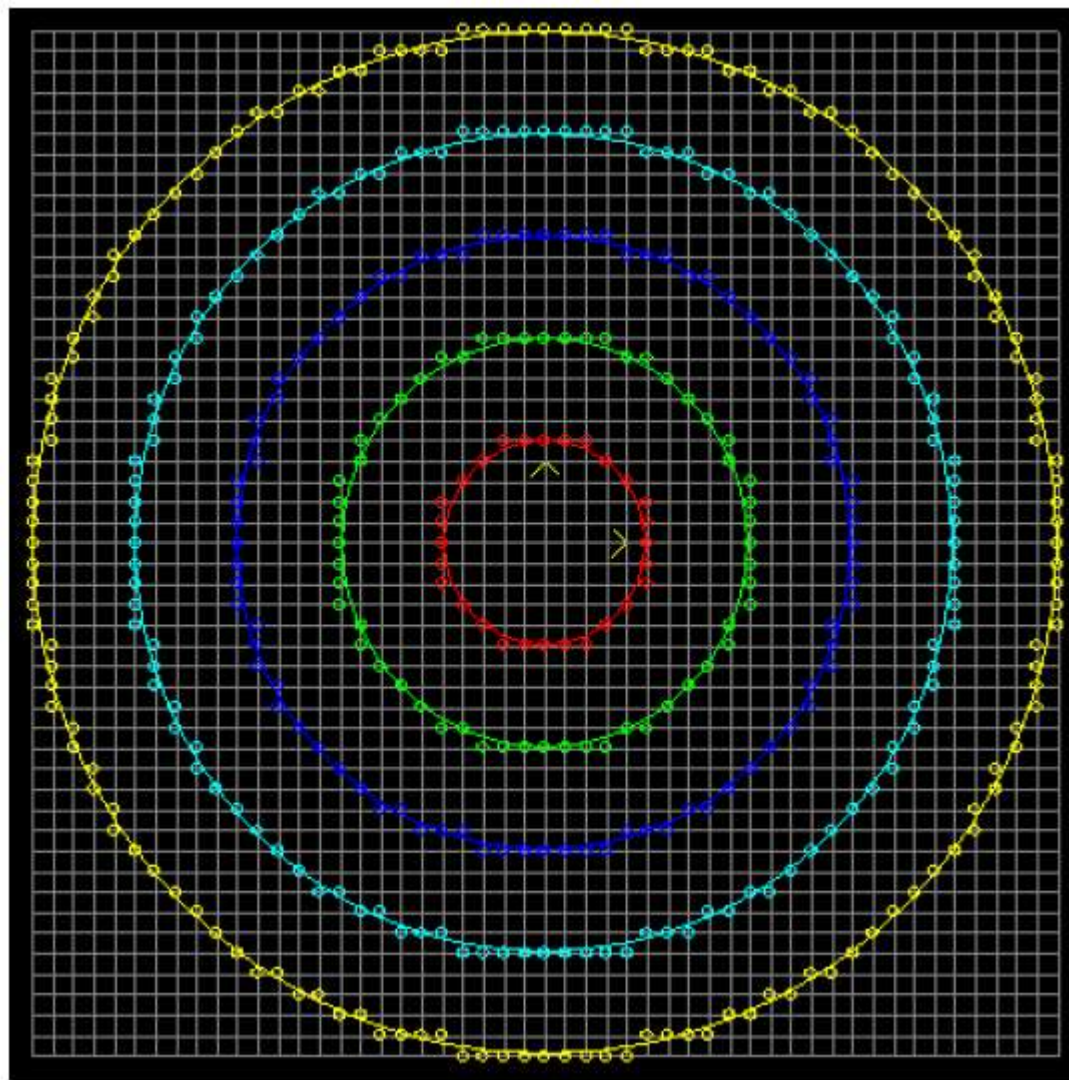
第 2 章 基本图形生成算法 (I)

```
While (y>=x) {  
    CirclePoints(x, y,value);  
    if (d<0) //选择 T  
        d=d+dt;  
        dt+=2;  
        db+=2;  
  
    else { //选择 B  
        d=d+db;  
        dt+=2;  
        db+=4;  
        y--;  
    }  
    x++;  
} //End of while  
} //Finish
```





2.2.5 中点圆整数中点圆整数优化算法 — 例子





总结

- 直线光栅化算法
 - DDA算法
 - Bresenham算法
- 圆光栅化算法
 - 中点算法
 - 中点整数算法
 - 中点整数优化算法
- 基本方法
 - 增量算法
 - 符号算法





2.3 椭圆光栅化算法





2.3.1 椭圆的扫描转换

中点画圆法可以推广到一般二次曲线的生成, 下面以中心在原点的标准椭圆的扫描转换为例说明。 设椭圆的方程为

$$F(x,y)=b^2x^2+a^2y^2-a^2b^2=0$$

其中, a 为沿 x 轴方向的长半轴长度, b 为 y 轴方向的短半轴长度, a 、 b 均为整数。 不失一般性, 我们只讨论第一象限椭圆弧的生成。 需要注意的是, 在处理这段椭圆时, 必须以弧上斜率为-1的点(即法向量两个分量相等的点)作为分界把它分为上部分和下部分, 如图2.6所示。



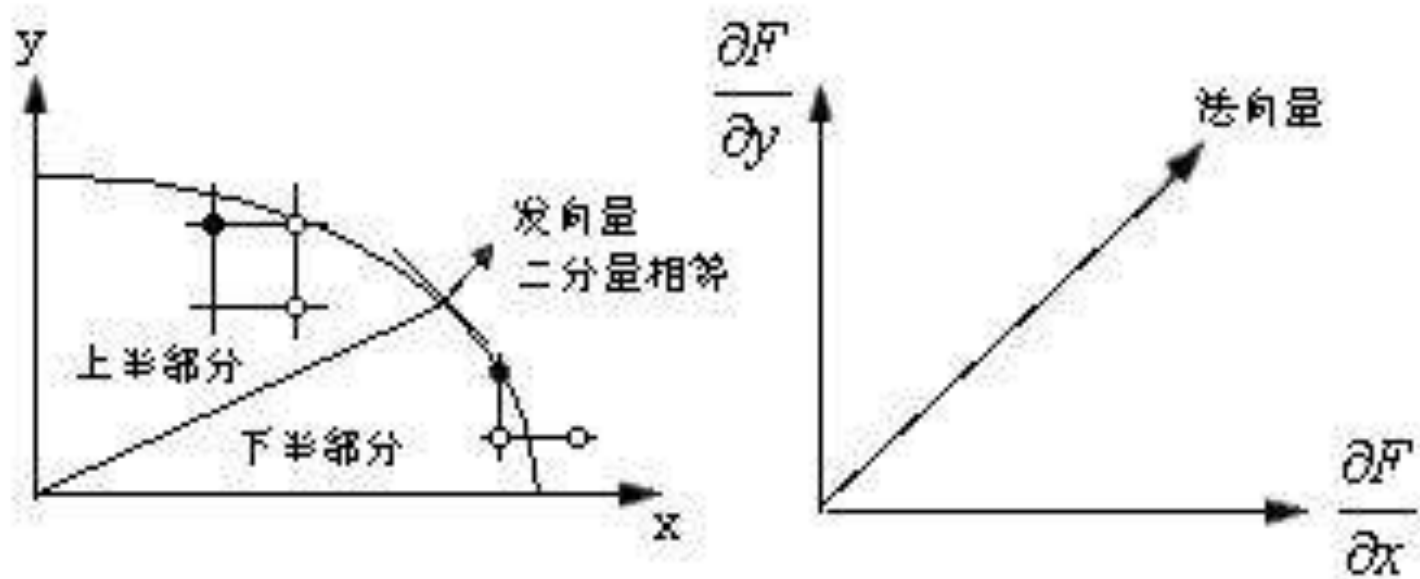


图 2.6 第一象限的椭圆弧





该椭圆上一点 (x, y) 处的法向量为

$$N(x, y) = \frac{\partial F}{\partial x} i + \frac{\partial F}{\partial y} j = 2b^2 xi + 2a^2 yj$$





其中, i 和 j 分别为沿 x 轴和 y 轴方向的单位向量。从图2.6可看出, 在上部分, 法向量的 y 分量更大, 而在下部分, 法向量的 x 分量更大, 因而, 在上部分若当前最佳逼近理想椭圆弧的像素 (x_p, y_p) 满足下列不等式

$$b^2(x_p+1) < a^2(y_p-0.5)$$

而确定的下一个像素不满足上述不等式, 则表明椭圆弧从上部分转入下部分。





在上部分，假设横坐标为 x_p 的像素中与椭圆弧更接近点是 (x_p, y_p) ，那么下一对候选像素的中点是 $(x_p+1, y_p-0.5)$ 。因此判别式为

$$d_1 = F(x_p+1, y_p-0.5) = b^2(x_p+1)^2 + a^2(y_p-0.5)^2 - a^2b^2$$

若 $d_1 < 0$ ，中点在椭圆内，则应取正右方像素，且判别式应更新为

$$\begin{aligned} d'_1 &= F(x_p+2, y_p-0.5) = b^2(x_p+2)^2 + a^2(y_p-0.5)^2 - a^2b^2 \\ &= d_1 + b^2(2x_p+3) \end{aligned}$$





当 $d_1 \geq 0$, 中点在椭圆之外, 这时应取右下方像素, 并且更新判别式为

$$\begin{aligned} d'_1 &= F(x_P+2, y_P-1.5) = b^2(x_P+2)^2 + a^2(y_P-1.5)^2 - a^2b^2 \\ &= d_1 + b^2(2x_P+3) + a^2(-2y_P+2) \end{aligned}$$

由于弧起点为 $(0, b)$, 因此, 第一中点是 $(1, b-0.5)$, 对应的判别式是

$$\begin{aligned} d_{10} &= F(1, b-0.5) = b^2 + a^2(b-0.5)^2 - a^2b^2 \\ &= b^2 + a^2(-b+0.25) \end{aligned}$$





在下部分,应改为从正下方和右下方两个像素中选择下一像素。如果在上部分所选择的最后一像素是 (x_p, y_p) , 则下部分的中点判别式 d_2 的初始值为

$$d_{20}=F(x_p+0.5, y_p-1)=b^2(x_p+0.5)^2+a^2(y_p-1)^2-a^2b^2$$

d_2 在正下方向与右下方向的增量计算与上部分类似,这里不再赘述。下部分弧的终止条件是 $y=0$ 。





第 2 章 基本图形生成算法 (I)

第一象限椭圆弧的扫描转换中点算法的伪C描述如下:

```
void MidpointEllipse(a, b, color)
int a, b, color;
{ int x, y;
  float d1, d2;
  x=0; y=b;
  d1=b*b+a*a*(-b+0.25);
  putpixel(x, y, color);
  while(b*b*(x+1)<a*a*(y-0.5))
  { if(d1<0)
    { d1+=b*b*(2*x+3);
      x++;
    }
  }
```





第 2 章 基本图形生成算法 (I)

```
else { d1+=(b*b*(2*x+3)+a*a*(-2*y+2));
```

```
    x++; y--;
```

```
}
```

```
    putpixel(x, y, color);
```

```
}/*上半部分*/
```

```
d2=sqr(b*(x+0.5))+sqr(a*(y-1))-sqr(a*b);
```

```
while(y>0)
```

```
{ if(d2<0)
```

```
    { d2+=b*b(2*x+2)+a*a*(-2*y+3);
```

```
      x++;
```

```
      y--;
```

```
    }
```

```
    else { d2+=a*a*(-2*y+3);
```

sqr() 为平方函数





```
y--;  
    }  
    putpixel(x, y, color);  
}  
}
```





第 2 章 基本图形生成算法 (I)



第三章 基本图形生成算法(Ⅱ)

- 如何在指定的输出设备上根据坐标描述构造基本二维几何图形（点、直线、圆、椭圆、多边形域、字符串及其相关属性等）。

图形生成的概念

- 图形的生成：是在指定的输出设备上，根据坐标描述构造二维几何图形。
- 图形的扫描转换：在光栅显示器等数字设备上确定一个最佳逼近于图形的象素集的过程。

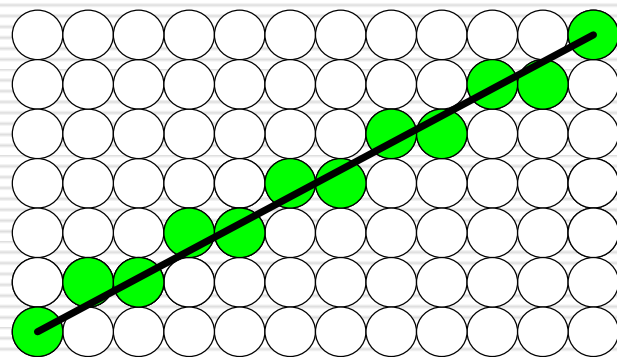


图3.1 用象素点集逼近直线

3.1 多边形的扫描转换与区域填充

- **多边形的扫描转换**主要是通过确定穿越区域的扫描线的覆盖区间来填充。
- **区域填充**是从给定的位置开始涂描直到指定的边界条件为止。

多边形的扫描转换与区域填充

- 多边形的扫描转换
- 边缘填充算法
- 区域填充
- 相关概念

多边形的扫描转换

- 顶点表示用多边形的顶点序列来刻画多边形。
- **点阵表示**是用位于多边形内的象素的集合来刻画多边形。
- **扫描转换多边形**：从多边形的顶点信息出发，求出位于其内部的各个象素，并将其颜色值写入帧缓存中相应单元的过程。

多边形的扫描转换

- X-扫描线算法
- 改进的有效边表算法

X-扫描线算法——原理

- 基本思想：按扫描线顺序，计算扫描线与多边形的相交区间，再用要求的颜色显示这些区间的所有像素。

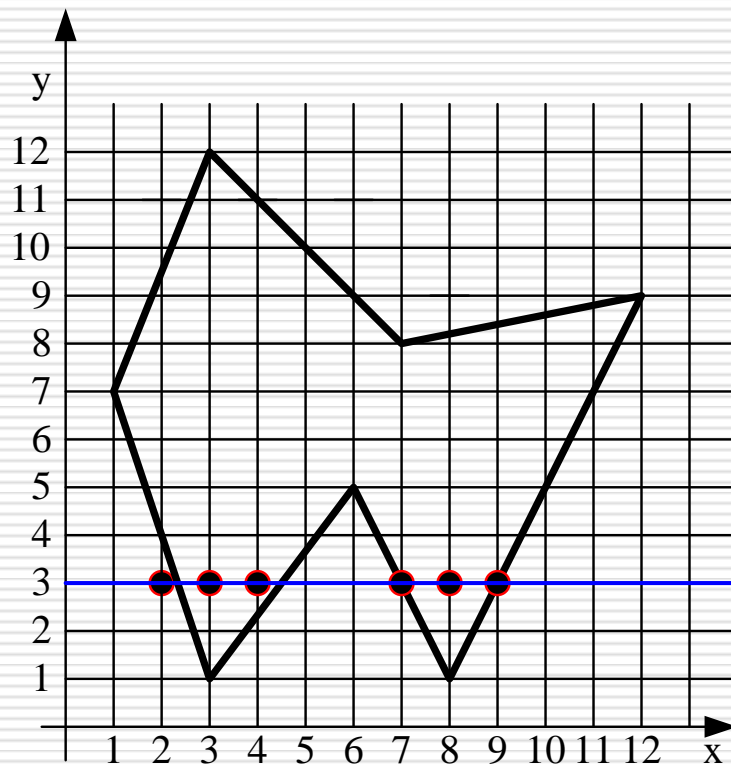


图3.2 x-扫描线算法填充多边形

X-扫描线算法——算法步骤

1. 确定多边形所占有的最大扫描线数，得到多边形顶点的最小和最大 y 值（ y_{\min} 和 y_{\max} ）。
2. 从 $y=y_{\min}$ 到 $y=y_{\max}$ ，每次用一条扫描线进行填充。
3. 对一条扫描线填充的过程可分为四个步骤：
求交；排序；交点配对；区间填色。

X-扫描线算法——取整规则

- 交点的取整规则：使生成的像素全部位于多边形之内。（用于直线等图元扫描转换的四舍五入原则可能导致部分像素位于多边形之外，从而不可用）。
- 假定非水平边与扫描线 $y=e$ 相交，交点的横坐标为 x ，规则如下：

□ **规则1**: X 为小数, 即交点落于扫描线上两个相邻像素之间时:

- 交点位于左边界之上, 向右取整;
- 交点位于右边界之上, 向左取整;

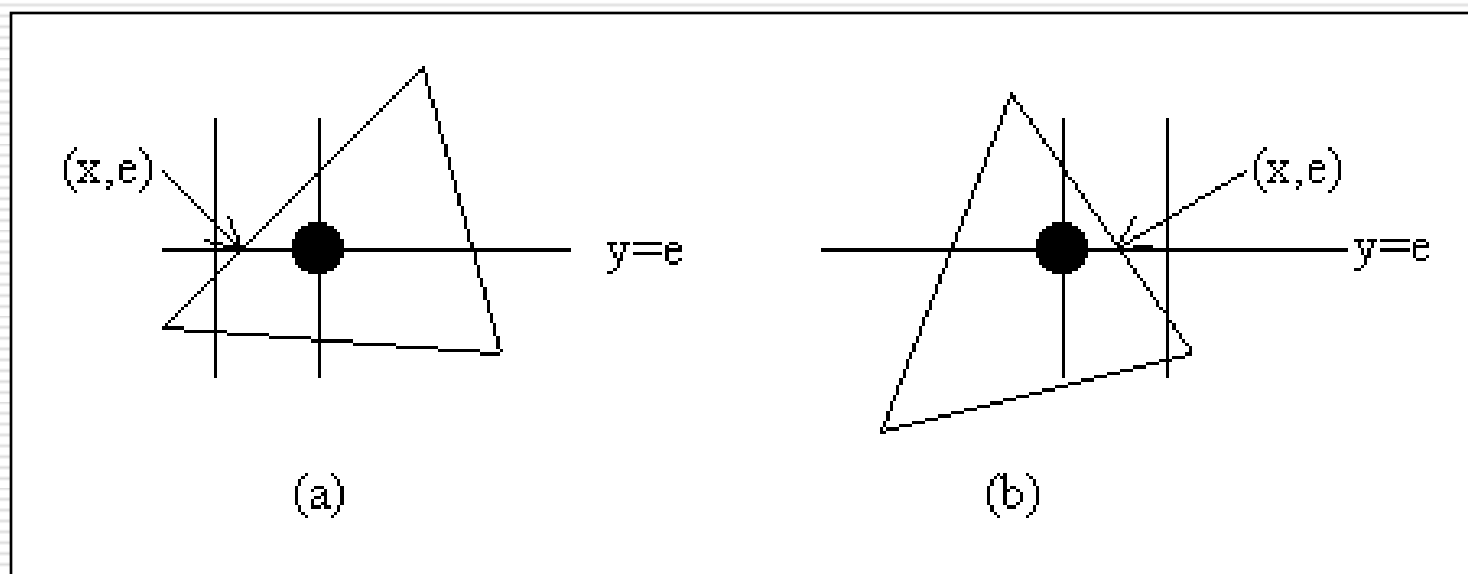


图3.3 取整规则1

X-扫描线算法——取整规则

□ 规则2: 边界上像素的取舍问题，避免填充扩大化。规定落在右边边界上的像素不予填充。（具体实现时，只要对扫描线与多边形的相交区间左闭右开）

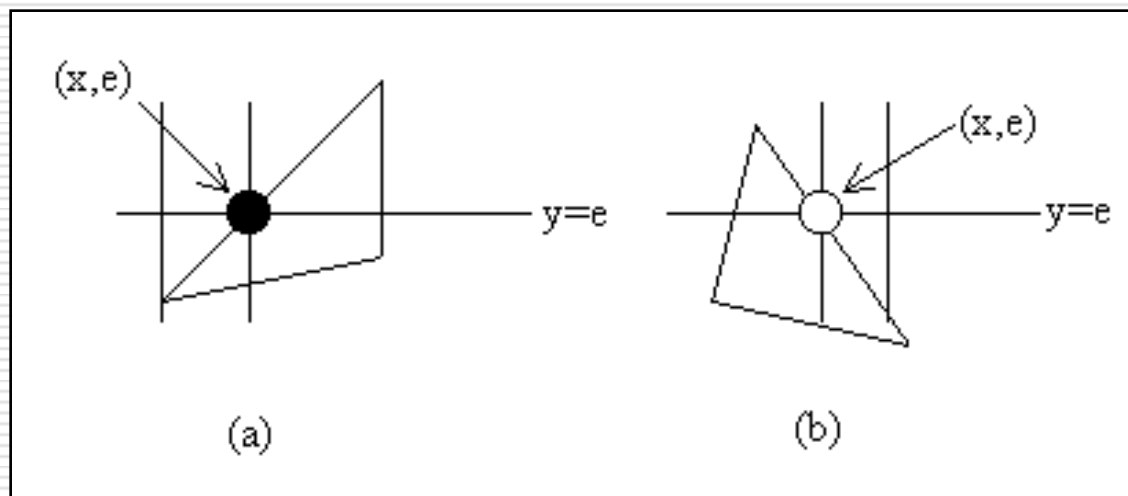


图3.4 取整规则2

X-扫描线算法——取整规则

□ **规则3**: 当扫描线与多边形顶点相交时, 交点的取舍, 保证交点正确配对。

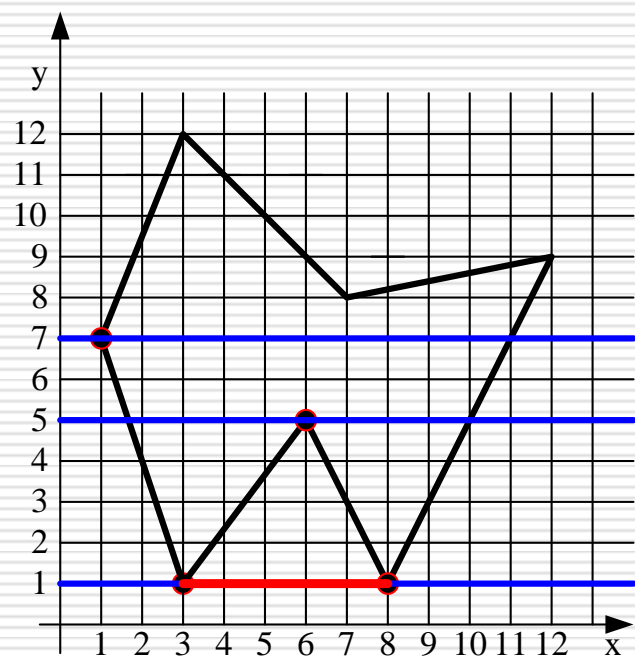


图3.5 取整规则3

X-扫描线算法——取整规则

解决方法:

当扫描线与多边形的顶点相交时,

- 若共享顶点的两条边分别落在扫描线的两边, 交点只算一个;
- 若共享顶点的两条边在扫描线的同一边, 这时交点作为零个或两个。

X-扫描线算法——取整规则

实际处理：只要检查顶点的两条边的另外两个端点的Y值，两个Y值中大于交点Y值的个数是0，1，2，来决定取0，1，2个交点。

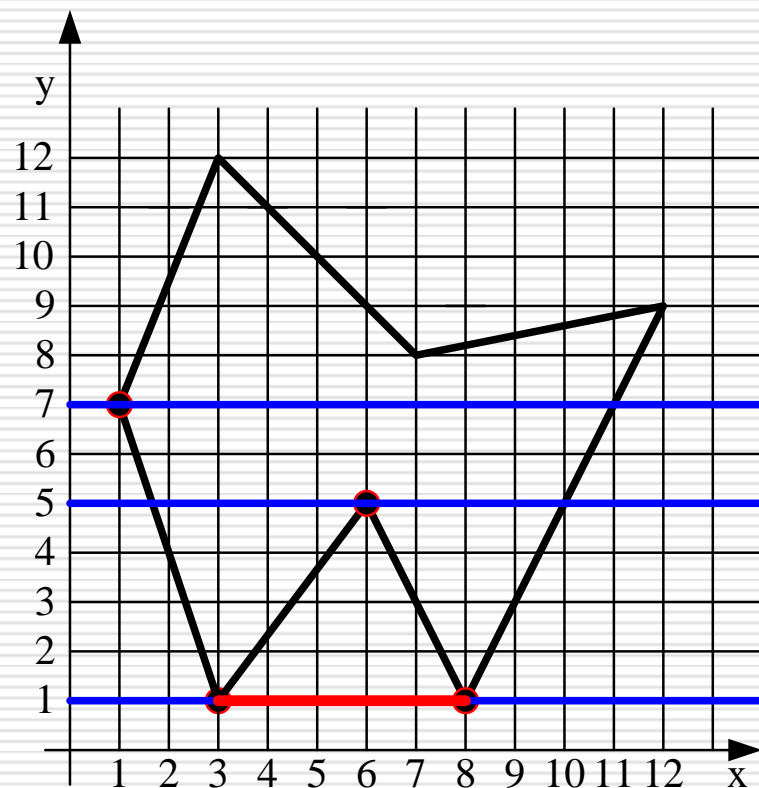


图3.6 取整规则3

X-扫描线算法——取整规则

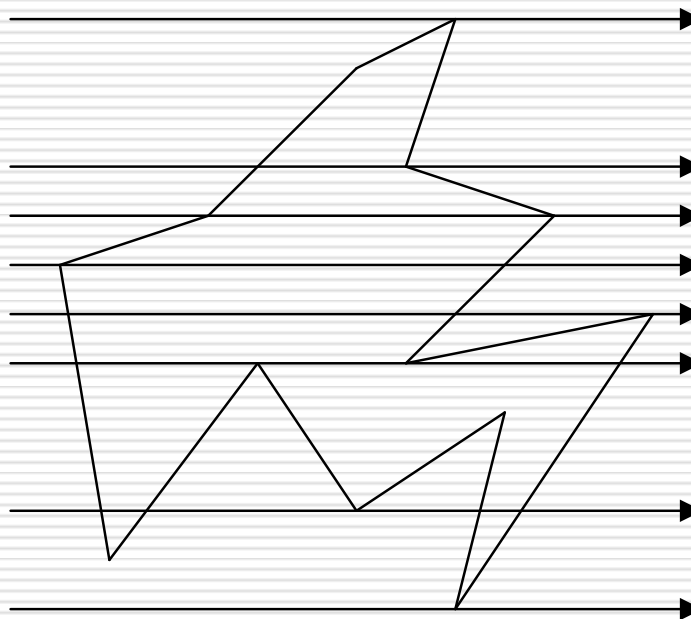


图5.21 与扫描线相交的多边形顶点的交点数

填充过程实例

改进的有效边表算法 (Y连贯性算法)

改进原理:

- 处理一条扫描线时，仅对有效边求交。
- 利用扫描线的连贯性。
- 利用多边形边的连贯性。

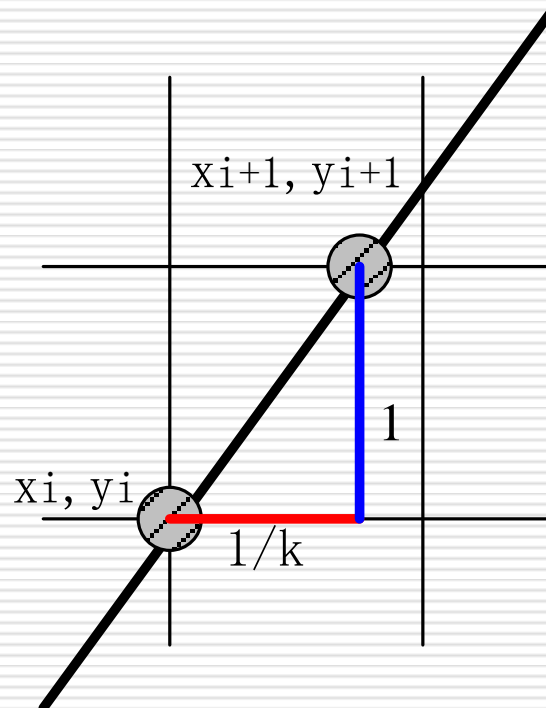


图3.7 与多边形边界相交的两条连续扫描线交点的相关性

改进的有效边表算法 (Y连贯性算法)

- 有效边 (Active Edge): 指与当前扫描线相交的多边形的边, 也称为活性边。
- 有效边表 (Active Edge Table, AET): 把有效边按与扫描线交点 x 坐标递增的顺序存放在一个链表中, 此链表称为有效边表。
- 有效边表的每个结点:

x y_{\max} $1/k$ next

改进的有效边表算法——构造边表

- 首先构造一个纵向链表，链表的长度为多边形所占有的最大扫描线数，链表的每个结点，称为一个桶，则对应多边形覆盖的每一条扫描线。
- 将每条边的信息链入与该边最小 y 坐标 (y_{\min}) 相对应的桶处。也就是说，若某边的较低端点为 y_{\min} ，则该边就放在相应的扫描线桶中。

改进的有效边表算法——构造边表

- 每条边的数据形成一个结点，内容包括：该扫描线与该边的初始交点 x （即较低端点的 x 值）， $1/k$ ，以及该边的最大 y 值 y_{\max} 。

$x|_{y_{\min}} \quad y_{\max} \quad 1/k \quad \text{NEXT}$

- 同一桶中若干条边按 $x|_{y_{\min}}$ 由小到大排序，若 $x|_{y_{\min}}$ 相等，则按照 $1/k$ 由小到大排序。

解决顶点交点计为1时的情形:

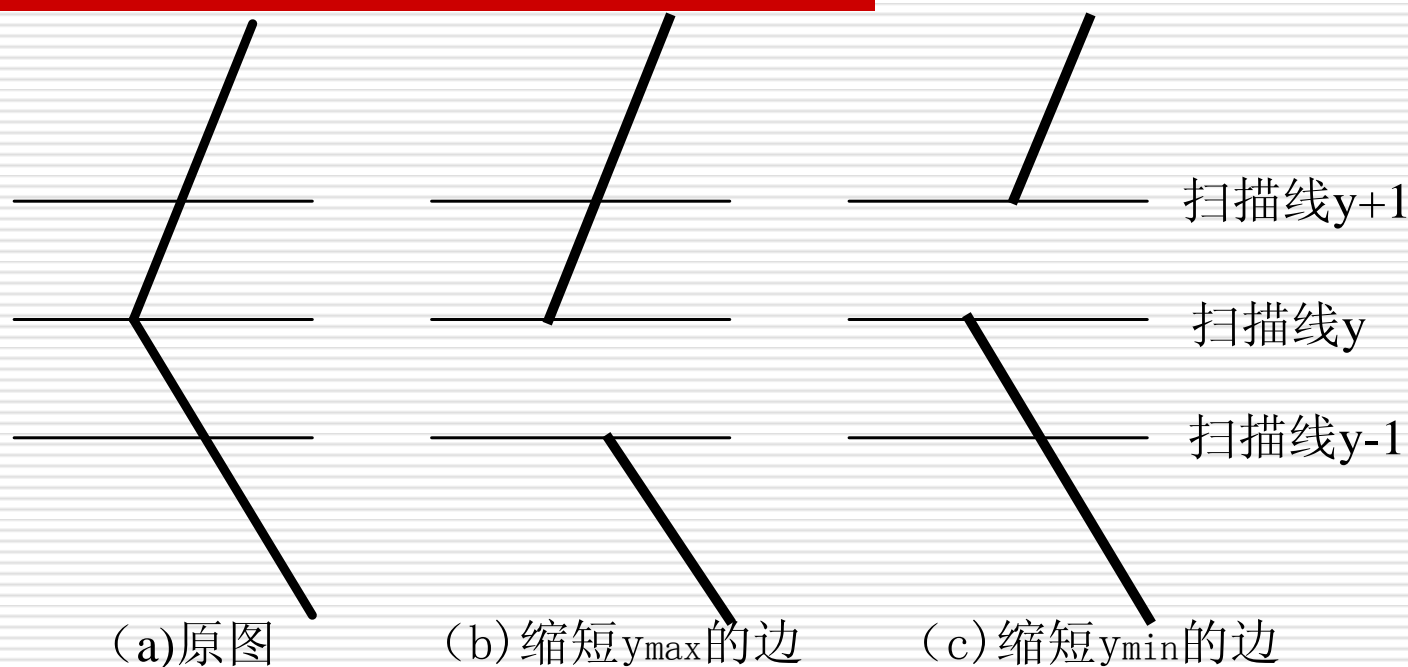


图3.8 将多边形的某些边缩短以分离那些应计为1个交点的顶点

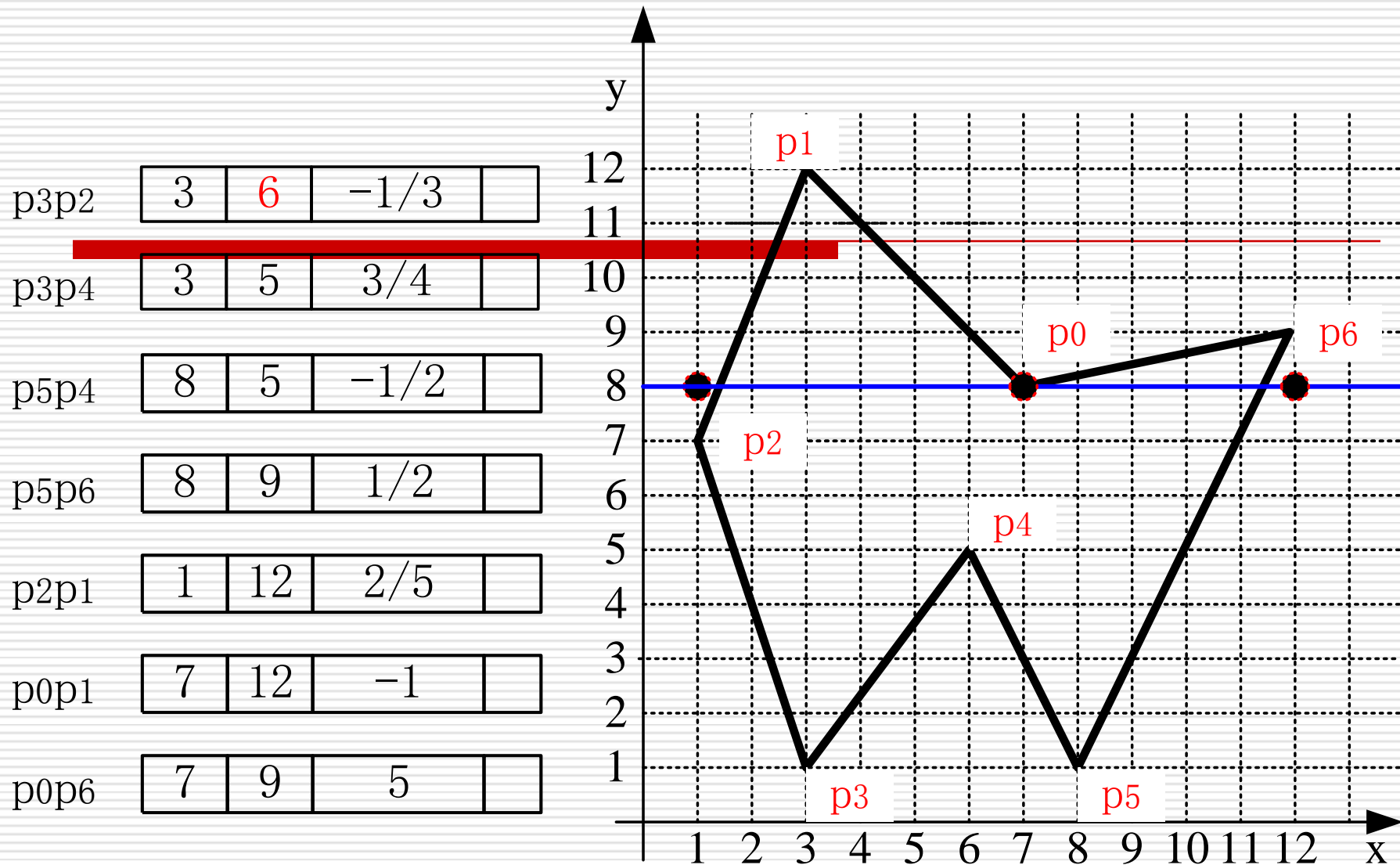


图3.9 多边形 $P_0P_1P_2P_3P_4P_5P_6$

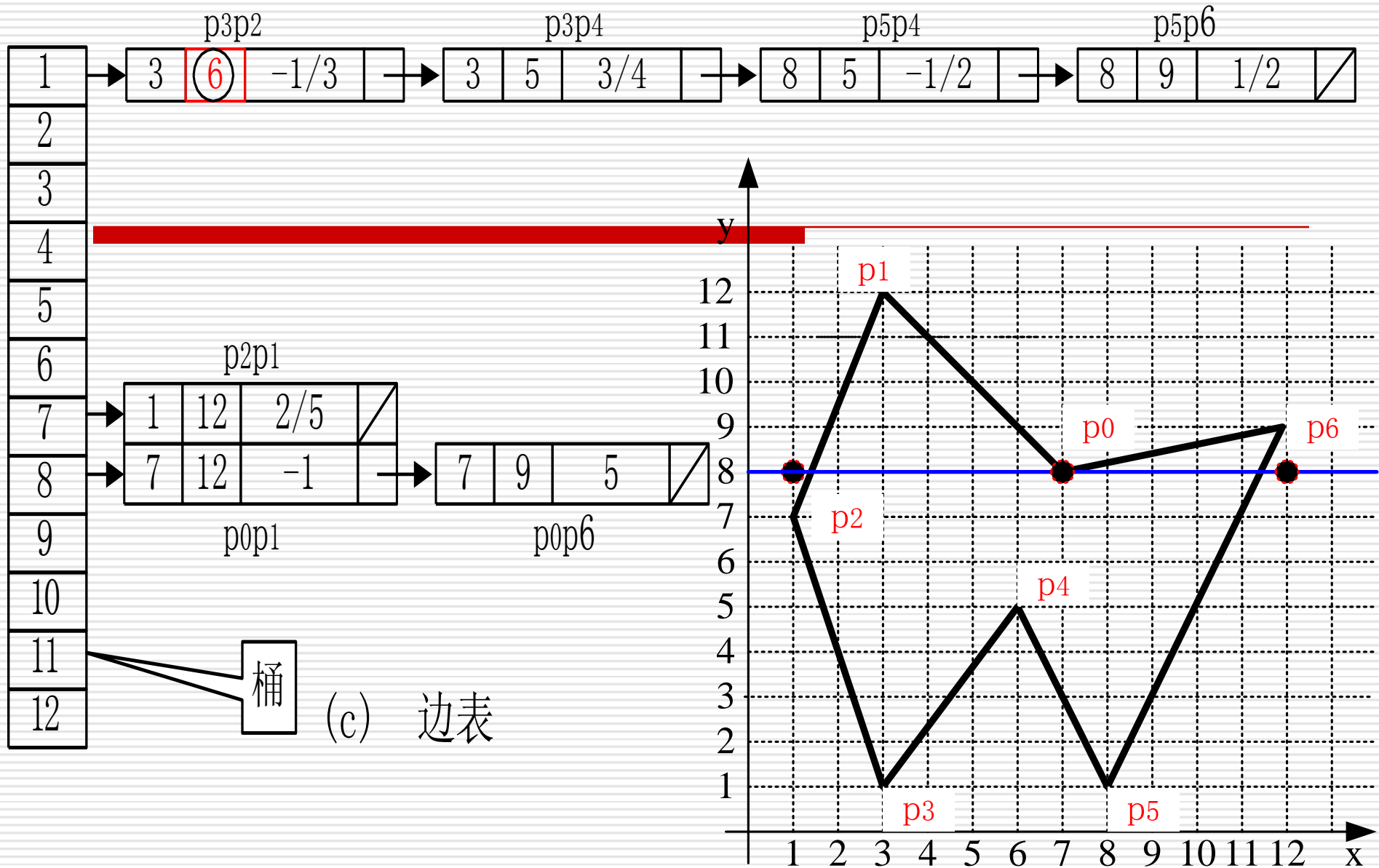


图3.10 多边形 $P_0P_1P_2P_3P_4P_5P_6$

改进的有效边表算法——算法步骤

- (1)初始化：构造边表，AET表置空；
- (2)将第一个不空的ET表中的边与AET表合并；
- (3)由AET表中取出交点对进行填充。填充之后删除 $y=y_{\max}$ 的边；
- (4) $y_{i+1}=y_i+1$,根据 $x_{i+1}=x_i+1/k$ 计算并修改AET表，同时合并ET表中 $y=y_{i+1}$ 桶中的边，按次序插入到AET表中，形成新的AET表；
- (5)AET表不为空则转(3)，否则结束。

边缘填充算法

□ 基本思想：按任意顺序处理多边形的每条边。

处理时，先求出该边与扫描线的交点，再对扫描线上交点右方的所有像素取反。

□ 算法简单，但对于复杂图型，每一像素可能被访问多次

栅栏填充算法

- ❑ 栅栏指的是一条过多边形顶点且与扫描线垂直的直线。它把多边形分为两半。
- ❑ 基本思想：按任意顺序处理多边形的每一条边，但处理每条边与扫描线的交点时，将交点与栅栏之间的像素取反。
- ❑ 这种算法尽管减少了被重复访问像素的数目，但仍有一些像素被重复访问。

边标志算法

- 基本思想：先用特殊的颜色在帧缓存中将多边形的边界勾画出来，然后将着色的象素点依 x 坐标递增的顺序配对，再把每一对象素构成的区间置为填充色。
- 分为两个步骤：打标记-多边形扫描转化；填充。
- 当用软件实现本算法时，速度与改进的有效边表算法相当，但本算法用硬件实现后速度会有很大提高。

区域填充（种子填充）

- 基本概念
- 区域的表示方法
- 区域的分类
- 区域填充算法

基本概念

- **区域填充**是指从区域内的某一个像素点（种子点）开始，由内向外将填充色扩展到整个区域内的过程。
- **区域**是指已经表示成点阵形式的填充图形，它是相互连通的一组像素的集合。

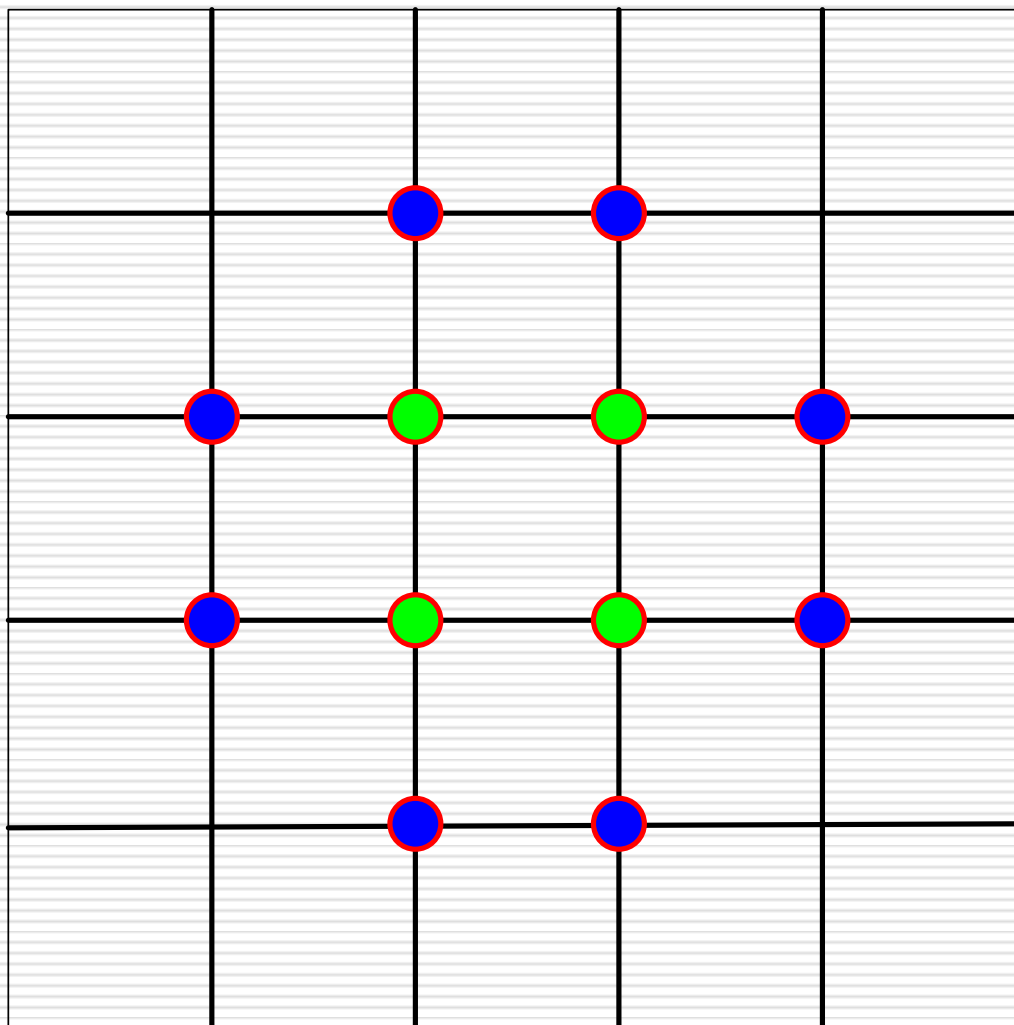


图3.12 区域的概念

区域的表示方法

- **边界表示法**：把位于给定区域的边界上的象素一一列举出来的方法。
- 边界表示法中，由于边界由特殊颜色指定，填充算法可以逐个象素地向外处理，直到遇到边界颜色为止，这种方法称为边界填充算法（Boundary-fill Algorithm）。

□ **内点表示法**：枚举出给定区域内所有象素的表示方法。以内点表示法为基础的区域填充算法称为**泛填充算法（Flood-fill Algorithm）**。

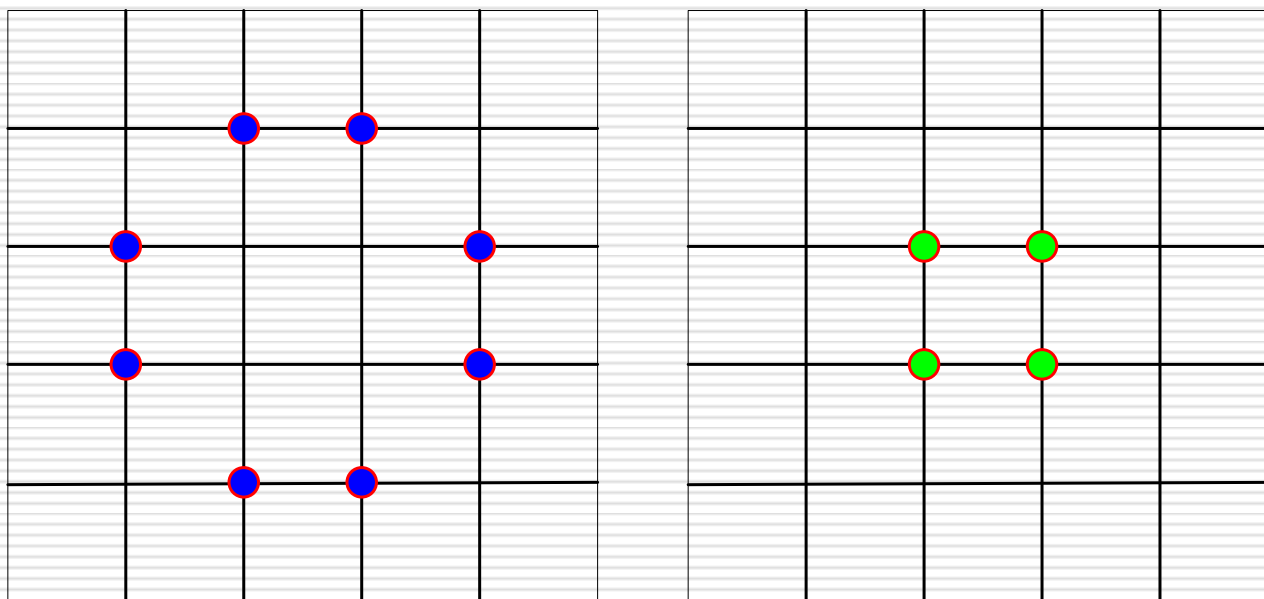
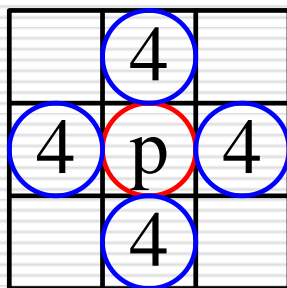


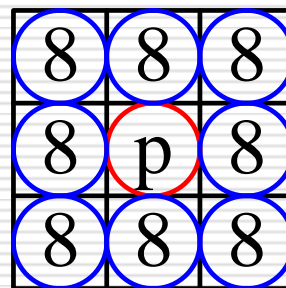
图3-13 区域的表示方法

区域的分类

4-连通区域，8-连通区域



(a) 4-邻接点



(b) 8-邻接点

图3-14 4-邻接点与8-邻接点

区域的分类

- **4-连通区域**: 从区域上的一点出发, 通过访问已知点的**4-邻接点**, 在不越出区域的前提下, 遍历区域内的所有象素点。
- **8-连通区域**: 从区域上的一点出发, 通过访问已知点的**8-邻接点**, 在不越出区域的前提下, 遍历区域内的所有象素点。

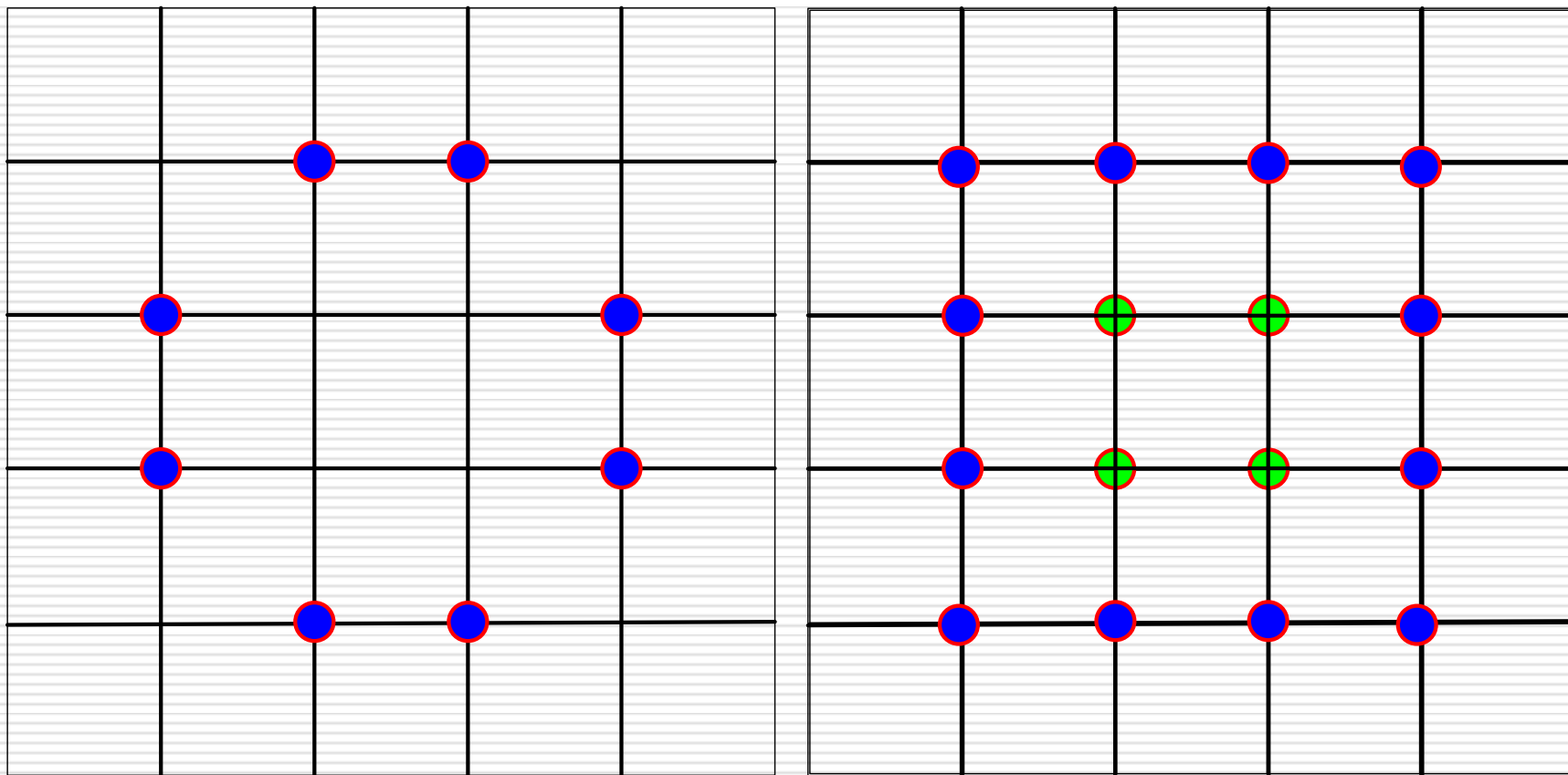


图3.15 4-连通与8-连通区域

4连通与8连通区域的区别

- 连通性： 4连通可看作8连通区域，但对边界有要求。
- 对边界的要求。

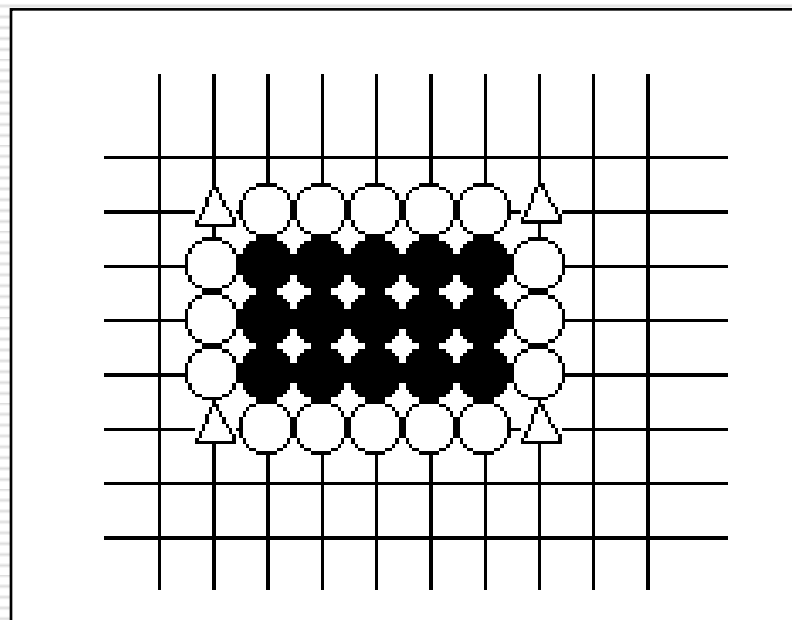


图3-164 - 连通与8 - 连通区域

区域填充算法

- 区域填充算法（边界填充算法和泛填充算法）
是根据区域内的一个已知像素点（种子点）出发，找到区域内其他像素点的过程，所以把这一类算法也成为种子填充算法。

区域填充算法——边界填充算法

- ❑ 算法的输入：种子点坐标 (x, y) ，填充色以及边界颜色。
- ❑ 利用堆栈实现简单的种子填充算法

算法从种子点开始检测相邻位置是否是边界颜色，若不是就用填充色着色，并检测该像素点的相邻位置，直到检测完区域边界颜色范围内的所有像素为止。

区域填充算法——边界填充算法

栈结构实现4-连通边界填充算法的算法步骤为：

种子像素入栈；当栈非空时重复执行如下三步操作：

(a)栈顶像素出栈；

(b)将出栈像素置成填充色；

(c)检查出栈像素的4-邻接点，若其中某个像素点不是边界色且未置成多边形色，则把该像素入栈。

区域填充算法——边界填充算法

栈结构实现8-连通边界填充算法的算法步骤为：

种子像素入栈；当栈非空时重复执行如下三步操作：

(a)栈顶像素出栈；

(b)将出栈像素置成填充色；

(c)检查出栈像素的8-邻接点，若其中某个像素点不是边界色且未置成多边形色，则把该像素入栈。

区域填充算法——边界填充算法

- ❑ 可以用于填充带有内孔的平面区域。
- ❑ 把太多的像素压入堆栈，降低了效率，同时需要较大的存储空间。
- ❑ 递归执行，算法简单，但效率不高，区域内每一像素都引起一次递归，进/出栈费时费内存。
- ❑ 通过沿扫描线填充水平像素段，来代替处理4-邻接点和8-邻接点。

区域填充算法——边界填充算法

- 扫描线种子填充算法：
扫描线通过在任意不间断扫描线区间中只取一个种子像素的方法使堆栈的尺寸极小化。不间断区间是指在一条扫描线上的一组相邻像素。

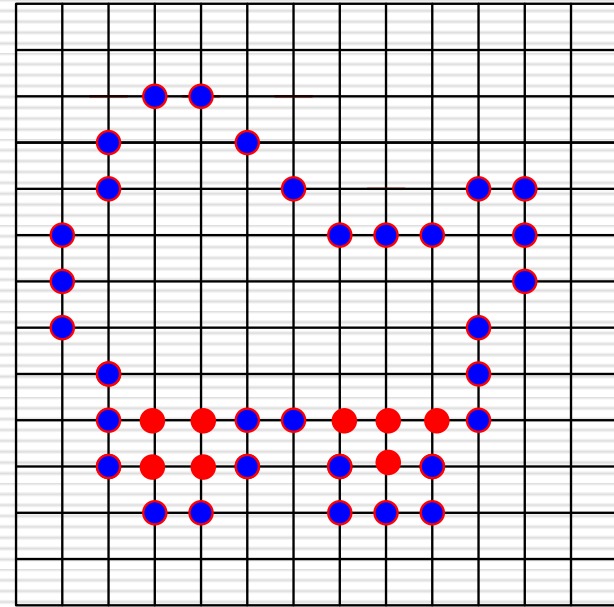


图3-17 扫描线种子填充算法

区域填充算法——边界填充算法

- 基本过程：当给定种子点时，首先填充种子点所在的扫描线上的位于给定区域的一个区段，然后确定与这一区段相通的上下两条扫描线上位于给定区域内的区段，并依次保存下来。反复这个过程，直到填充结束。

区域填充算法——边界填充算法

- 扫描线种子填充算法：我们可以在任意一个扫描线与多边形的相交区间中，只取一个种子像素，并将种子像素入栈，当栈非空时作如下四步操作：

- (1) 栈顶像素出栈;
- (2) 沿扫描线对出栈像素的左右像素进行填充, 直至遇到边界像素为止, 也就是对包含出栈像素的整个区间进行填充;
- (3) 上述区间内最左最右的像素分别记为 x_l 和 x_r ;
- (4) 在区间 $[x_l, x_r]$ 中检查与当前扫描线相邻的上下两条扫描线的有关像素是否全为边界像素或已填充的像素, 若存在非边界、非填充的像素, 则把每一区间的最右像素入栈。

上述改进之后的算法称之为扫描线种子填充算法，其伪C语言描述如下：

```
void scanline-seed-fill(polydef, color, x, y)
    多边形定义 polydef;
    int color, x, y; /* (x, y)为种子像素*/

    {
        int x, y, x0, x1, xr
        push(seed(x, y)); /* 种子像素x栈 */
```


while (栈非空)

```
{ pop (pixel(x, y)); /* 栈顶像素出栈, 并置为
    (x, y) */
    putpixel(x, y, color);
    x0=x+1;
    while (getpixel(x0, y)的值不等于边界像素颜色值) /* 填充出栈像素的右方像素*/
        { putpixel(x0, y, color);
            x0=x0+1;
        }
    xr=x0-1; /*最右像素*/
    x0=x-1;
```

```
while(getpixel(x0, y) 的值不等于边界像素颜色值)
    /* 填充出栈像素的左方 像素*/
        { putpixel(x0, y, color);
          x0=x0-1;
        }
    x1=x0+1; /*最左像素*/
    /* 检查上一条扫描线, 若存在非边界且未填充
    的像素, 则将各连续区间的最右像素入栈 */
    x0=x1;  y=y+1;
```

```

while(x0<=xr)
{
    flag=0;
    while(( getpixel(x0, y)的值不等于边界像素颜色值)&& (getpixel(x0, y)的值不等于多边形内像素颜色值)&& (x0<xr))
    {
        if (flag==0) flag=1;
        x0++;
    }
    if(flag==1)
    {
        if(( x0==xr)&&(getpixel(x0, y)的值不等于边界像素颜色值)&& (getpixel(x0, y) 的值不等于多边形内像素颜色值))    push((x0, y));
    }
}

```

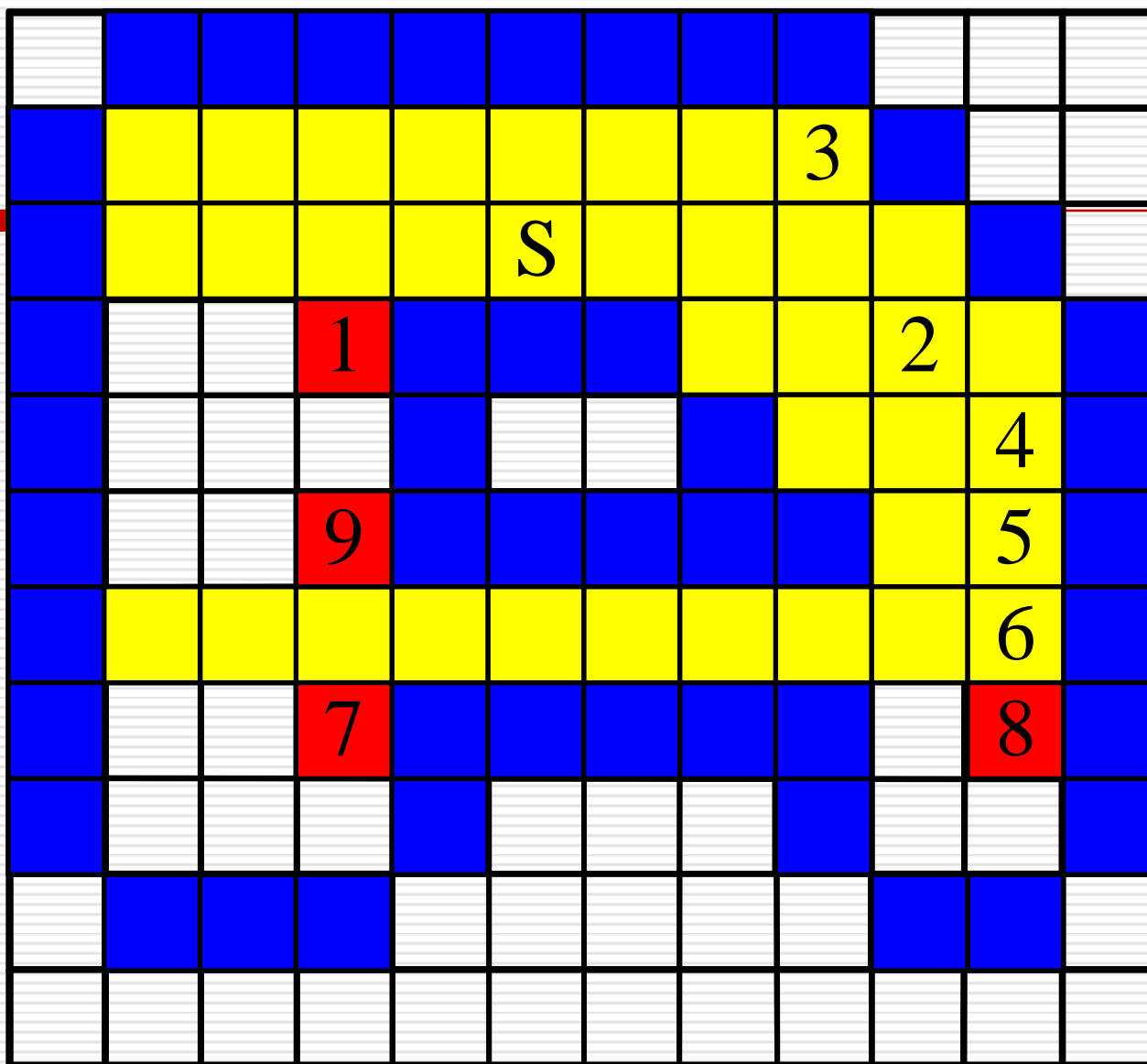
```

else    push((x0-1, y));
        flag=0;
    }
    xnextspan = x0;
    while(( pixel(x0, y1) 等于边界像素颜色值
) || (pixel(x0, y1) 等于多边形内像素颜色值
)&&(x0<=xr))
        x0++;
        if(xnextspan ==x0)      x0++;
} /* while(x0<=xr) */

```

/* 检查下一条扫描线，若存在非边界，未填充的像素，则将各连续区间的最右像素入栈；处理与前面一条扫描线的算法相同，只要把 $y+1$ 换为 $y-1$ 即可 */

```
    } /* while (栈非空) */  
}
```



相关概念——内外测试

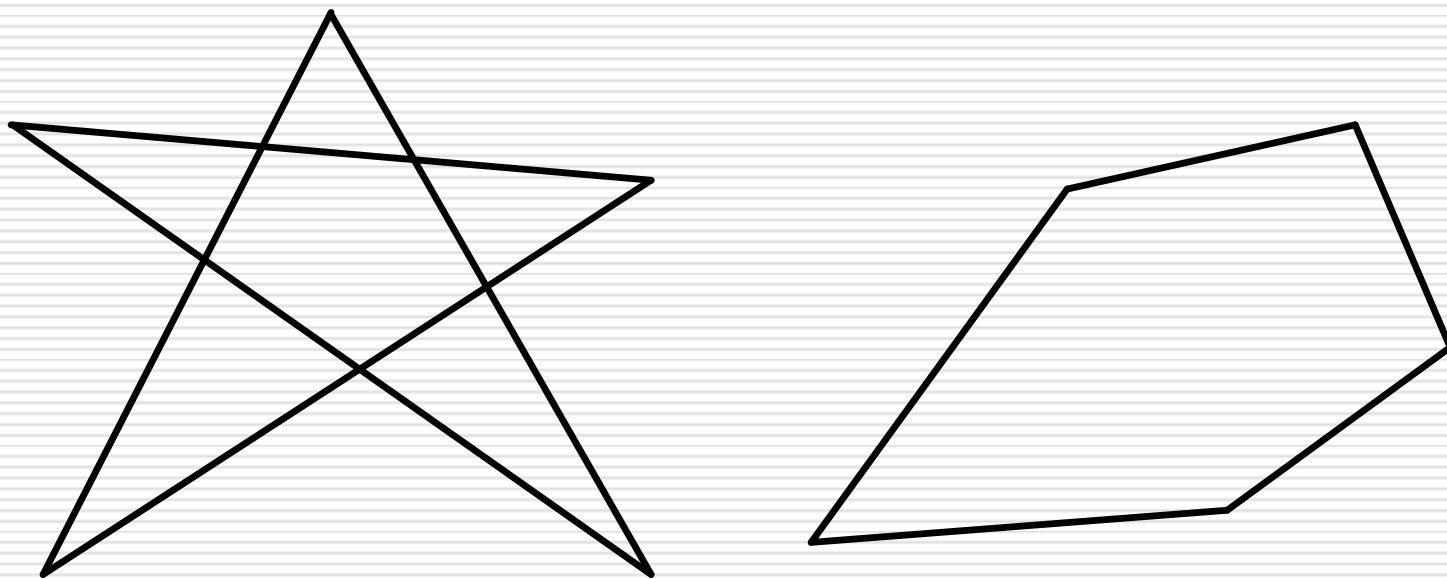
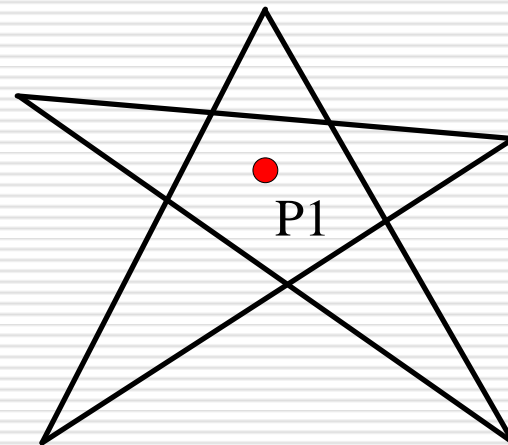


图3.19 不自交的多边形与自相交的多边形

相关概念——内外测试

□ 奇-偶规则 (Odd-even Rule)

从任意位置 p 作一条射线，若与该射线相交的多边形边的数目为奇数，则 p 是多边形内部点，否则是外部点。



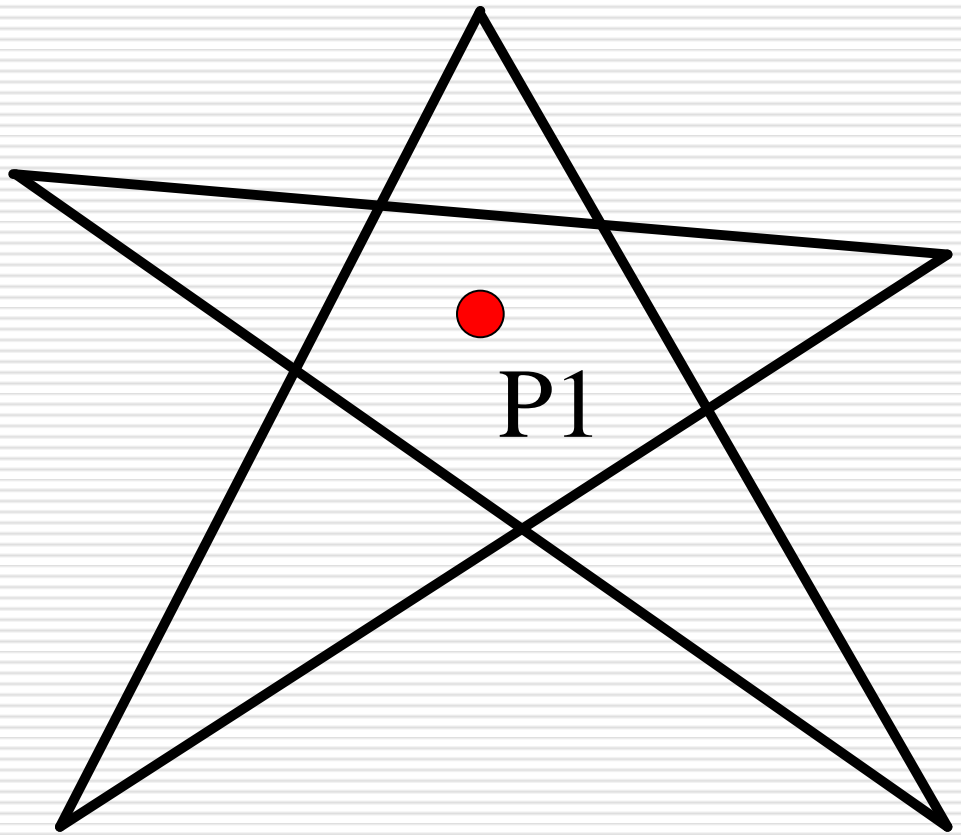
相关概念——内外测试

非零环绕数规则 (Nonzero Winding Number Rule)

- 首先使多边形的边变为矢量。
- 将环绕数初始化为零。
- 再从任意位置 p 作一条射线。当从 p 点沿射线方向移动时，对在每个方向上穿过射线的边计数，每当多边形的边从右到左穿过射线时，环绕数加1，从左到右时，环绕数减1。
- 处理完多边形的所有相关边之后，若环绕数为非零，则 p 为内部点，否则， p 是外部点。

相关概念——内外测试

□ 两种规则的比较



相关概念——曲线边界区域填充

□ 相交计算中包含了非线性边界。

□ 对于简单曲线：

（1）计算曲线相对两侧的两个扫描线交点

（2）简单填充在曲线两侧上的边界点间的
水平像素区间。

3.2 字符处理

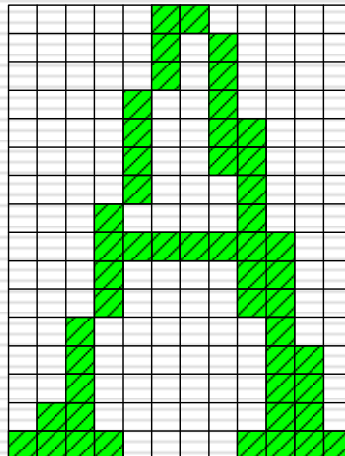
- ❑ **ASCII 码**：“美国信息交换用标准代码集”（**American Standard Code for Information Interchange**），简称**ASCII**码。
- ❑ **国际码**：“中华人民共和国国家标准信息交换编码，简称为国际码，代号**GB2312-80**。
- ❑ **字库**：字库中储存了每个字符的图形信息。
- ❑ **矢量字库和点阵字库**。

字符处理——点阵字符

- 在点阵表示中，每个字符由一个点阵位图来表示。
- 显示时：形成字符的像素图案。

0	0	0	0	0	1	1	0	0	0	0	0
0	0	0	0	0	1	0	1	0	0	0	0
0	0	0	0	0	1	0	1	0	0	0	0
0	0	0	0	1	0	0	1	0	0	0	0
0	0	0	0	1	0	0	1	1	0	0	0
0	0	0	0	1	0	0	1	1	0	0	0
0	0	0	0	1	0	0	0	1	0	0	0
0	0	0	1	0	0	0	0	1	0	0	0
0	0	0	1	1	1	1	1	1	1	0	0
0	0	0	1	0	0	0	0	1	1	0	0
0	0	0	1	0	0	0	0	1	1	0	0
0	0	1	0	0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	0	1	1	0
0	0	1	0	0	0	0	0	0	1	1	0
0	1	1	0	0	0	0	0	0	1	1	0
1	1	1	1	0	0	0	0	1	1	1	1

(a)字符A的点阵位图



(a)字符A的像素图案

图3.20 字符A的点阵表示

字符处理——点阵字符

- 定义和显示直接、简单。
- 存储需要耗费大量空间。
 - 从一组点阵字符生成不同尺寸和不同字体的其他字符。
 - 采用压缩技术。

字符处理——矢量字符

- ❑ 矢量字符采用直线和曲线段来描述字符形状，矢量字符库中记录的是笔划信息。
- ❑ 显示时：解释字符的每个笔划信息。

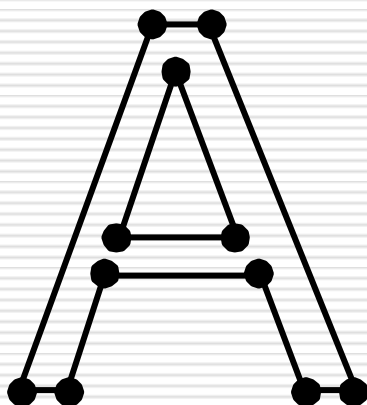


图3.21 字符A的矢量表示

字符处理——矢量字符

- 轮廓字型法采用直线或二、三次曲线的集合来描述一个字符的轮廓线，轮廓线构成了一个或若干个封闭区域，显示时只要填充这些区域就可产生相应的字符。
- 显示时可以“无极放缩”。
- 占用空间少，美观，变换方便。

3.3 属性处理

- 图素或图段的外观由其属性决定。
- 图形软件中实现属性选择的方法是提供一张系统当前属性值表，通过调用标准函数提供属性值表的设置和修改。进行扫描转换时，系统使用属性值表中的当前属性值进行显示和输出。

属性处理

- 线型与线宽
- 区域填充属性

线型与线宽——线型

- 线型的显示可用象素段方法实现：针对不同的线型，画线程序沿路径输出一些连续象素段，在每两个实心段之间有一段中间空白段，他们的长度（象素数目）可用象素模板(pixel mask)指定。
- 存在问题：如何保持任何方向的划线长度近似地相等。

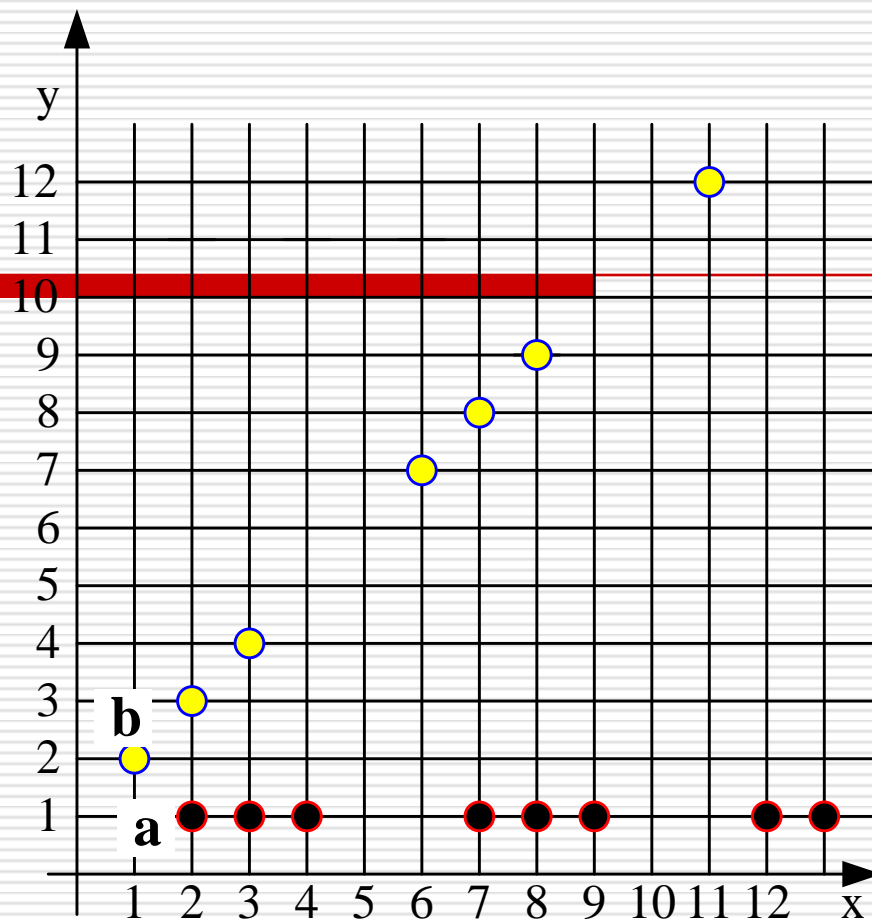


图3.22 相同数目象素显示的不等长划线

□ 可根据线的斜率来调整实心段和中间空白段的像素数目。

线型与线宽——线宽

□ 线刷子：垂直刷子、水平刷子

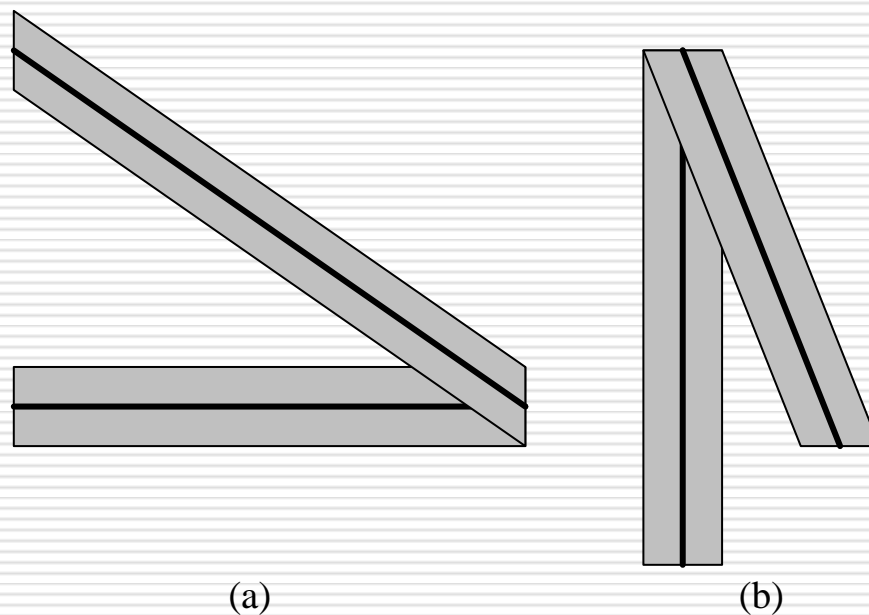


图3.23 线刷子

线型与线宽——线型

- ❑ 实现简单、效率高。
- ❑ 斜线与水平(或垂直)线不一样粗。
- ❑ 当线宽为偶数个像素时，线的中心将偏移半个像素。
- ❑ 利用线刷子生成线的始末端总是水平或垂直的，看起来不太自然。

解决：添加“线帽（line cap）”

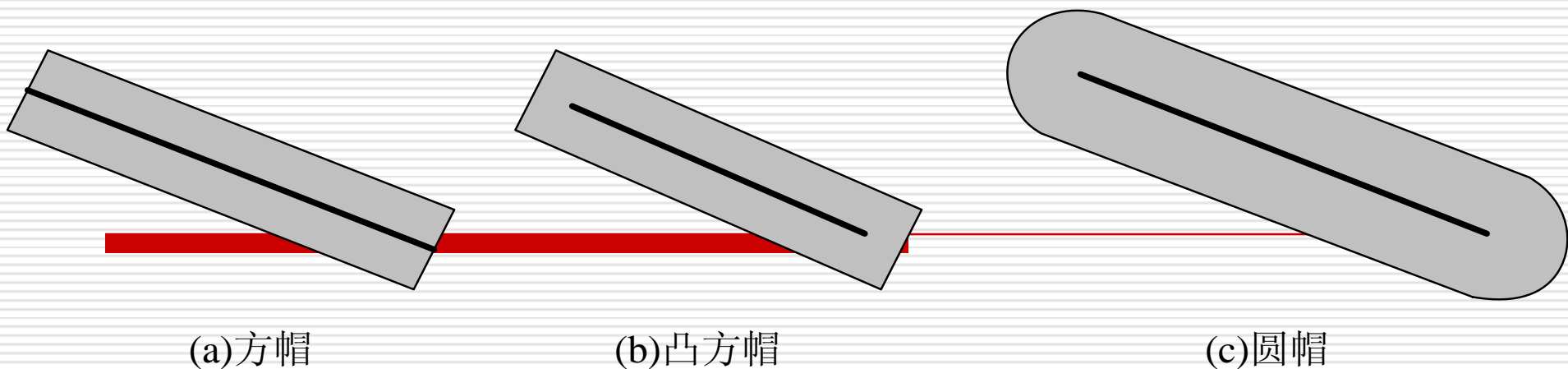


图3.24 线帽

- 方帽：调整端点位置，使粗线的显示具有垂直于线段路径的正方形端点。
- 凸方帽：简单将线向两头延伸一半线宽并添加方帽。
- 圆帽：通过对每个方帽添加一个填充的半圆得到。

线型与线宽——线型

- 当比较接近水平的线与比较接近垂直的线汇合时，汇合处外角将有缺口。

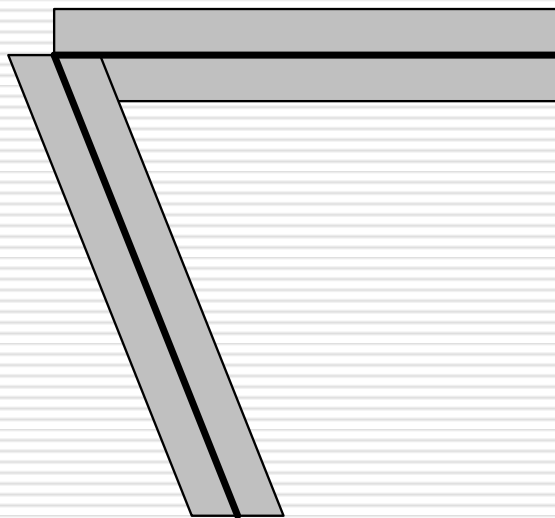
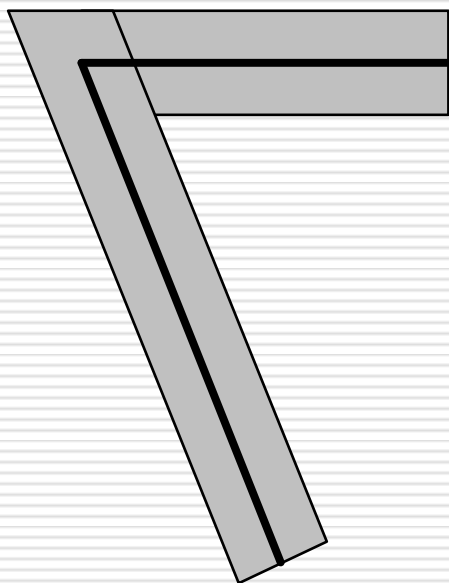
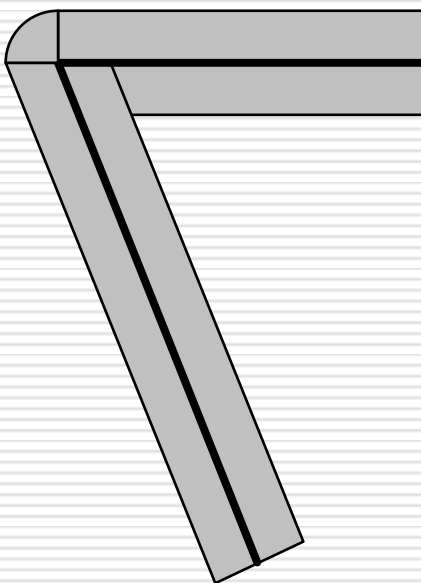


图3.25 线刷子产生的缺口

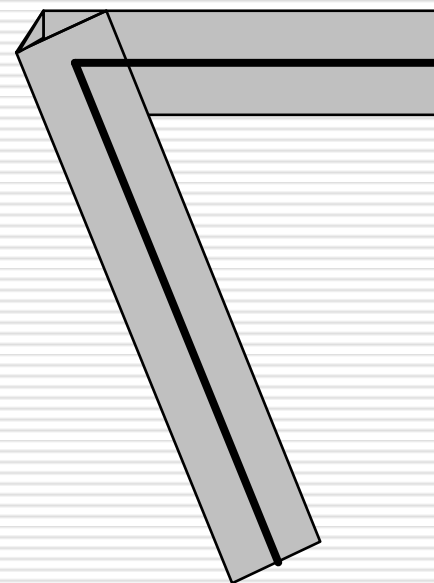
□ 解决：斜角连接（miter join）、圆连接
~~（round join）、斜切连接（bevel join）~~



(a)斜角连接



(b)圆连接



(c)斜切连接

图3.26 线刷子产生的缺口

方刷子

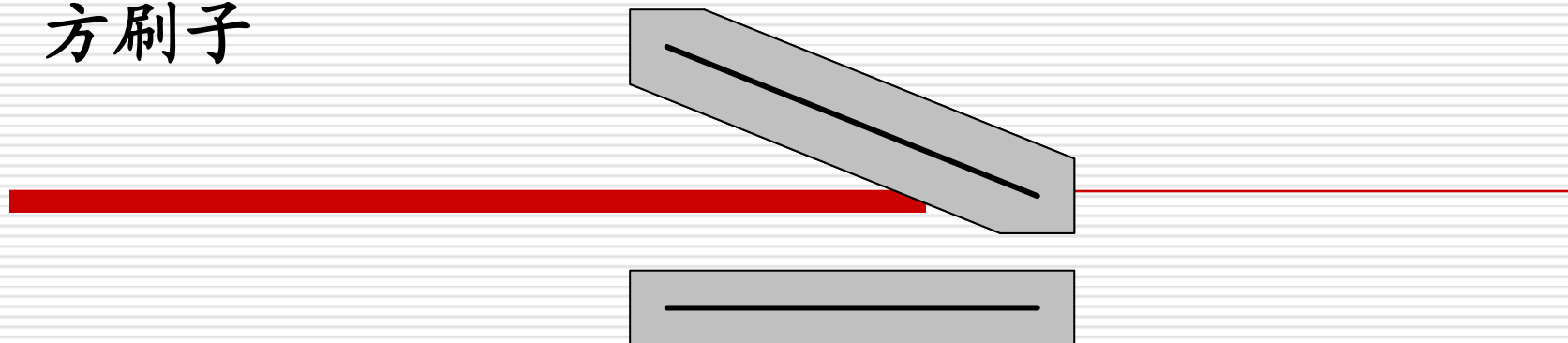


图3.27 方刷子

特点:

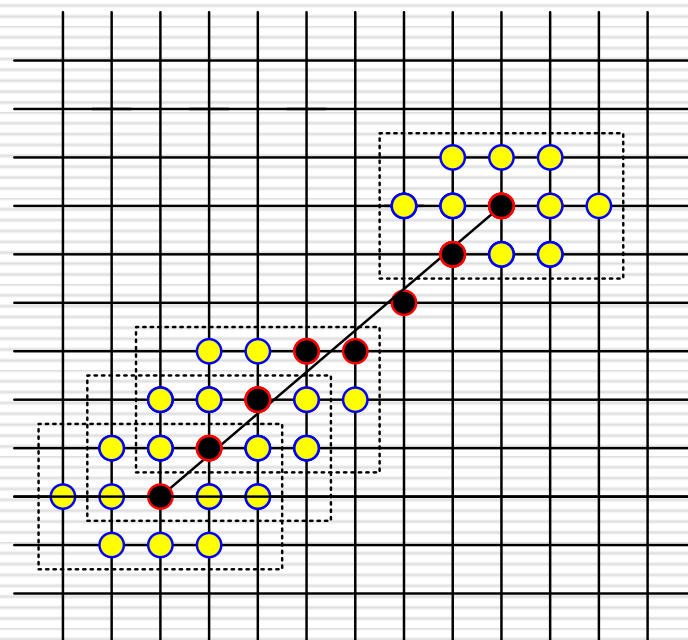
- ❑ 方刷子绘制的线条（斜线）比用线刷子所绘制的线条要粗一些。
- ❑ 方刷子绘制的斜线与水平（或垂直）线不一样粗。
- ❑ 方刷子绘制的线条自然地带有一个“方线帽”。

线型与线宽——线型

- 区域填充
- 改变刷子形状

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

(a) 像素模板



(b) 用该模板进行线宽处理

图3.28 利用像素模板进行线宽处理

线型与线宽——曲线的线型和线宽

□ 线型：可采用像素模板的方法。

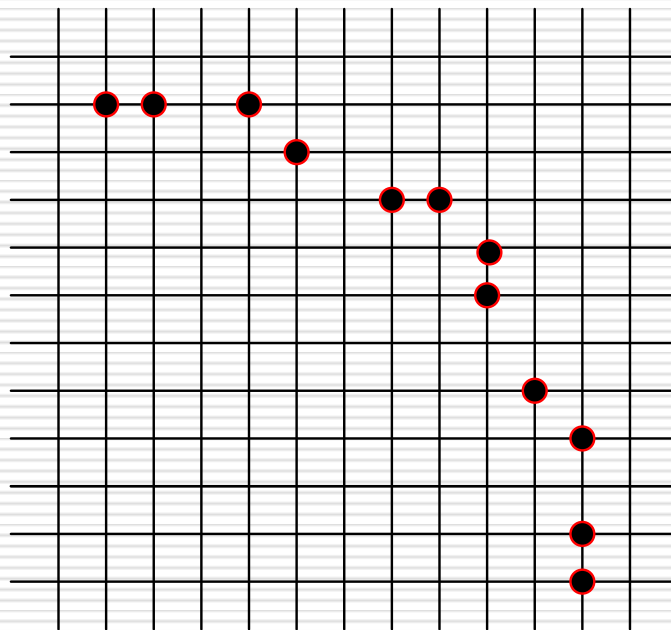


图3.29 利用模板110进行圆的线型处理

线型与线宽——曲线的线型和线宽

- 从一个八分象限转入下一个八分象限时要交换像素的位置，以保持划线长度近似相等。
- 在沿圆周移动时调整画每根划线的像素数目以保证划线长度近似相等。
- 改进：可以采用沿等角弧画像素代替用等长区间的像素模板来生成等长划线。

线型与线宽——曲线的线型和线宽

线宽:

□ 线刷子: 经过曲线斜率为1和-1处, 必须切换刷子。

曲线接近水平与垂直的地方, 线条更粗。

□ 方刷子: 接近水平垂直的地方, 线条更细

要显示一致的曲线宽度可通过旋转刷子方向以使其在沿曲线移动时与斜率方向一致

□ 圆弧刷子

□ 采用填充的办法。

区域填充属性

□ 区域填充属性选择包括颜色、图案和透明度。

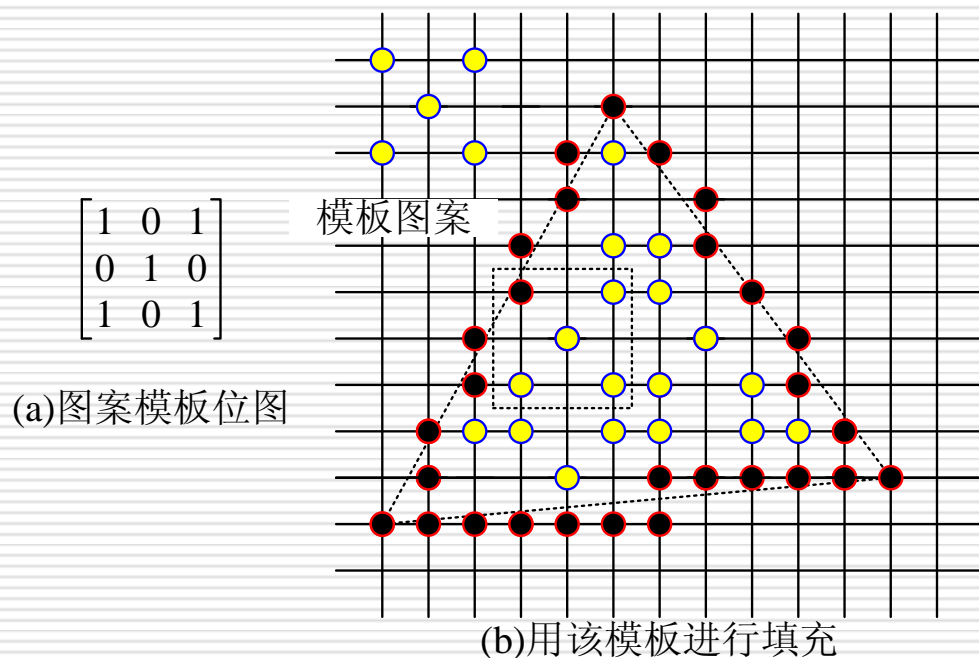


图3.30 利用图案模板进行三角形的填充

根据图案和透明度属性来填充平面区域的基本思想是：

- 首先用模板定义各种图案。
- 然后，修改填充的扫描转换算法：在确定了区域内一像素之后，不是马上往该像素填色而是先查询模板位图的对应位置。若是以透明方式填充图案，则当模板位图的对应位置为1时，用前景色写像素，否则，不改变该像素的值。若是以不透明方式填充图案，则视模板位图对应位置为1或0来决定是用前景色还是背景色去写像素。

区域填充属性

□ 确定区域与模板之间的位置关系（对齐方式）

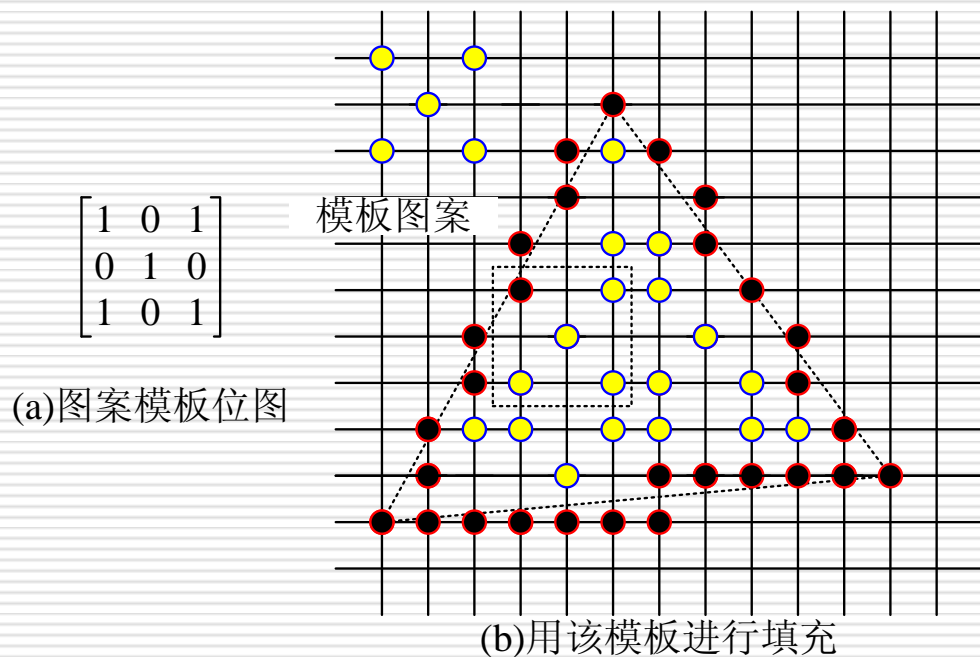


图3.31 利用图案模板进行三角形的填充

3.4 反走样

□ 用离散量表示连续量引起的失真，就叫做走样（Aliasing）。



图3.32 绘制直线时的走样现象

反走样

产生原因:

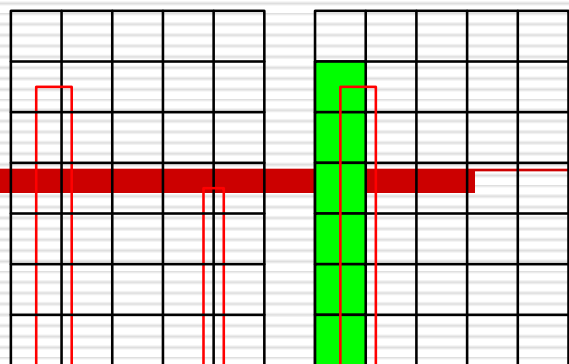
数学意义上的图形是由无限多个连续的、面积为零的点构成；但在光栅显示器上，用有限多个离散的，具有一定面积的像素来近似地表示他们。

反走样

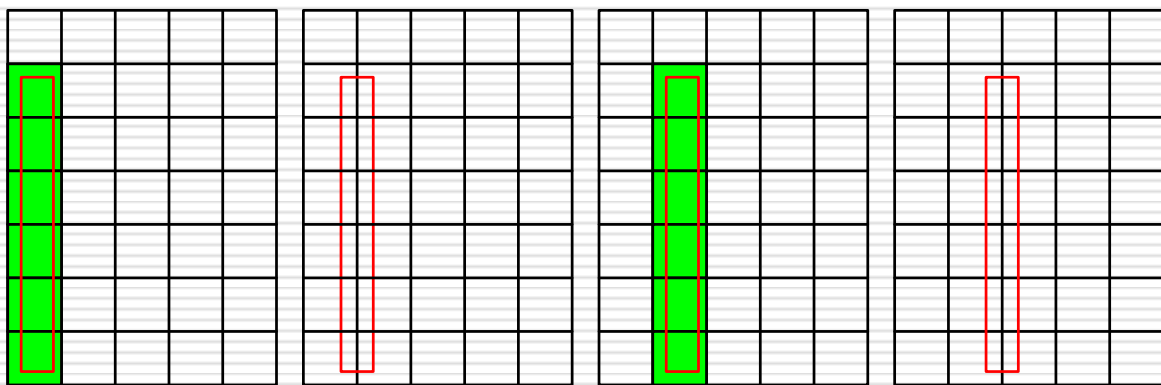
走样现象:

- 一是光栅图形产生的阶梯形。
- 一是图形中包含相对微小的物体时，这些物体在静态图形中容易被丢弃或忽略，在动画序列中时隐时现，产生闪烁。

例子



(a)需显示的矩形 (b)显示结果



(a)显示 (b)不显示 (c)显示 (d)不显示

图3.33 丢失细节与运动图形的闪烁

□ 用于减少或消除这种效果的技术，称为反走样（**antialiasing**，图形保真）。

□ 一种简单方法：

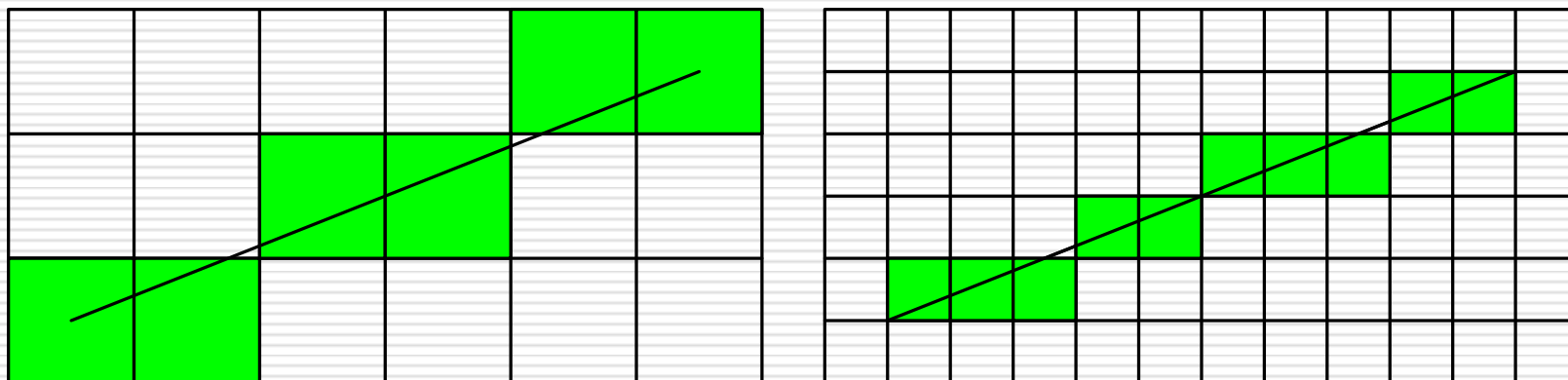


图3.34 分辨率提高一倍，阶梯程度减小一倍

□ 过取样（**supersampling**），或后滤波

□ 区域取样（**area sampling**），或前滤波

反走样——过取样 (super sampling)

- **过取样**: 在高于显示分辨率的较高分辨率下用点取样方法计算, 然后对几个象素的属性进行平均得到较低分辨率下的象素属性。

简单过取样

在x, y方向把分辨率都提高一倍, 使每个象素对应4个子象素, 然后扫描转换求得各子象素的颜色亮度, 再对4个象素的颜色亮度进行平均, 得到较低分辨率下的象素颜色亮度。

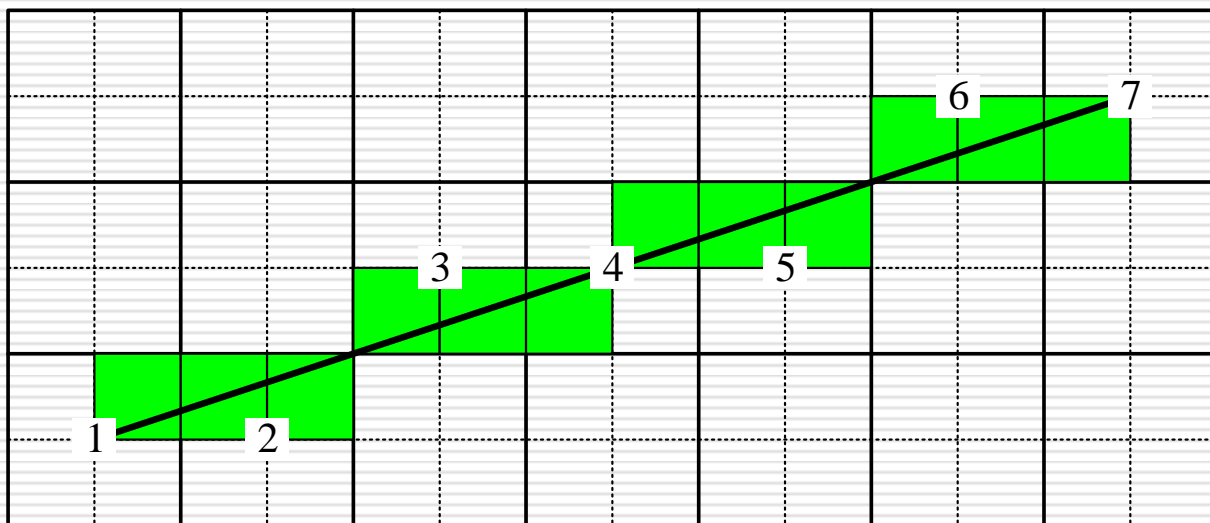
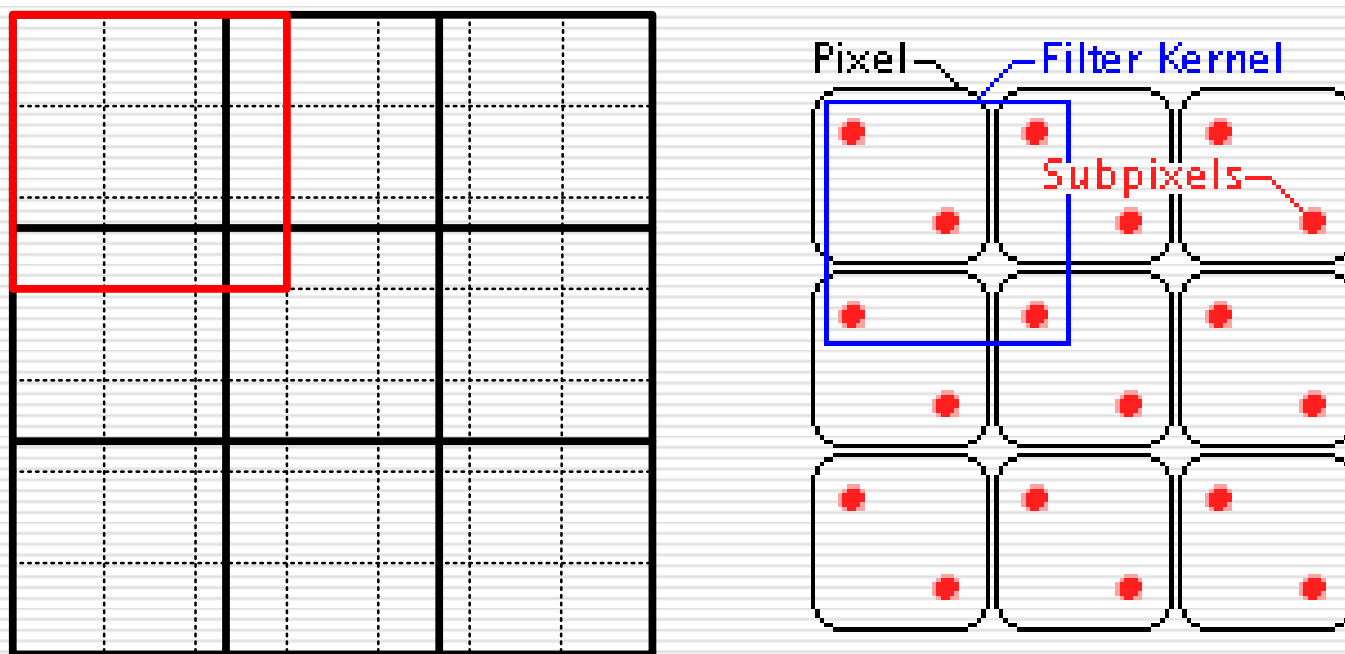


图3.35 简单的过取样方式

□ **重叠过取样**: 为了得到更好的效果, 在对一个像素点进行着色处理时, 不仅仅只对其本身的子像素进行采样, 同时对其周围的多个像素的子像素进行采样, 来计算该点的颜色属性。



□ 基于加权模板的过取样：前面在确定像素的亮度时，仅仅是对所有子像素的亮度进行简单的平均。更常见的做法是给接近像素中心的子像素赋予较大的权值，即对所有子像素的亮度进行加权平均。

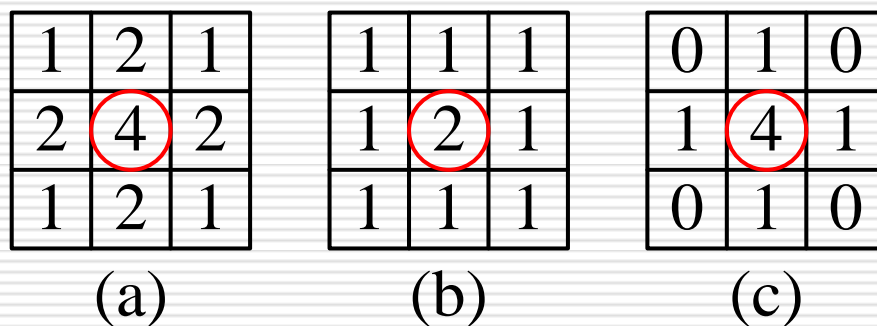


图3.37 常用的加权模板

反走样——简单的区域取样

- 在整个像素区域内进行采样，这种技术称为区域取样。又由于像素的亮度是作为一个整体被确定的，不需要划分子像素，故也被称为前置滤波。

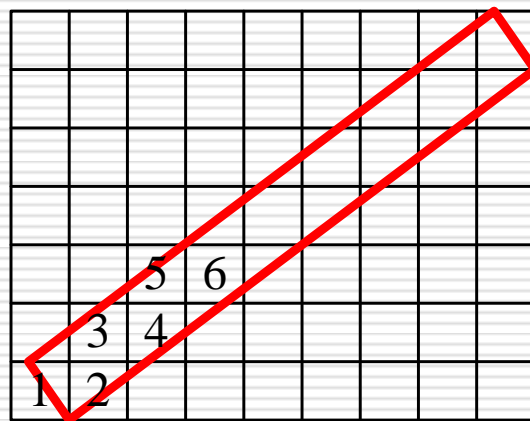


图3.38 有宽度的直线段

反走样——简单的区域取样

如何计算直线段与像素相交区域的面积？

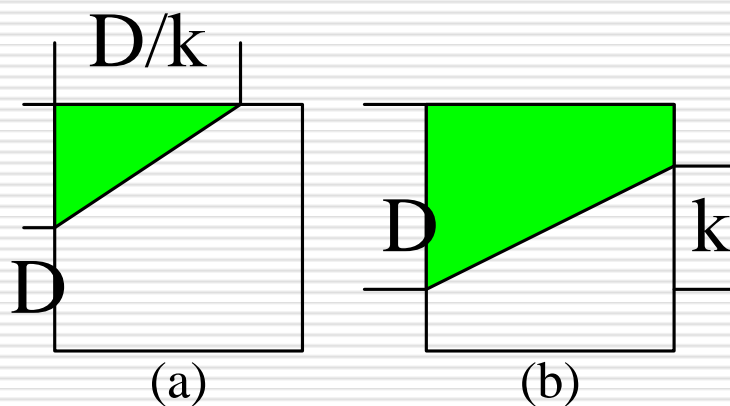


图3.39 重叠区域面积的计算

反走样——简单的区域取样

- 可以利用一种求相交区域的近似面积的离散计算方法：
 - (1)将屏幕像素分割成 n 个更小的子像素,
 - (2)计算中心落在直线段内的子像素的个数 m ,
 - (3) m/n 为线段与像素相交区域面积的近似值。
- 直线段对一个像素亮度的贡献与两者重叠区域的面积成正比。
- 相同面积的重叠区域对像素的贡献相同。

反走样——加权区域取样

- 过取样中，我们对所有子像素的亮度进行简单平均或加权平均来确定像素的亮度。
- 在区域取样中，我们使用覆盖像素的连续的加权函数（**Weighting Function**）或滤波函数（**Filtering Function**）来确定像素的亮度。

加权区域取样原理

加权函数 $W(x,y)$ 是定义在二维显示平面上的函数。对于位置为 (x,y) 的小区域 dA 来说，函数值 $W(x,y)$ （也称为在 (x,y) 处的高度）表示小区域 dA 的权值。将加权函数在整个二维显示图形上积分，得到具有一定体积的滤波器（**Filter**），该滤波器的体积为1。将加权函数在显示图形上进行积分，得到滤波器的一个子体，该子体的体积介于0到1之间。用它来表示像素的亮度。

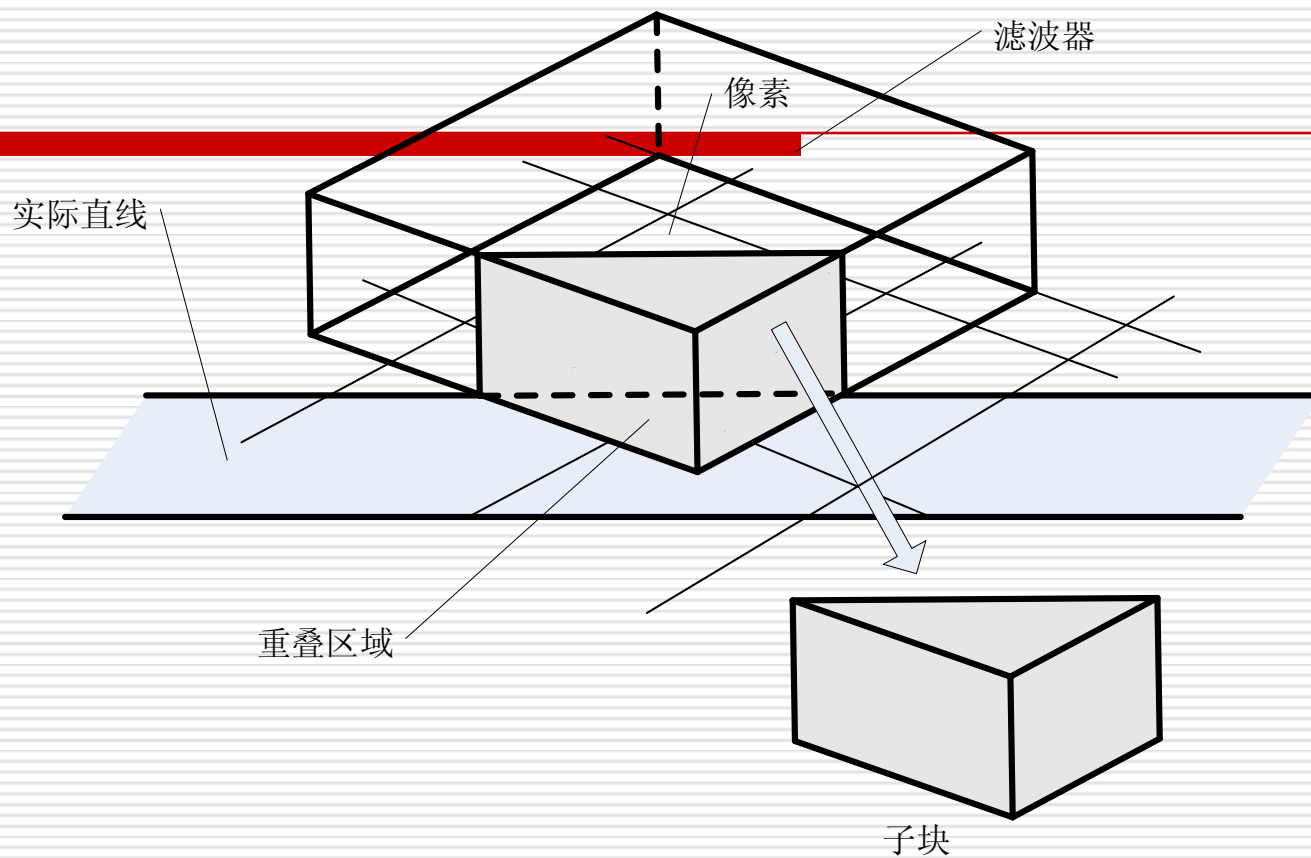


图3.40 盒式滤波器的加权区域取样

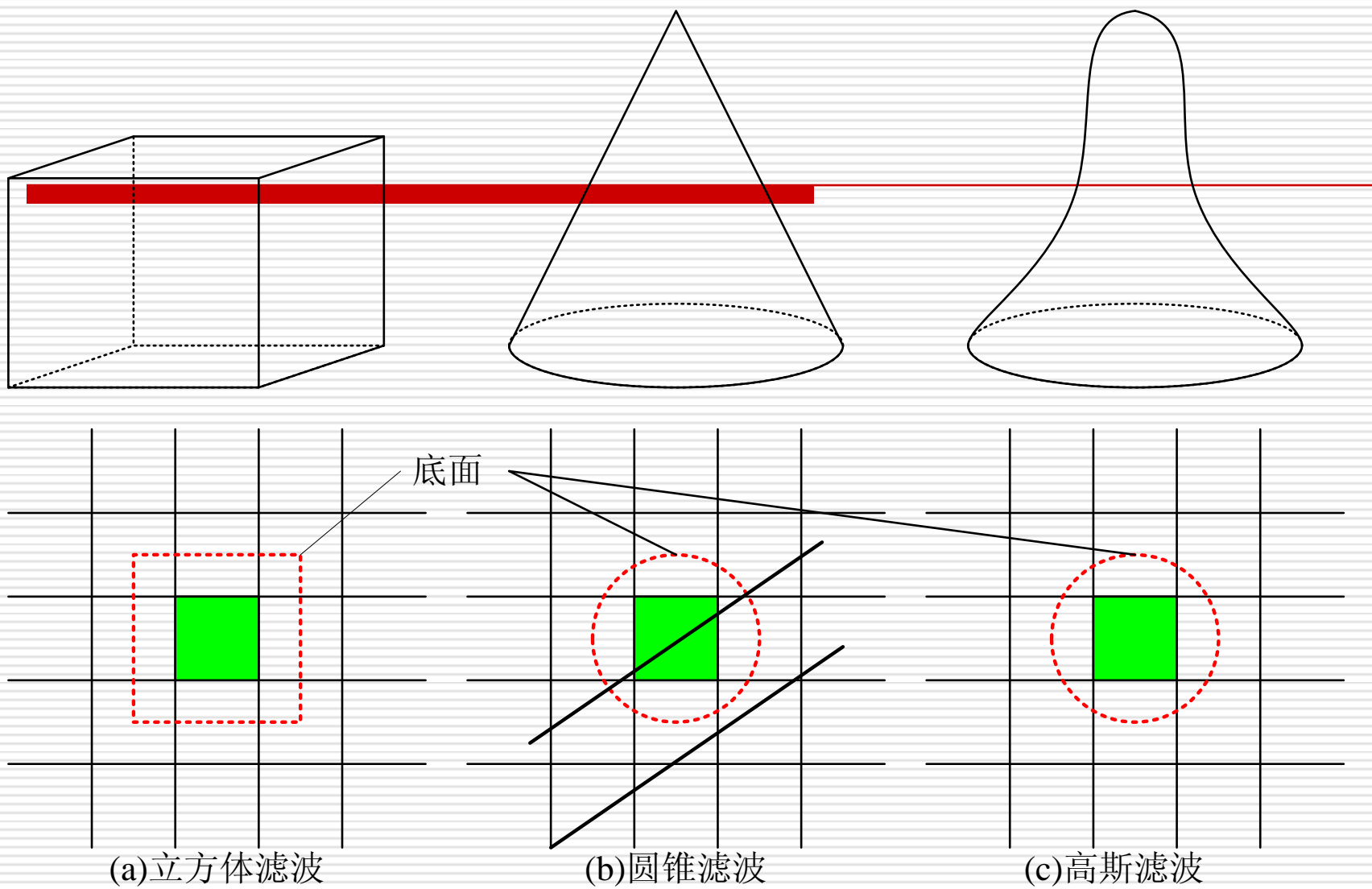


图3.41 常用的滤波函数

反走样——加权区域取样

特点:

- 接近理想直线的象素将被分配更多的灰度值;
- 相邻两个象素的滤波器相交, 有利于缩小直线条上相邻象素的灰度差。

3.5 在OpenGL中绘图

- 点的绘制
- 直线的绘制
- 多边形面的绘制
- **OpenGL中的字符函数**
- **OpenGL中的反走样**

点的绘制

□ 点的绘制

```
glBegin(GL_POINTS);  
    glVertex3f(0.0f, 0.0f, 0.0f);  
    glVertex3f(10.0f, 0.0f, 0.0f);  
glEnd();
```

□ 点的属性（大小）

```
void glPointSize(GLfloat size);
```

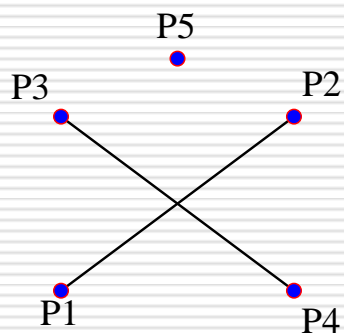
直线的绘制

□ 直线的绘制模式

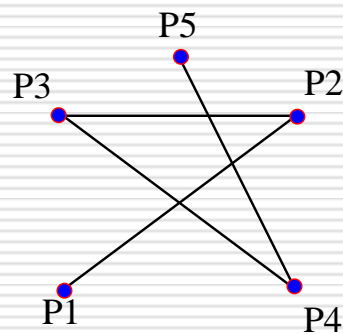
■ GL_LINES

■ GL_LINE_STRIP

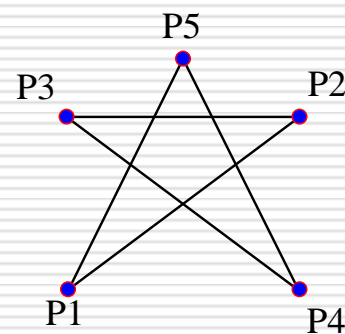
■ GL_LINE_LOOP



(a) GL_LINES画线模式



(b) GL_LINE_LOOP画线模式



(c) GL_LINE_STRIP画线模式

图3.42 OpenGL画线模式

直线的绘制

□ 直线的属性

■ 线宽

void glLineWidth(GLfloat width)

■ 线型

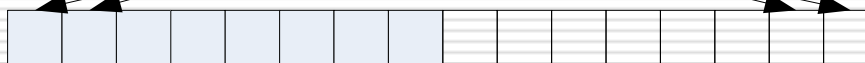
glEnable(GL_LINE_STIPPLE);
glLineStipple(GLint factor, GLushort pattern);

直线的绘制

模式：0X00FF = 255

二进制表示：0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

画线模式：



线：



图3.43 画线模式用于构造线段

多边形面的绘制

□ 三角形面的绘制

- **GL_TRIANGLES**

- **GL_TRIANGLE_STRIP**

- **GL_TRIANGLE_FAN**

□ 四边形面的绘制

- **GL_QUADS**

- **GL_QUADS_STRIP**

□ 多边形面的绘制 (**GL_POLYGON**)

多边形面的绘制

□ 多边形面的绘制规则

- 所有多边形都必须是平面的。

- 多边形的边缘决不能相交，而且多边形必须是凸的。

□ 解决：对于非凸多边形，可以把它分割成几个凸多边形（通常是三角形），再将它绘制出来。

多边形面的绘制

- 问题：轮廓图形状状态会看到组成大表面的所有小三角形。处理OpenGL提供了一个特殊标记来处理这些边缘，称为边缘标记。

glEdgeFlag (True)

glEdgeFlag (False)

多边形面的属性

□ 多边形面的正反属性（绕法）

指定顶点时顺序和方向的组合称为“绕法”。绕法是一切多边形图元的一个重要特性。一般默认情况下，OpenGL认为逆时针绕法的多边形是正对着的。

glFrontFace(GL_CW);

多边形面的属性

□ 多边形面的颜色

- **glShadeModel(GL_FLAT)** 用指定多边形最后一个顶点时的当前颜色作为填充多边形的纯色，唯一例外是**GL_POLYGON**图元，它采用的是第一个顶点的颜色。
- **glShadeModel(GL_SMOOTH)** 从各个顶点给三角形投上光滑的阴影，为各个顶点指定的颜色之间进行插值。

多边形面的属性

□ 多边形面的显示模式

```
glPolygonMode(GLenum face,  
GLenum mode);
```

- 参数**face**用于指定多边形的哪一个面受到模式改变的影响。
- 参数**mode**用于指定新的绘图模式。

多边形面的属性

□ 多边形面的填充

多边形面既可以用纯色填充，也可以用
 32×32 的模板位图来填充。

```
void glPolygonStipple(const GLubyte  
*mask);
```

```
glEnable(GL_POLYGON_STIPPLE);
```

多边形面的属性

□ 多边形面的法向量

- 法向量是垂直于面的方向上点的向量，它确定了几何对象在空间中的方向。
- 在OpenGL中，可以为每个顶点指定法向量。

```
void glNormal3{bsidf} ( TYPE nx, TYPE  
ny, TYPE nz);
```

```
void glNormal3{bsidf}v (const TYPE* v);
```

OpenGL中的字符函数

□ GLUT位图字符

```
void glutBitmapCharacter(void *font,  
int character);
```

□ GLUT矢量字符

```
void glutStrokeCharacter(void *font,  
int character);
```


OpenGL中的反走样

□ 启用反走样

`glEnable(primitiveType);`

□ 启用OpenGL颜色混和并指定颜色混合函数

`glEnable(GL_BLEND);`

`glBlendFunc(GL_SRC_ALPHA,
GL_ONE_MINUS_SRC_ALPHA);`

OpenGL中的反走样

- 颜色混和函数用于计算两个相互重叠的对象的颜色。
- 在RGBA颜色模式（A表示透明度）中，已知源像素的颜色值为 (S_r, S_g, S_b, S_a) ，目标像素的颜色值为 (D_r, D_g, D_b, D_a) ，颜色混合后像素的颜色为：

$$(R_S.S_r + R_D.D_r, G_S.S_g + G_D.D_g, B_S.S_b + B_D.D_b, A_S.S_a + A_D.D_a)$$

OpenGL中的反走样

□ 定义混合因子

```
void glBlendFunc(GLenum srcfactor,  
GLenum destfactor);
```

表5-1 源混和因子和目标混合因子

常量	RGB混合因子	Alpha混合因子
GL_ZERO	(0, 0, 0)	0
GL_ONE	(1, 1, 1)	1
GL_SRC_COLOR	(R_S , G_S , B_S)	A_S
GL_ONE_MINUS_SRC_COLOR	$(1, 1, 1) - (R_S, G_S, B_S)$	$1 - A_S$
GL_DST_COLOR	(R_D , G_D , B_D)	A_D
GL_ONE_MINUS_DST_COLOR	$(1, 1, 1) - (R_D, G_D, B_D)$	$1 - A_D$
GL_SRC_ALPHA	(A_S , A_S , A_S)	A_S
GL_ONE_MINUS_SRC_ALPHA	$(1, 1, 1) - (A_S, A_S, A_S)$	$1 - A_S$
GL_DST_ALPHA	(A_D , A_D , A_D)	A_D
GL_ONE_MINUS_DST_ALPHA	$(1, 1, 1) - (A_D, A_D, A_D)$	$1 - A_D$

第六章 二维变换及二维观察

- 如何对二维图形进行方向、尺寸和形状方面的变换。
- 如何进行二维观察。

二维变换及二维观察

- 基本几何变换与基本概念
- 二维图形几何变换的计算
- 复合变换
- 变换的性质

6.2 基本几何变换

- **图形的几何变换**是指对图形的几何信息经过平移、比例、旋转等变换后产生新的图形，是图形在方向、尺寸和形状方面的变换。(位置、尺寸-形状、方向)
- **基本几何变换**都是相对于坐标原点和坐标轴进行的几何变换。

基本几何变换——平移变换

□ 平移是指将 p 点沿直线路径从一个坐标位置移到另一个坐标位置的重定位过程。

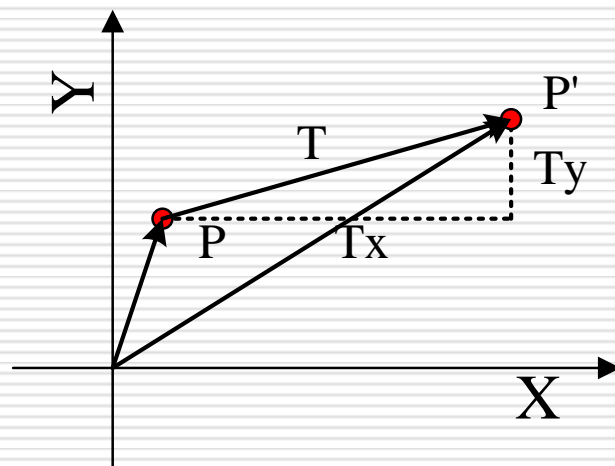


图6-1 平移变换

基本几何变换——平移变换

推导:

$$x' = x + T_x$$

$$y' = y + T_y$$

矩阵形式:

$$\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} + \begin{bmatrix} T_x & T_y \end{bmatrix}$$

T_x , T_y 称为平移矢量。

基本几何变换——比例变换

推导:

$$x' = S_x \cdot x$$

$$y' = S_y \cdot y$$

矩阵形式:

$$\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \cdot \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix}$$

基本几何变换——比例变换

□ 比例变换是指对 p 点相对于坐标原点沿 x 方向放缩 S_x 倍，沿 y 方向放缩 S_y 倍。其中 S_x 和 S_y 称为比例系数。

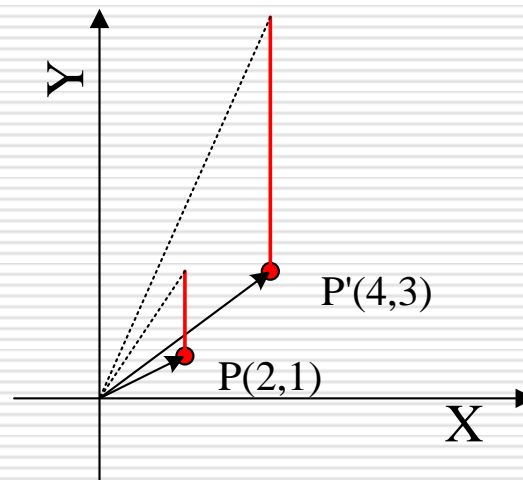
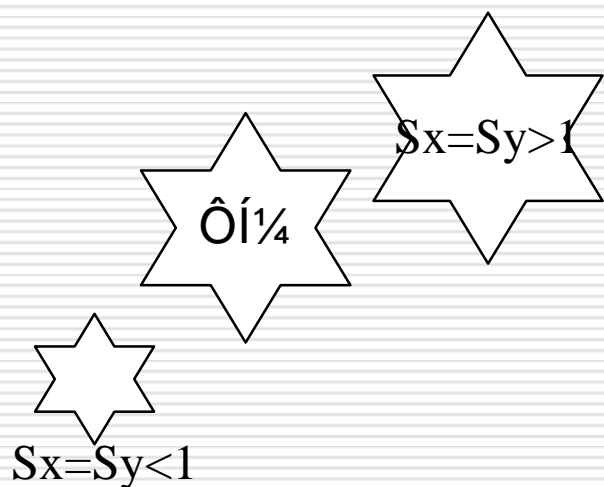
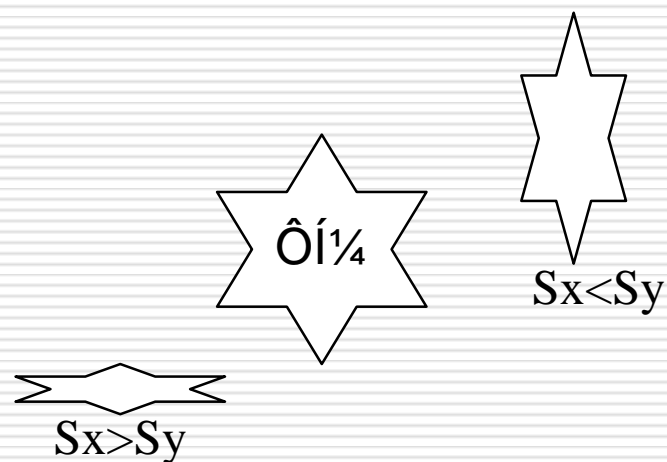


图6-2 比例变换($S_x=2, S_y=3$)

基本几何变换——比例变换



(a) $S_x = S_y$ 比例



(b) $S_x \neq S_y$ 比例

图6-3 比例变换

基本几何变换——旋转变换

- 二维旋转是指将 p 点绕坐标原点转动某个角度（逆时针为正，顺时针为负）得到新的点 p' 的重定位过程。

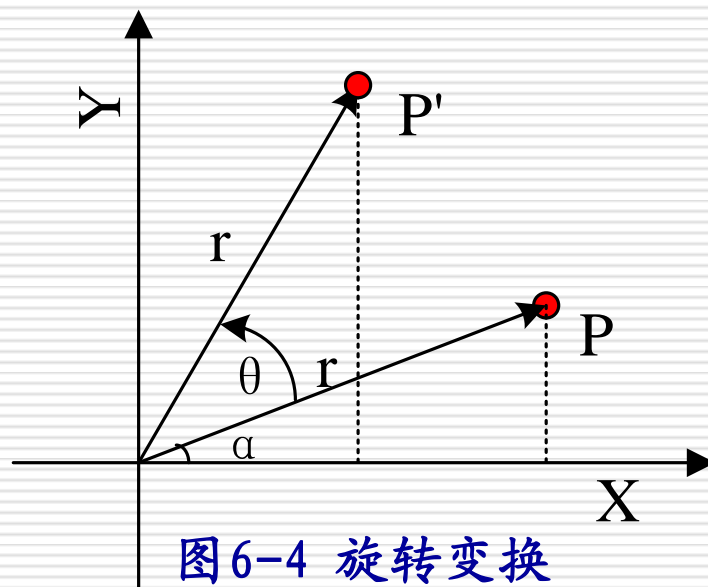


图6-4 旋转变换

基本几何变换——旋转变换

□ 推导：（极坐标）

$$x = r \cos \alpha \quad y = r \sin \alpha$$

$$x' = r \cos(\alpha + \theta) = x \cos \theta - y \sin \theta$$

$$y' = r \sin(\alpha + \theta) = x \sin \theta + y \cos \theta$$

□ 矩阵：逆时针旋转 θ 角

$$\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \cdot \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix}$$

基本几何变换

□ 平移、缩放、旋转变换的矩阵表示:

$$P' = P + T$$

$$P' = P \cdot S \quad \longrightarrow \quad P' = P \cdot T_1 + T_2$$

$$P' = P \cdot R$$

□ 图形通常要进行一系列基本几何变换，希望能够把二维变换统一表示为矩阵的乘法。

基本几何变换——规范化齐次坐标

□ 齐次坐标表示就是用 $n+1$ 维向量表示一个 n 维向量。

$$(x, y) \Leftarrow (xh, yh, h) \quad h \neq 0$$

□ 规范化齐次坐标表示就是 $h=1$ 的齐次坐标表示。

$$(x, y) \Leftarrow (x, y, 1)$$

基本几何变换

平移: $[x' \quad y'] = [x \quad y] + [T_x \quad T_y]$



$$[x' \quad y' \quad 1] = [x \quad y \quad 1] \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{bmatrix}$$

基本几何变换

比例:

$$\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \cdot \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix}$$



$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \cdot \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

基本几何变换

整体比例变换:

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & S \end{bmatrix}$$

基本几何变换

旋转变换:

$$\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} + \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix}$$

\Downarrow

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

基本几何变换——二维变换矩阵

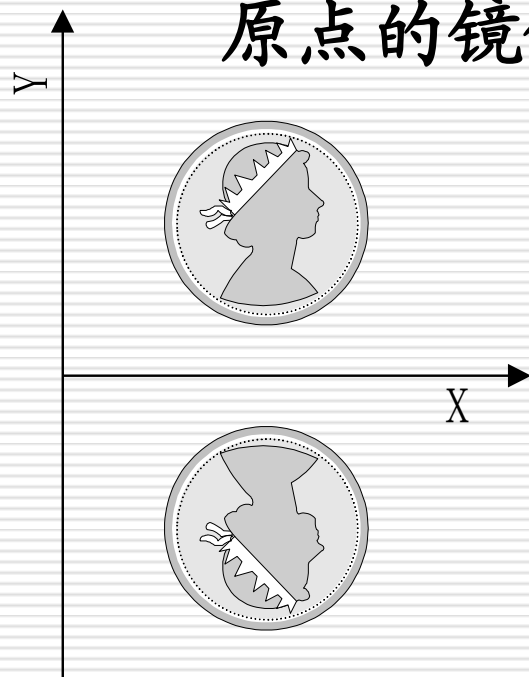
$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \cdot T_{2D} = \begin{bmatrix} x & y & 1 \end{bmatrix} \cdot \begin{bmatrix} a & b & p \\ c & d & q \\ l & m & s \end{bmatrix}$$

$$x' = \frac{ax + cy + l}{px + qy + s}$$

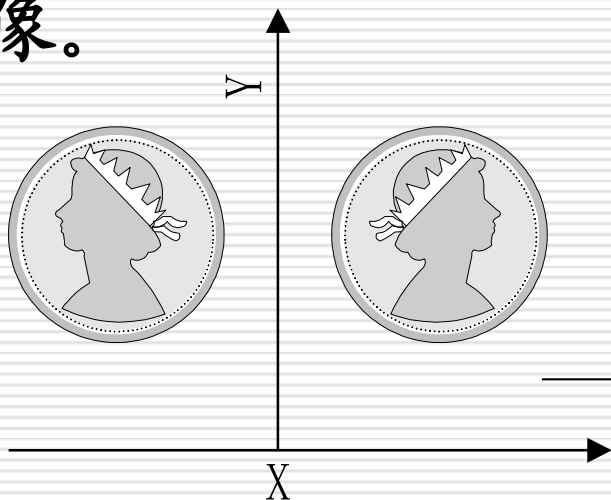
$$y' = \frac{bx + dy + m}{px + qy + s}$$

基本几何变换——对称变换

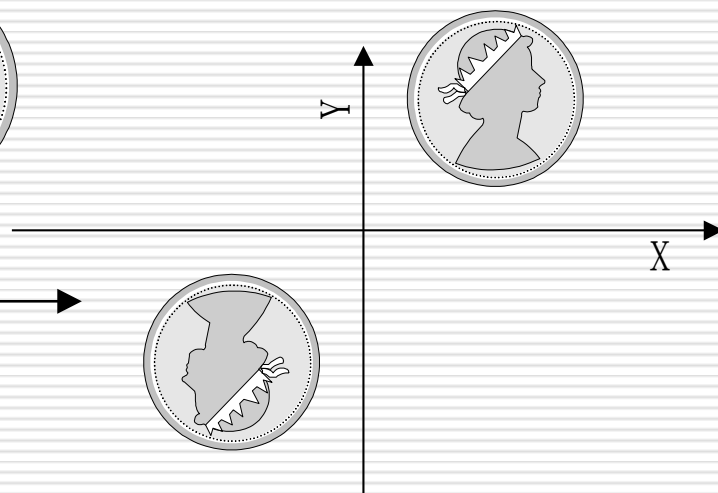
□ 对称变换后的图形是原图形关于某一轴线或原点的镜像。



(a) 关于x轴对称

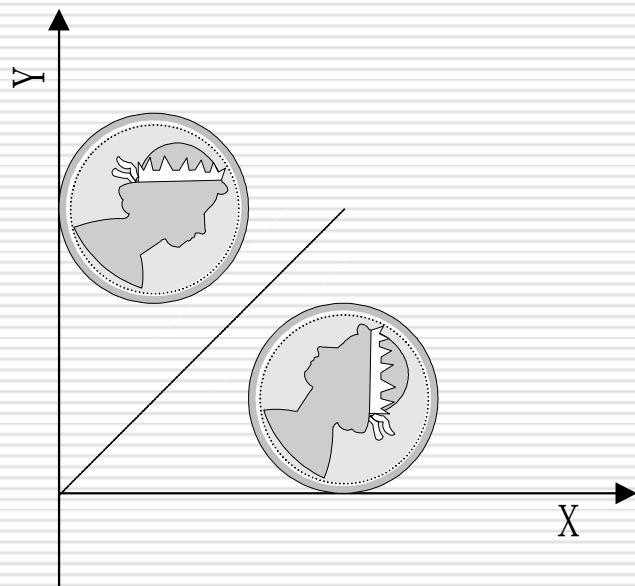


(b) 关于y轴对称

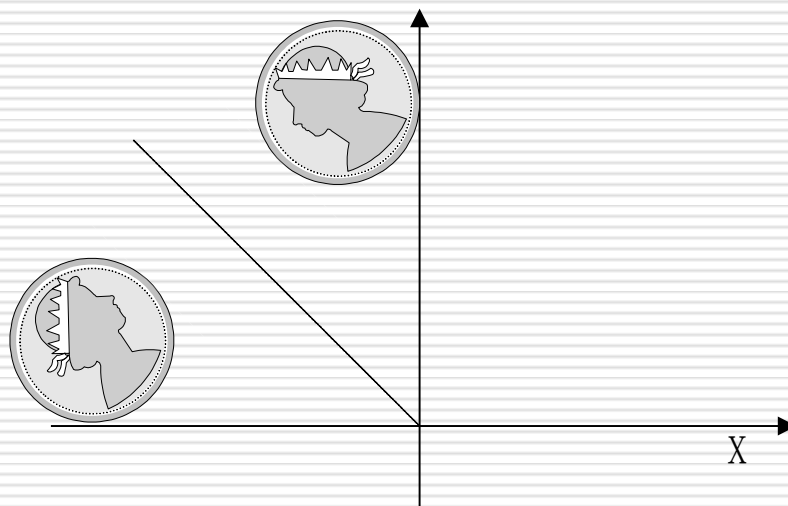


(c) 关于原点对称

基本几何变换——对称变换



(d) 关于 $x=y$ 对称



(e) 关于 $x=-y$ 对称

基本几何变换——对称变换

(1)关于x轴对称

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

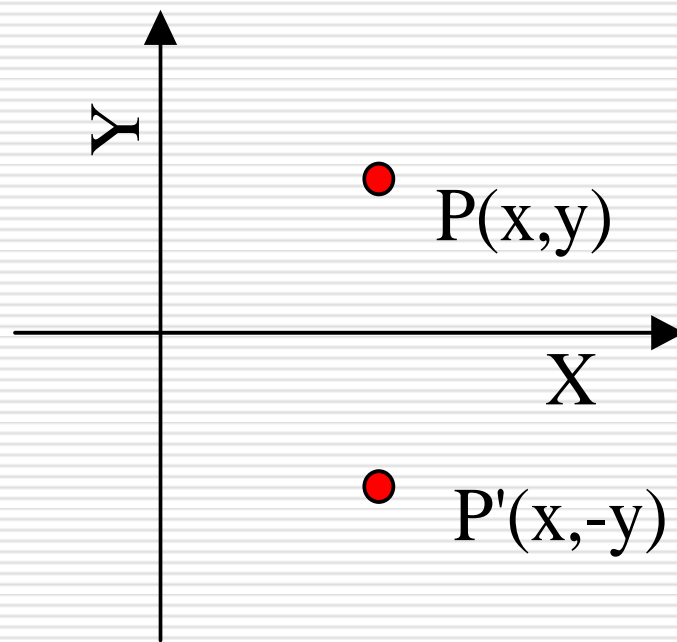


图6-5 关于x轴对称

基本几何变换——对称变换

(2)关于y轴对称

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

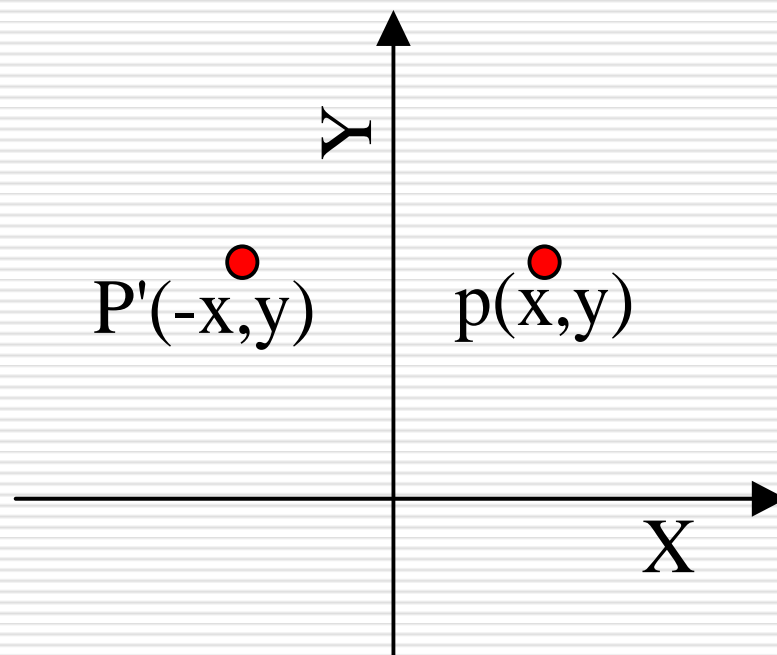


图6-6 关于y轴对称

基本几何变换——对称变换

(3)关于原点对称

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

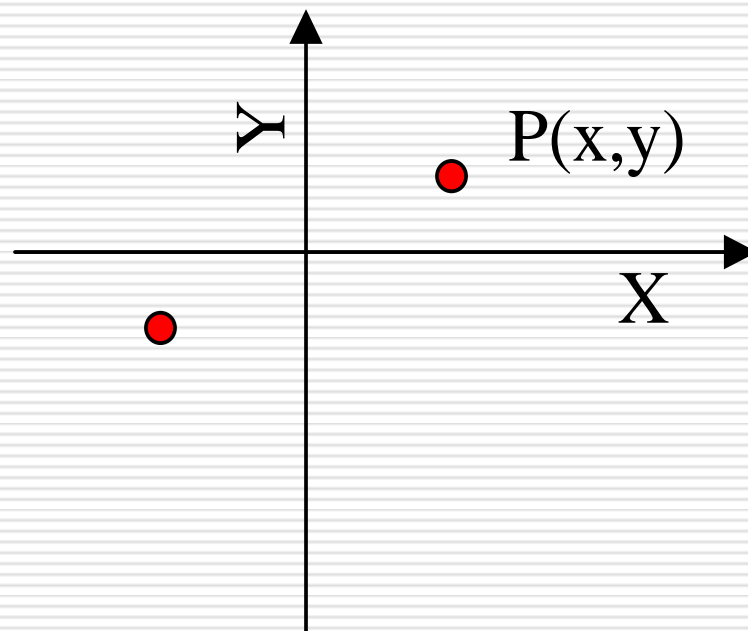


图6-7 关于原点对称

基本几何变换——对称变换

(4)关于 $y=x$ 轴对称

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

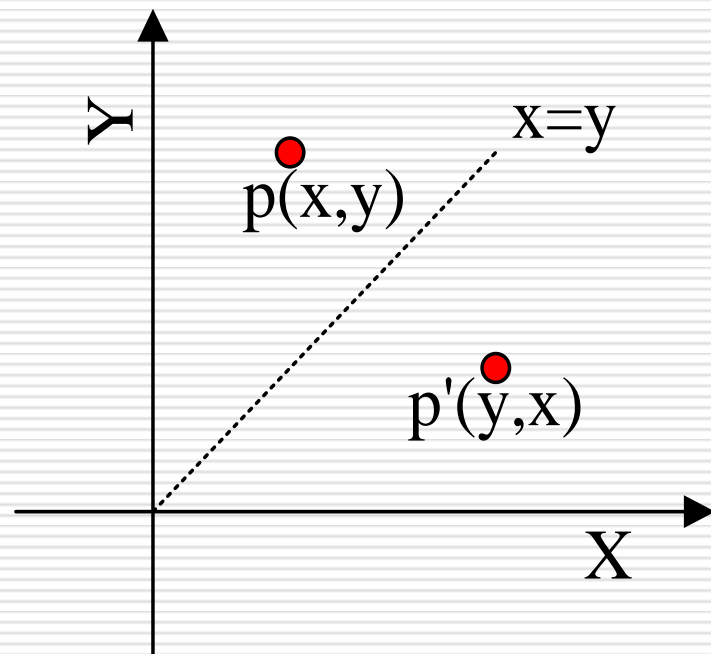


图6-8 关于 $x=y$ 对称

基本几何变换——对称变换

(5)关于 $y=-x$ 轴对称

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

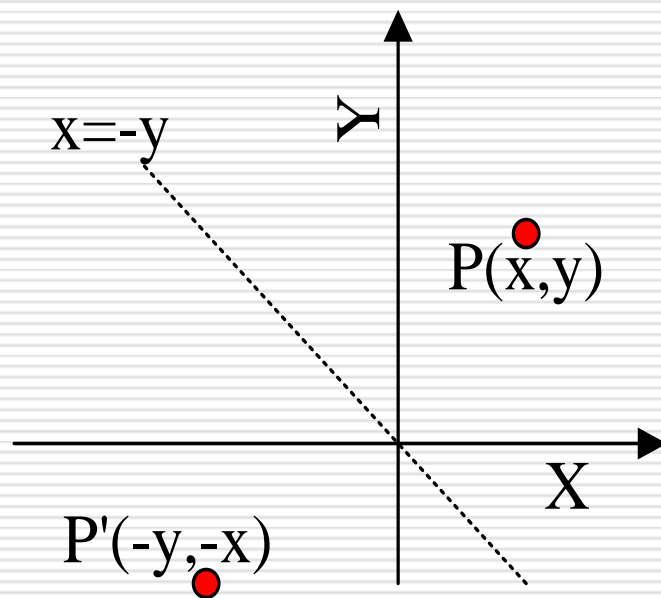


图6-9 关于 $x = -y$ 对称

基本几何变换——错切变换

□ 错切变换，也称为剪切、错位变换，用于产生弹性物体的变形处理。

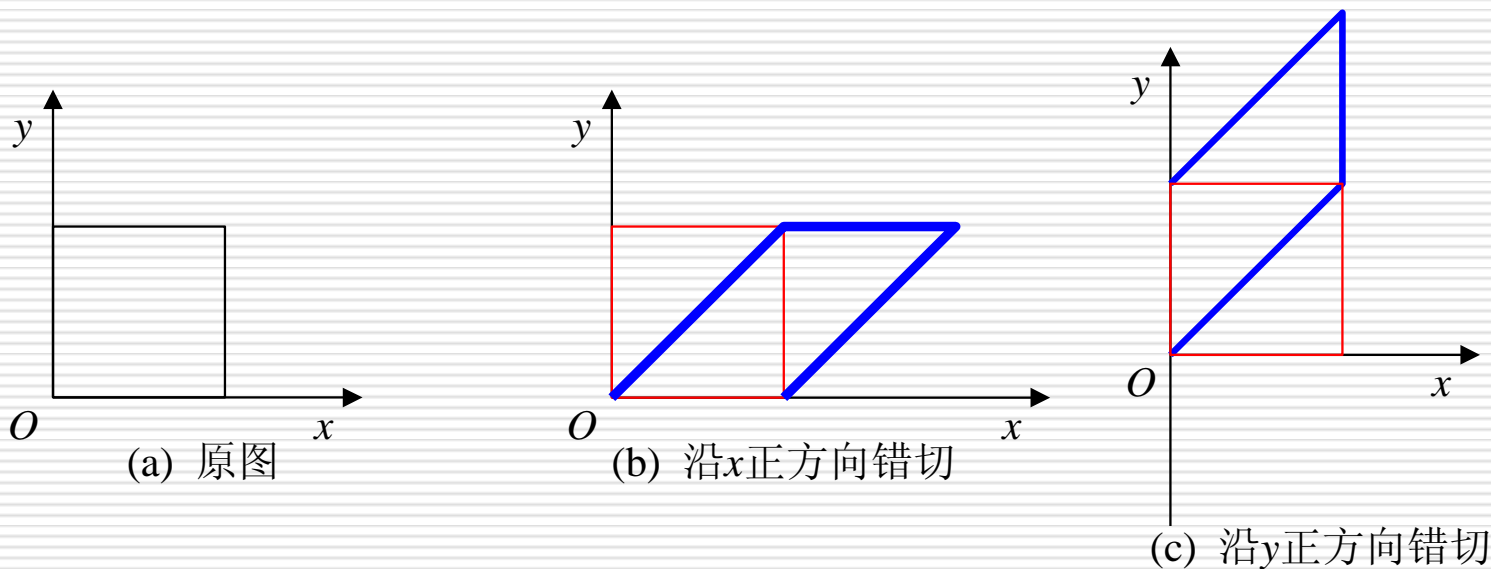


图6-10 错切变换

基本几何变换——错切变换

其变换矩阵为：

$$\begin{bmatrix} 1 & b & 0 \\ c & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(1)沿x方向错切

(2)沿y方向错切

(3)两个方向错切

二维图形几何变换的计算

几何变换均可表示成 $P'=P*T$ 的形式。

1. 点的变换

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \cdot \begin{bmatrix} a & b & p \\ c & d & q \\ l & m & r \end{bmatrix}$$

二维图形几何变换的计算

2. 直线的变换

$$\begin{bmatrix} x_1' & y_1' & 1 \\ x_2' & y_2' & 1 \end{bmatrix} = \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{bmatrix} \cdot \begin{bmatrix} a & b & p \\ c & d & q \\ l & m & r \end{bmatrix}$$

二维图形几何变换的计算

3. 多边形的变换

$$\begin{bmatrix} x'_1 & y'_1 & 1 \\ x'_2 & y'_2 & 1 \\ x'_3 & y'_3 & 1 \\ \dots & \dots & \dots \\ x'_n & y'_n & 1 \end{bmatrix} = \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \\ \dots & \dots & \dots \\ x_n & y_n & 1 \end{bmatrix} \cdot \begin{bmatrix} a & b & p \\ c & d & q \\ l & m & r \end{bmatrix}$$

复合变换

- 图形作一次以上的几何变换，变换结果是每次变换矩阵的乘积。
- 任何一复杂的几何变换都可以看作基本几何变换的组合形式。
- 复合变换具有形式：

$$\begin{aligned} P' &= P \cdot T = P \cdot (T_1 \cdot T_2 \cdot T_3 \cdots T_n) \\ &= P \cdot T_1 \cdot T_2 \cdot T_3 \cdots T_n \quad (n > 1) \end{aligned}$$

复合变换——二维复合平移

$$\begin{aligned} T_t &= T_{t1} \cdot T_{t2} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_{x1} & T_{y1} & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_{x2} & T_{y2} & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_{x1} + T_{x2} & T_{y1} + T_{y2} & 1 \end{bmatrix} \end{aligned}$$

复合变换——二维复合比例

$$\begin{aligned} T_s = T_{s1} \cdot T_{s2} &= \begin{bmatrix} S_{x1} & 0 & 0 \\ 0 & S_{y1} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} S_{x2} & 0 & 0 \\ 0 & S_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} S_{x1} \cdot S_{x2} & 0 & 0 \\ 0 & S_{y1} \cdot S_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

复合变换——二维复合旋转

$$\begin{aligned} T_r = T_{r1} \cdot T_{r2} &= \begin{bmatrix} \cos \theta_1 & \sin \theta_1 & 0 \\ -\sin \theta_1 & \cos \theta_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos \theta_2 & \sin \theta_2 & 0 \\ -\sin \theta_2 & \cos \theta_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos(\theta_1 + \theta_2) & \sin(\theta_1 + \theta_2) & 0 \\ -\sin(\theta_1 + \theta_2) & \cos(\theta_1 + \theta_2) & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

$$R = R_{(\theta_1)} \bullet R_{(\theta_2)} = R(\theta_1 + \theta_2)$$

复合变换

$$\begin{aligned} R &= \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & 0 \\ 0 & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & \operatorname{tg}\theta & 0 \\ -\operatorname{tg}\theta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & \operatorname{tg}\theta & 0 \\ -\operatorname{tg}\theta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & 0 & 0 \\ 0 & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

相对任一参考点的二维几何变换

□ 相对某个参考点 (x_F, y_F) 作二维几何变换，其变换过程为：

(1) 平移；

(2) 针对原点进行二维几何变换；

(3) 反平移。

相对任一参考点的二维几何变换

例1. 相对点 (x_F, y_F) 的旋转变换

$$\begin{aligned} T_{RF} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_F & -y_F & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_F & y_F & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ x_F - x_F \cos \theta + y_F \sin \theta & y_F - y_F \cos \theta - x_F \sin \theta & 1 \end{bmatrix} \end{aligned}$$

相对任意方向的二维几何变换

□ 相对任意方向作二维几何变换，其变换的过程是：

(1) 旋转变换；

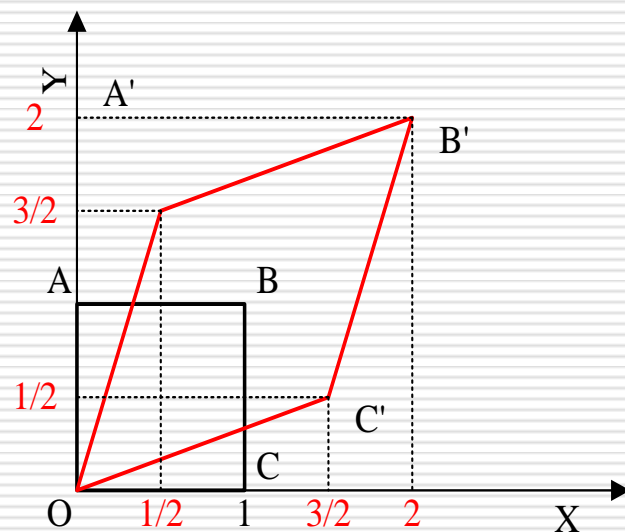
(2) 针对坐标轴进行二维几何变换；

(3) 反向旋转。

□ 例2. 相对直线 $y=x$ 的反射变换 (z dq???)

复合变换

例3. 将正方形 $ABCO$ 各点沿下图所示的 $(0,0) \rightarrow (1,1)$ 方向进行拉伸，结果为如图所示的，写出其变换矩阵和变换过程。



可能发生的变换：沿 $(0, 0)$
到 $(1, 1)$ 的比例变换

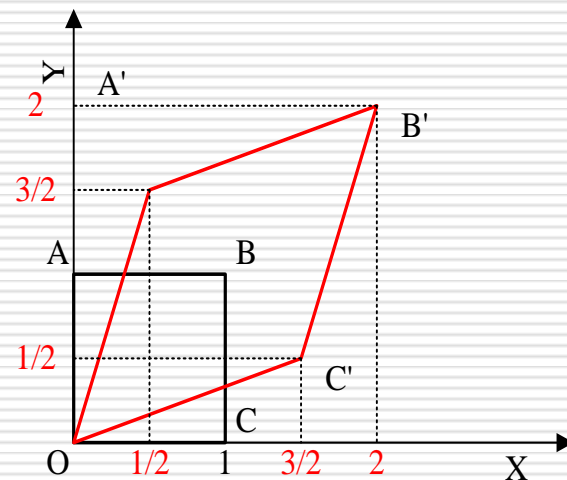


图6-11 沿固定方向拉伸

$$T = \begin{bmatrix} \cos(-45^\circ) & \sin(-45^\circ) & 0 \\ -\sin(-45^\circ) & \cos(-45^\circ) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos 45^\circ & \sin 45^\circ & 0 \\ -\sin 45^\circ & \cos 45^\circ & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$P' = P \cdot T$$

坐标系之间的变换

问题:

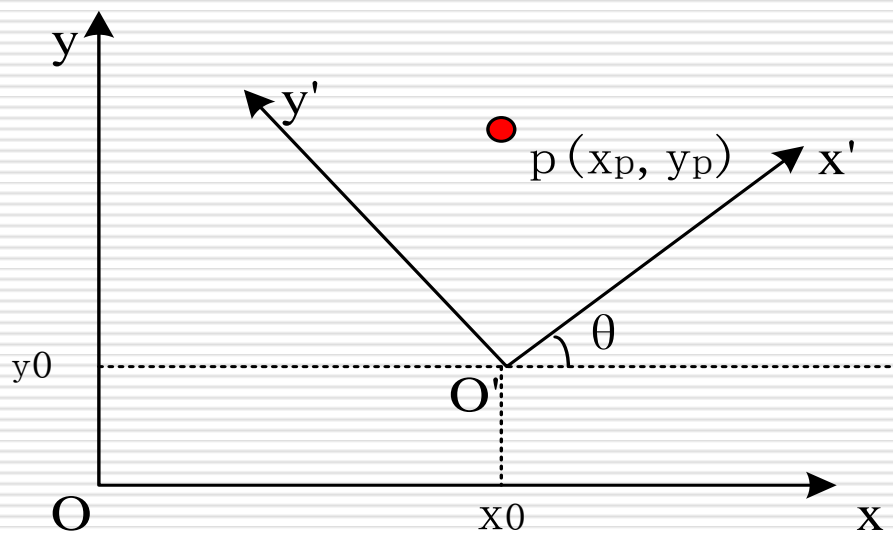


图6-12 坐标系间的变换

坐标系之间的变换

分析:

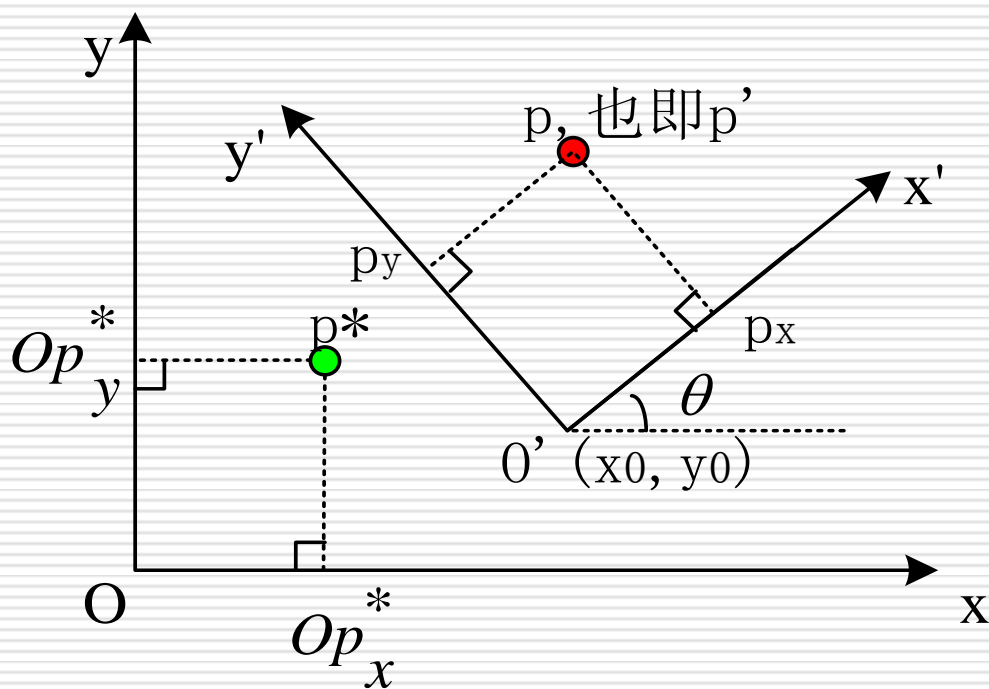


图6-13 坐标系间的变换的原理

可以分两步进行:

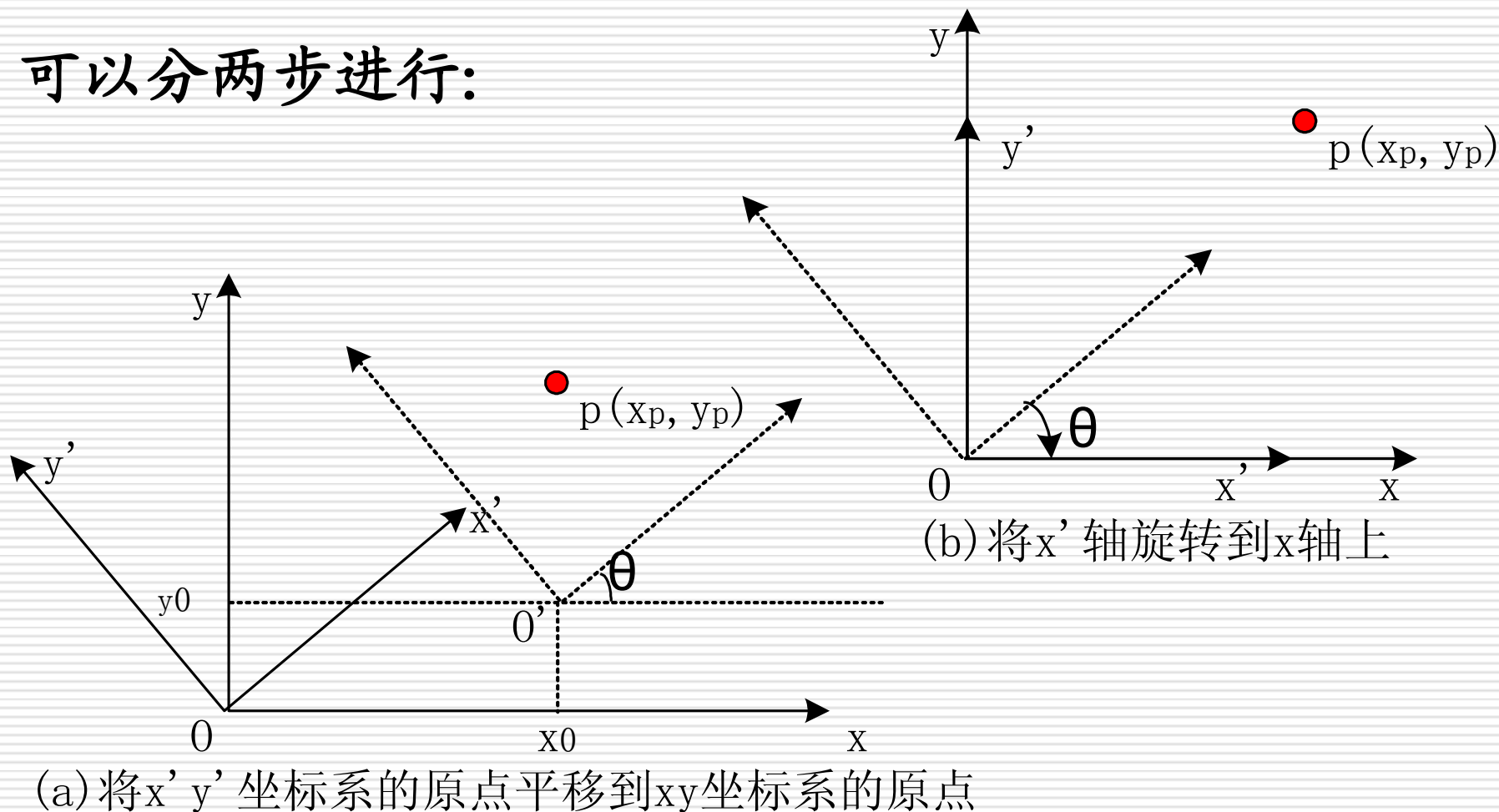


图6-14 坐标系间的变换的步骤

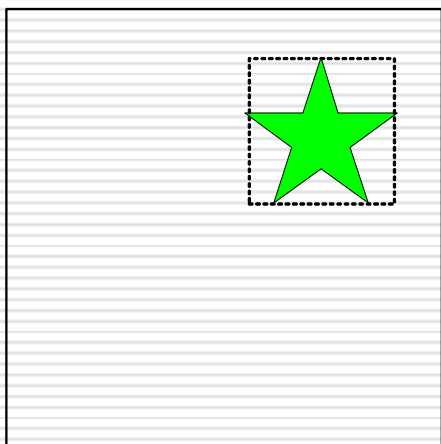
于是:

$$\begin{aligned} p' &= \begin{bmatrix} x' & y' & 1 \\ p & p & \end{bmatrix} = \begin{bmatrix} x & y & 1 \\ p & p & \end{bmatrix} \cdot T \\ &= p \cdot T = p \cdot T_t \cdot T_R \end{aligned}$$

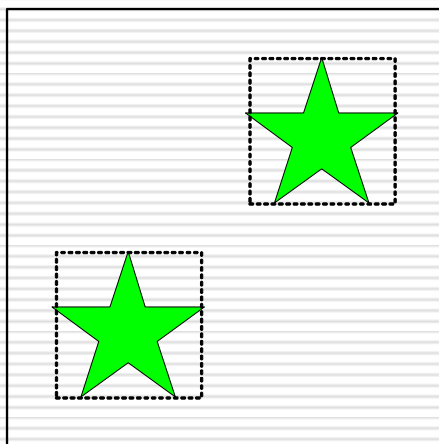
$$T = T_t \cdot T_r = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_0 & -y_0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

光栅变换

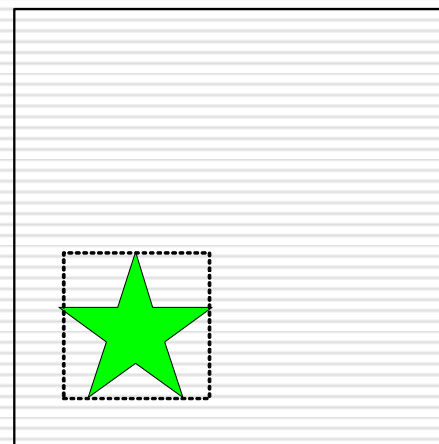
- ❑ 直接对帧缓存中象素点进行操作的变化称为光栅变换。
- ❑ 光栅平移变换:



(a) 读出象素块的内容



(b) 复制象素块的内容



(c) 擦除原象素块的内容

光栅变换

□ 90° 、 180° 和 270° 的光栅旋转变换:

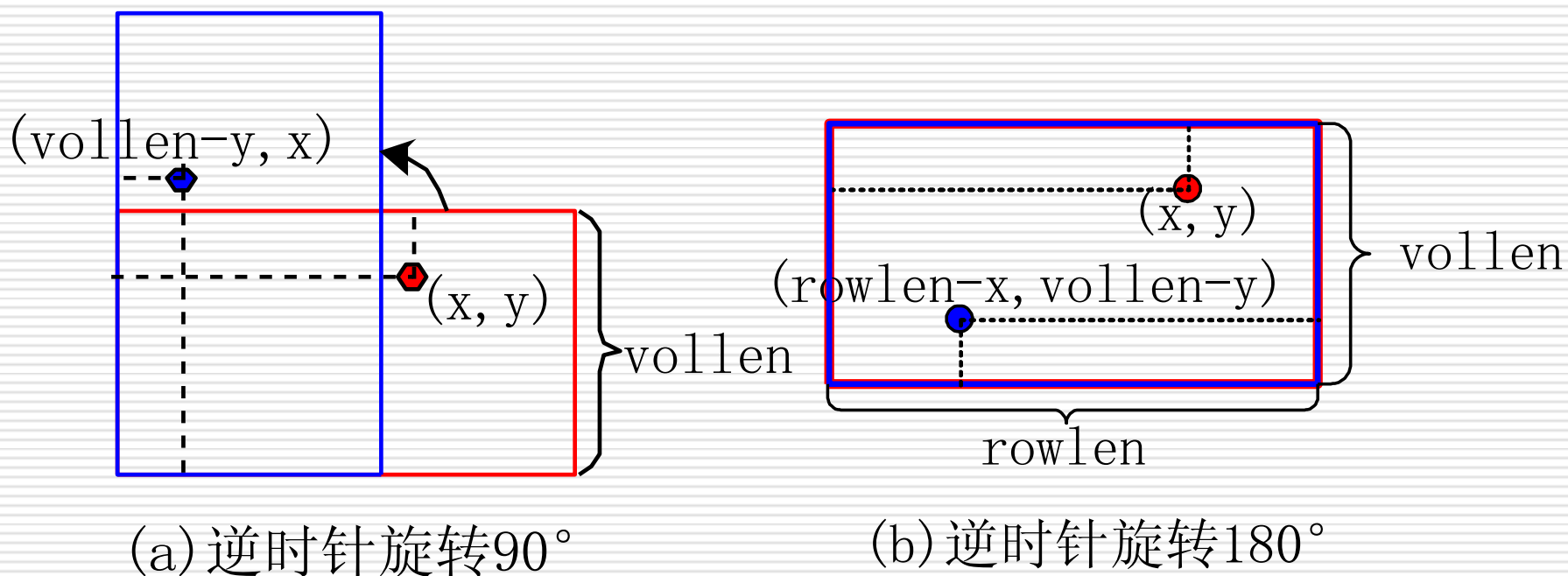


图6-15 光栅旋转变换

光栅变换

□ 任意角度的光栅旋转变换:

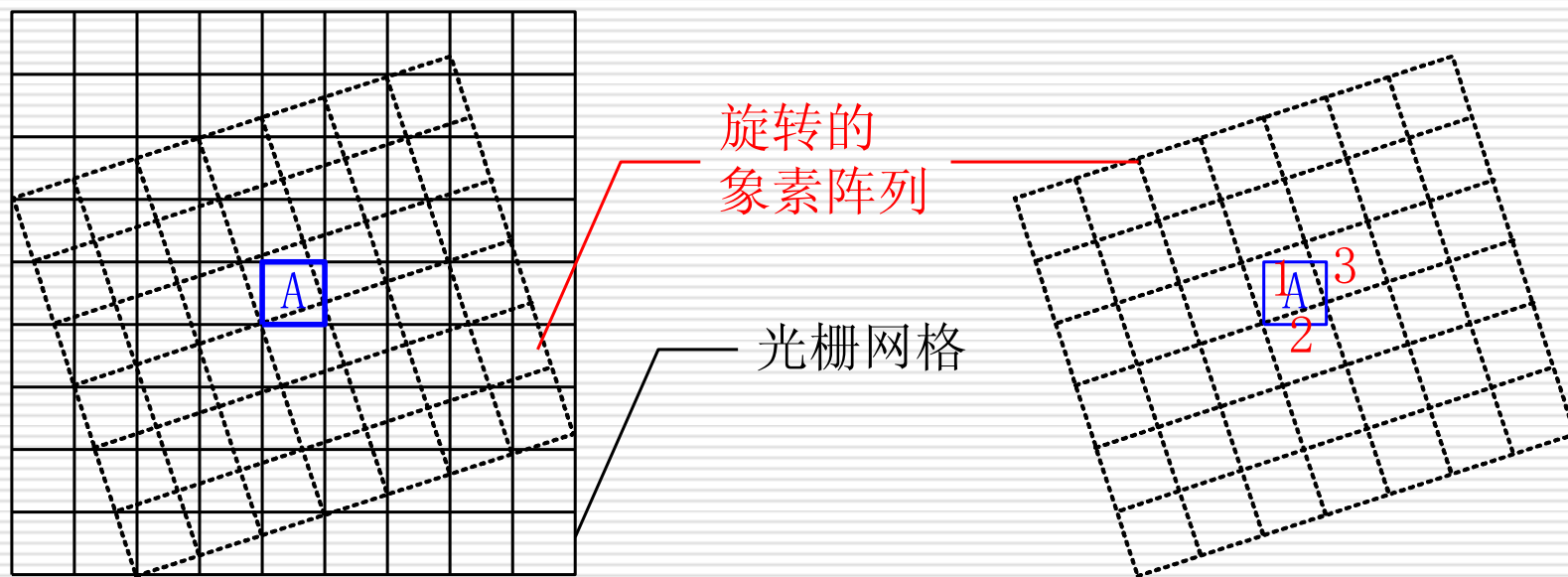


图6-16 任意角度的光栅旋转变换

光栅变换

□ 光栅比例变换：进行区域的映射处理。

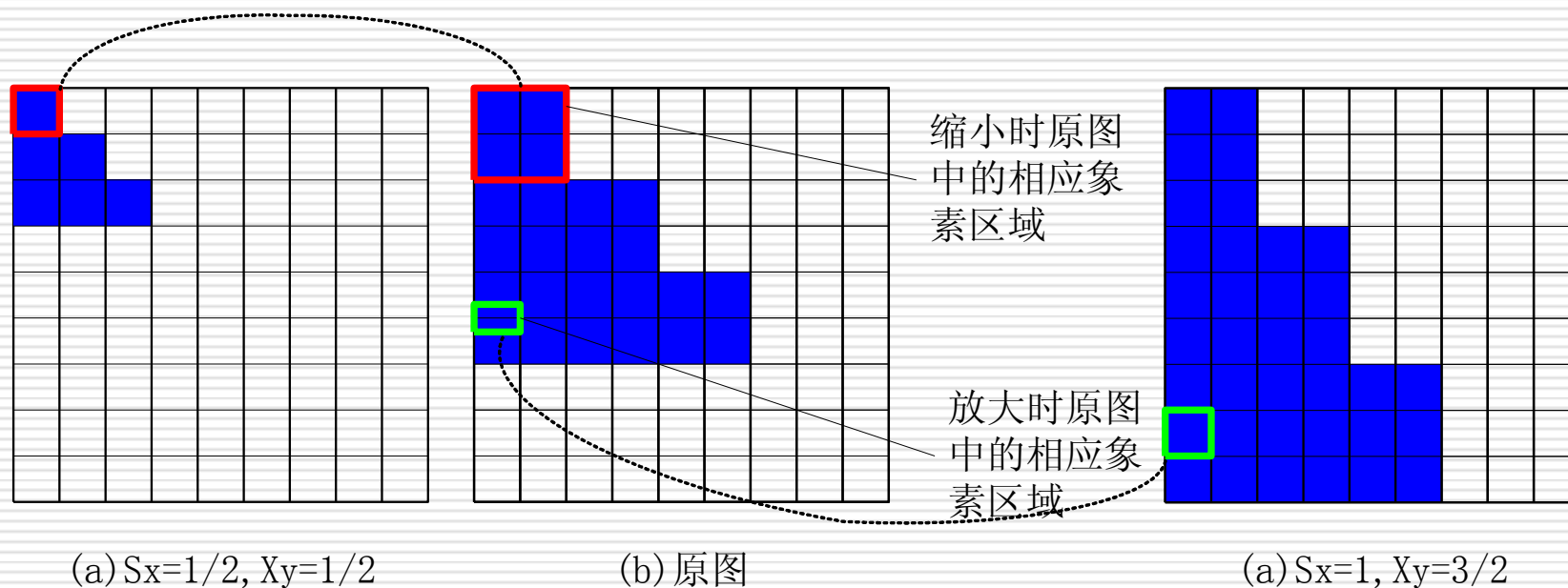


图6-16 光栅比例变换

变换的性质

二维仿射变换是具有如下形式的二维坐标变换:

$$\begin{cases} x' = ax + by + m \\ y' = cx + dy + n \end{cases}$$

- 平移、比例、旋转、错切和反射等变换均是二维仿射变换的特例，反过来，任何常用的二维仿射变换总可以表示为这五种变换的复合。

变换的性质

- 仅包含旋转、平移和反射的仿射变换维持角度和长度的不变性;
- 比例变换可改变图形的大小和形状;
- 错切变换引起图形角度关系的改变, 甚至导致图形发生畸变。

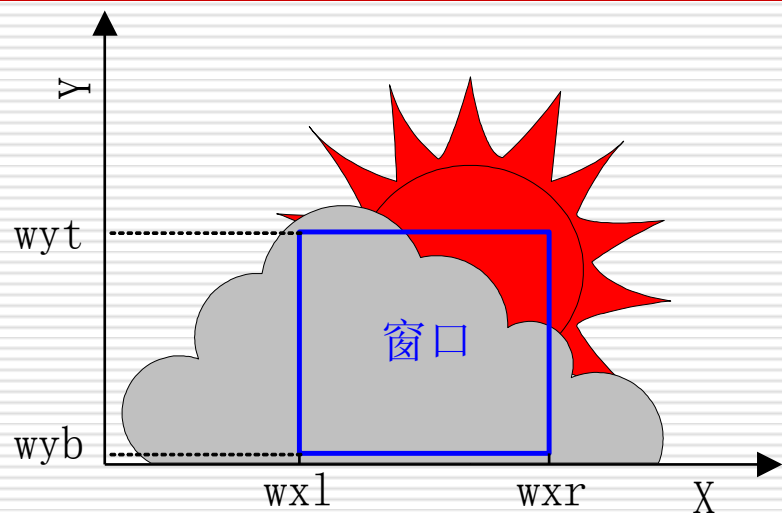
二维观察

- 基本概念
- 二维观察变换
- 二维裁剪
- **OpenGL**中的二维观察

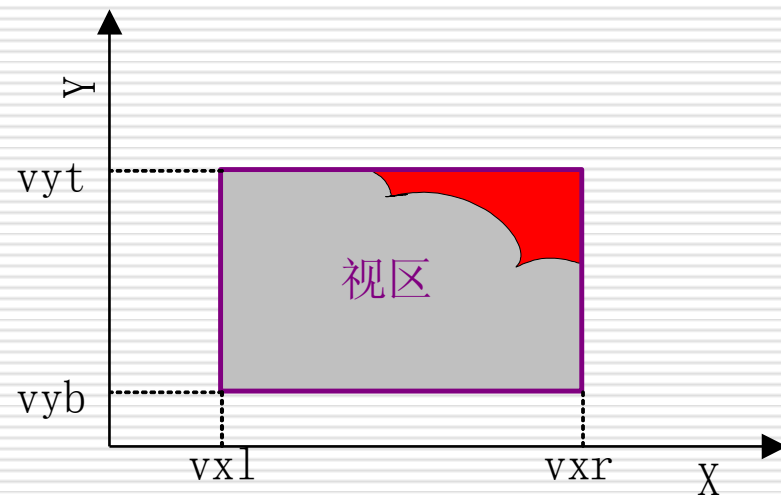
二维观察——基本概念

- 在计算机图形学中，将在用户坐标系中需要进行观察和处理的一个坐标区域称为窗口（Window）。
- 将窗口映射到显示设备上的坐标区域称为视区（Viewport）。

二维观察——基本概念



(a) 用户坐标系中的窗口



(b) 屏幕坐标系中的视区

- 要将窗口内的图形在视区中显示出来，必须经过将窗口到视区的变换（**Window-Viewport Transformation**）处理，这种变换就是观察变换（**Viewing Transformation**）。

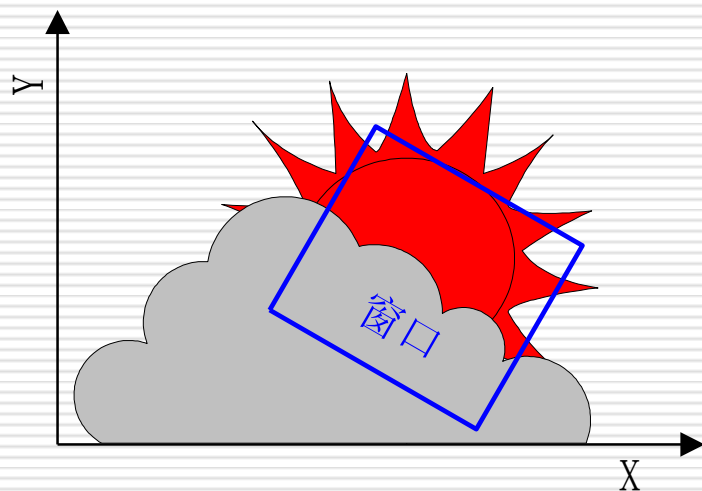
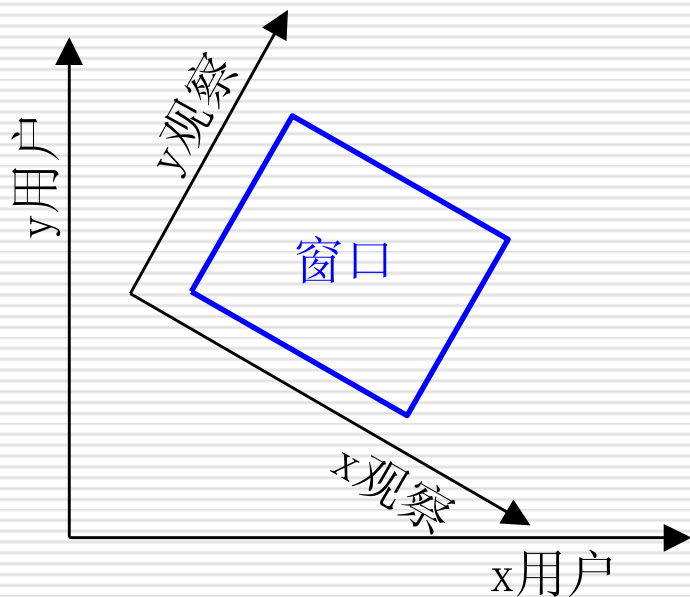


图6-17 用户坐标系中旋转的窗口



(a) 观察坐标系



(b) 规格化设备坐标系

二维观察——基本概念

- 观察坐标系(View Coordinate)是依据窗口的方向和形状在用户坐标平面中定义的直角坐标系。
- 规格化设备坐标系(Normalized Device Coordinate)也是直角坐标系，它是将二维的设备坐标系规格化到(0.0, 0.0)到(1.0, 1.0)的坐标范围内形成的。

二维观察——基本概念

- 引入了观察坐标系和规格化设备坐标系后，观察变换分为如下图所示的几个步骤，通常称为**二维观察流程**。



图6-17 二维观察流程

二维观察——基本概念

□ 变焦距效果

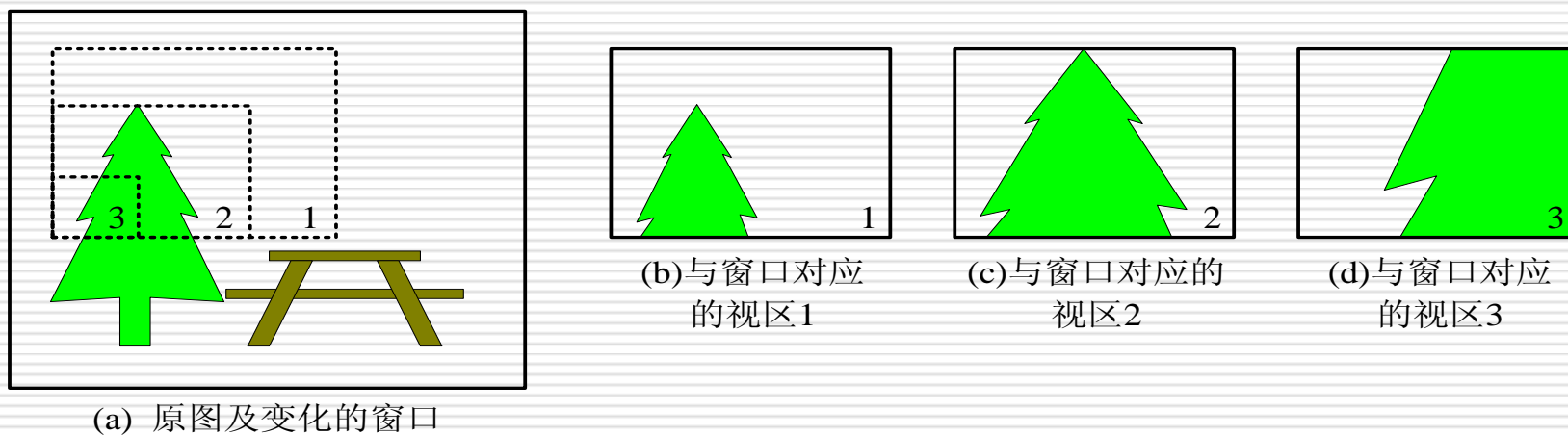


图6-18 变焦距效果（窗口变、视区不变）

二维观察——基本概念

□ 整体放缩效果

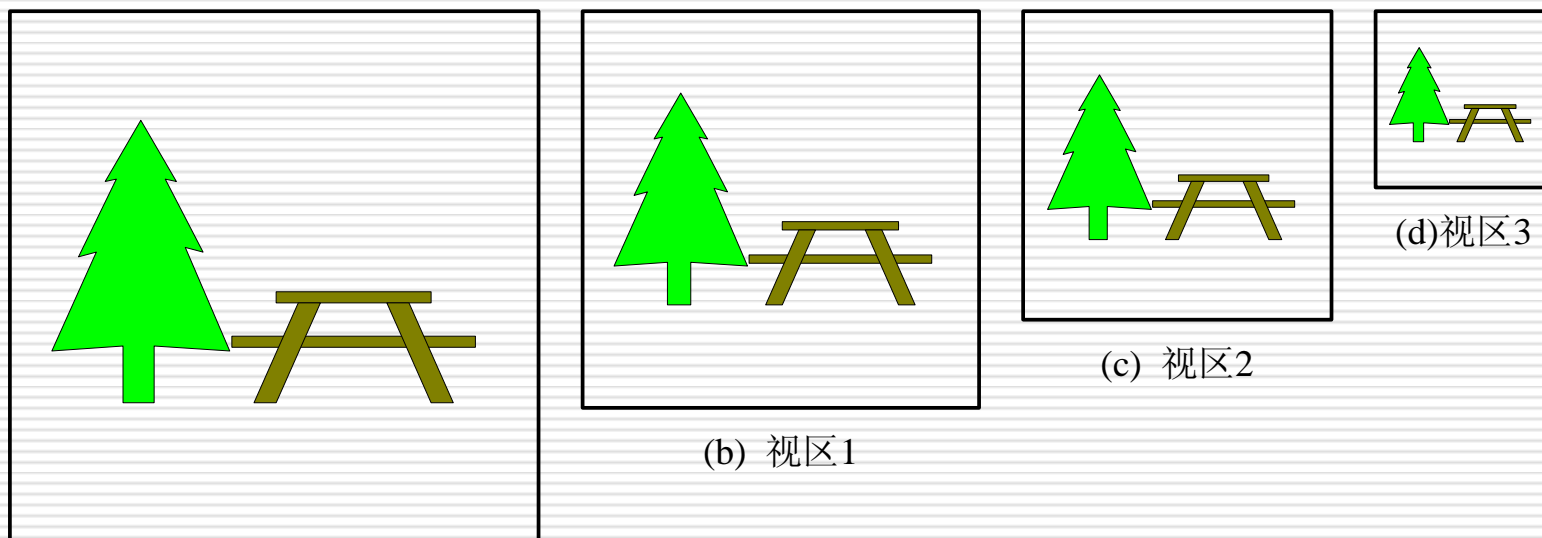
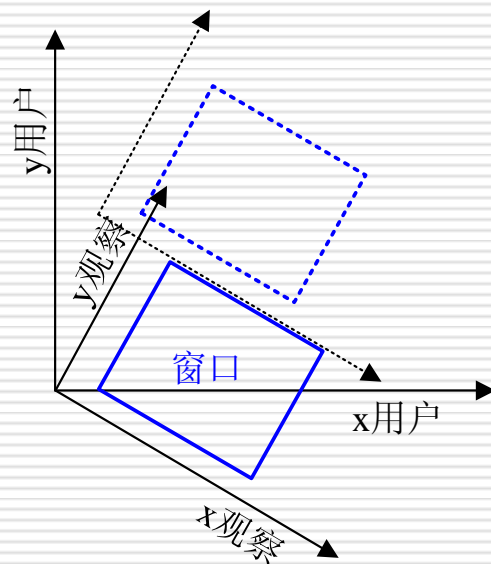


图6-17 整体放缩效果（窗口不变、视区变）

□ 漫游效果

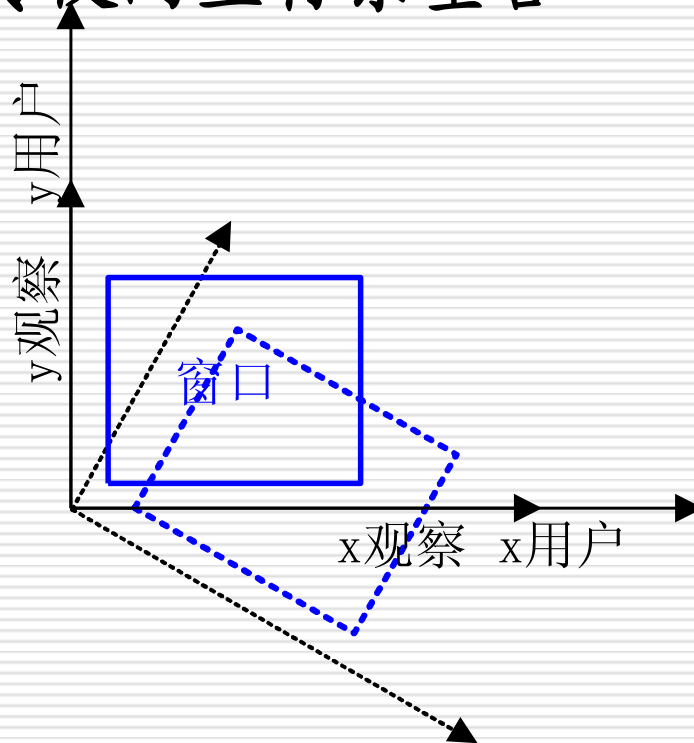
用户坐标系到观察坐标系的变换

- 用户坐标系到观察坐标系的变换分由两个变换步骤合成：
 - ◆ 将观察坐标系原点移动到用户坐标系原点；



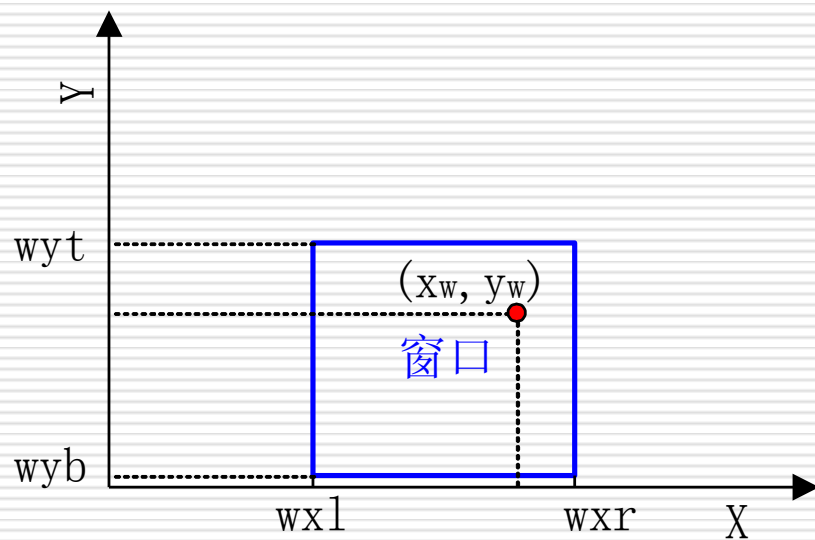
用户坐标系到观察坐标系的变换

◆ 绕原点旋转使两坐标系重合

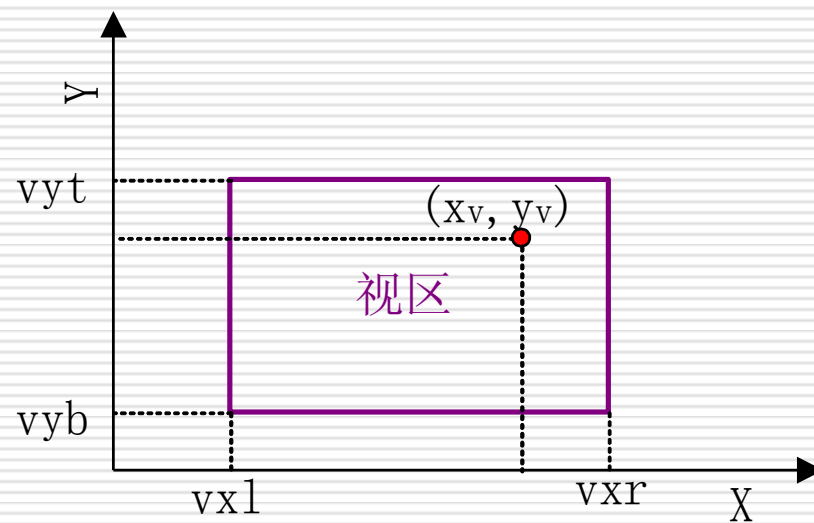


(b) 旋转变换

窗口到视区的变换



(a) 窗口中的点



(b) 视区中的点

图6-23 窗口到视区的变换

窗口到视区的变换

□ 要将窗口内的点 (x_w, y_w) 映射到相对应的视区内的点 (x_v, y_v) 需进行以下步骤:

(1) 将窗口左下角点移至用户系统系的坐标原点;

(2) 针对原点进行比例变换;

(3) 进行反平移。

裁剪

- 在二维观察中，需要在观察坐标系下对窗口进行裁剪，即只保留窗口内的那部分图形，去掉窗口外的图形。
- 假设窗口是标准矩形，即边与坐标轴平行的矩形，由上（ $y=wyt$ ）、下（ $y=wyb$ ）、左（ $x=wxl$ ）、右（ $x=wxr$ ）四条边描述。

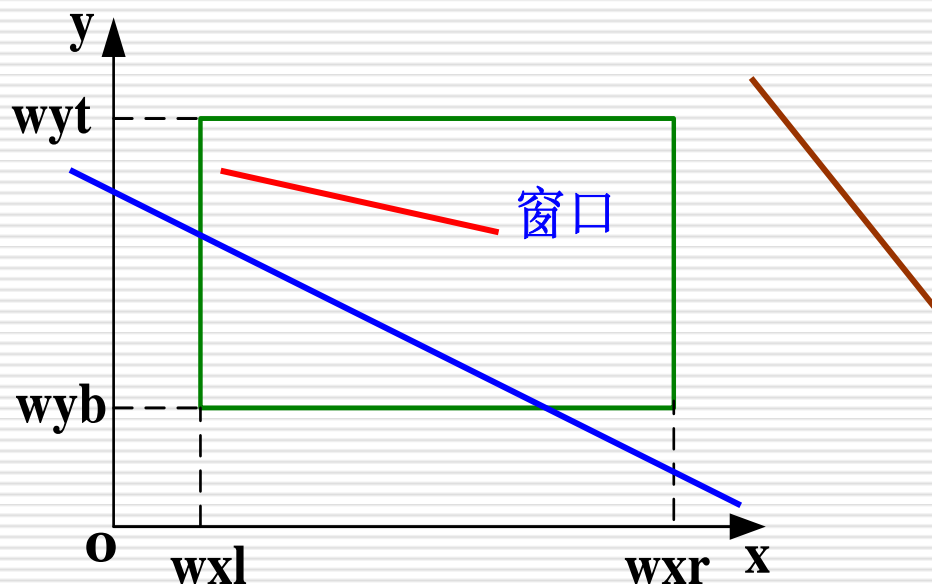
裁剪——点的裁剪

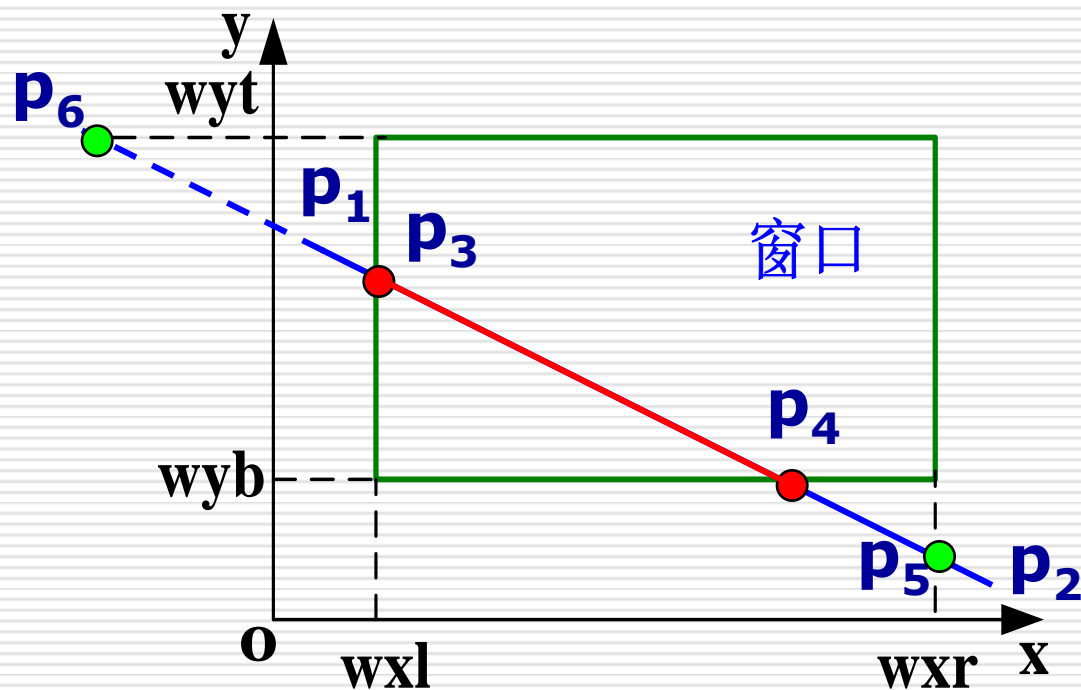
$$wxl \leq x \leq wxr,$$
$$\text{且 } wyb \leq y \leq wyt$$

二维直线段的裁剪

已知条件:

- (1) 窗口边界 wxl , wxr , wyb , wyt 的坐标值;
- (2) 直线段端点 p_1p_2 的坐标值 x_1, y_1, x_2, y_2 。





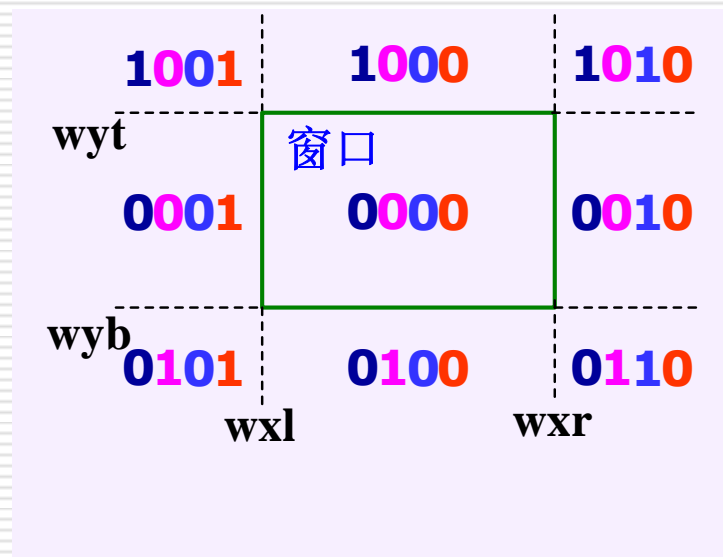
- 实交点：直线段与窗口矩形边界的交点；
- 虚交点：处于直线段延长线或窗口边界延长线上的交点。

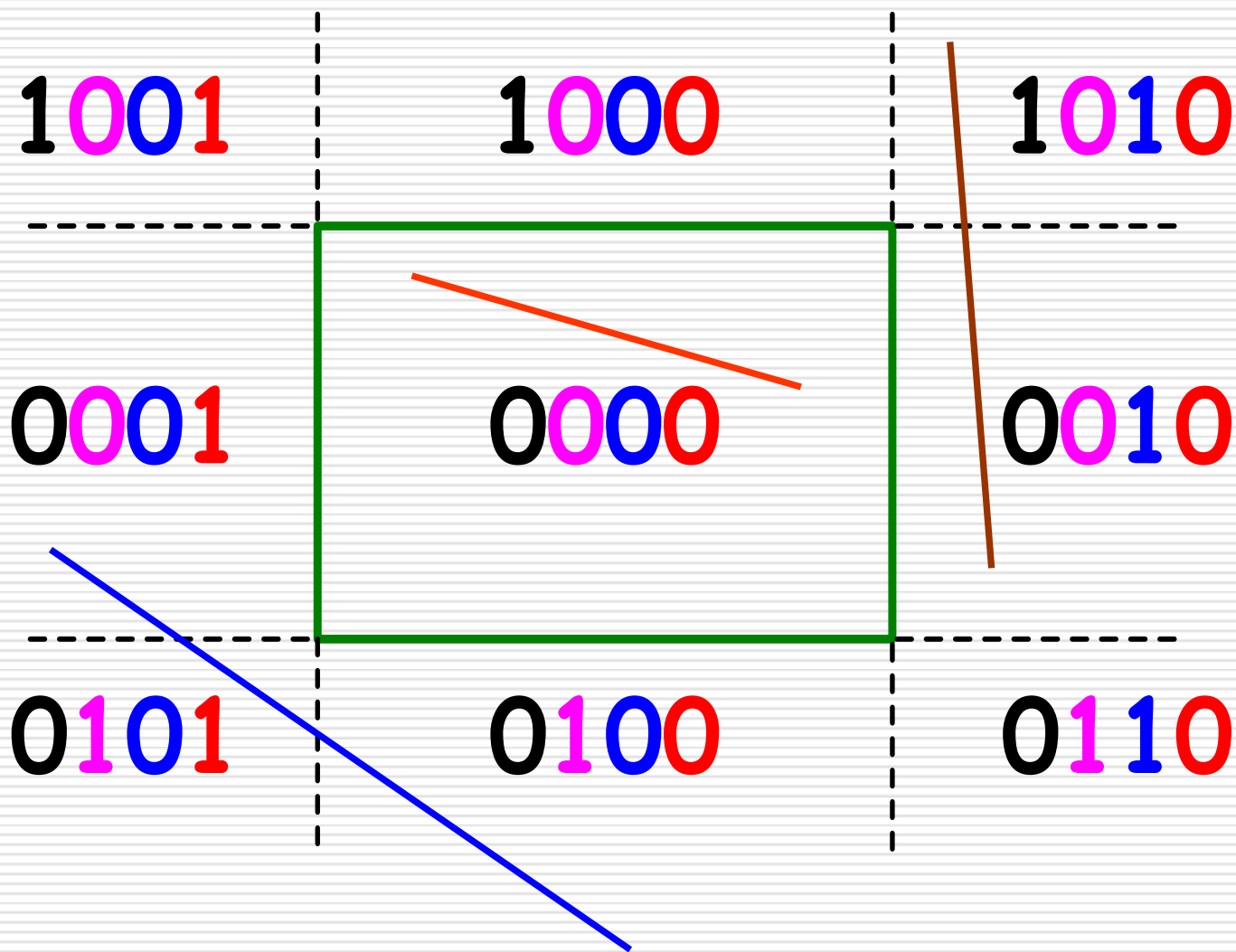
Cohen-Sutherland算法

编码：对于任一端点 (x, y) ，根据其坐标所在的区域，赋予一个4位的二进制码 $D_3D_2D_1D_0$ 。

编码规则如下：

- (1) 若 $x < wxl$, $D_0=1$, 否则 $D_0=0$;
- (2) 若 $x > wxr$, $D_1=1$, 否则 $D_1=0$;
- (3) 若 $y < wyb$, $D_2=1$, 否则 $D_2=0$;
- (4) 若 $y > wyt$, $D_3=1$, 否则 $D_3=0$ 。





Cohen-Sutherland算法

(1) 判断

裁剪一条线段时，先求出直线段端点 p_1 和 p_2 的编码 $code_1$ 和 $code_2$ ，然后：

a. 若 $code_1 | code_2 = 0$ ，对直线段简取之；

b. 若 $code_1 \& code_2 \neq 0$ ，对直线段简弃之；

Cohen-Sutherland算法

(2) 求交

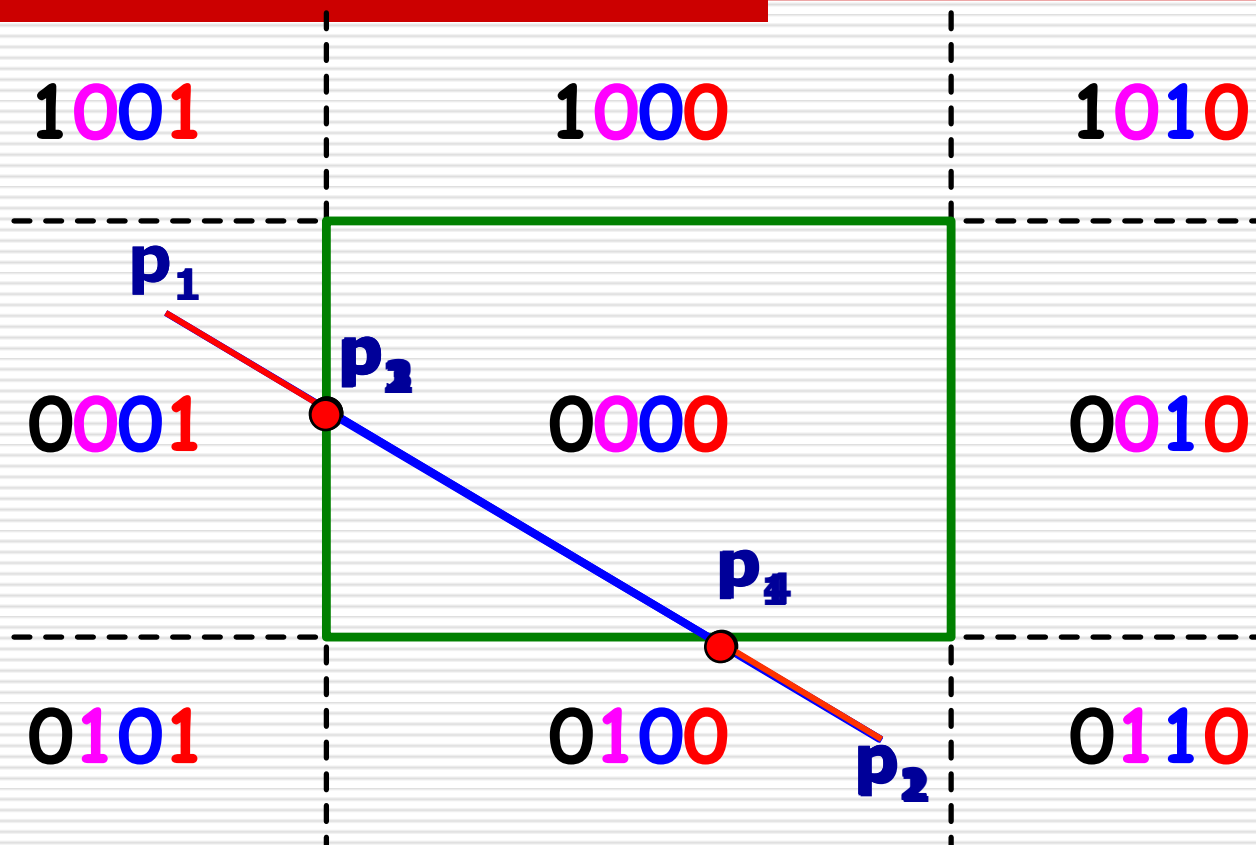
若上述判断条件不成立，则需求出直线段与窗口边界的交点。

a. 左、右边界交点的计算: $y = y_1 + k(x - x_1)$;

b. 上、下边界交点的计算: $x = x_1 + (y - y_1)/k$ 。

其中, $k = (y_2 - y_1) / (x_2 - x_1)$ 。

Cohen-Sutherland算法

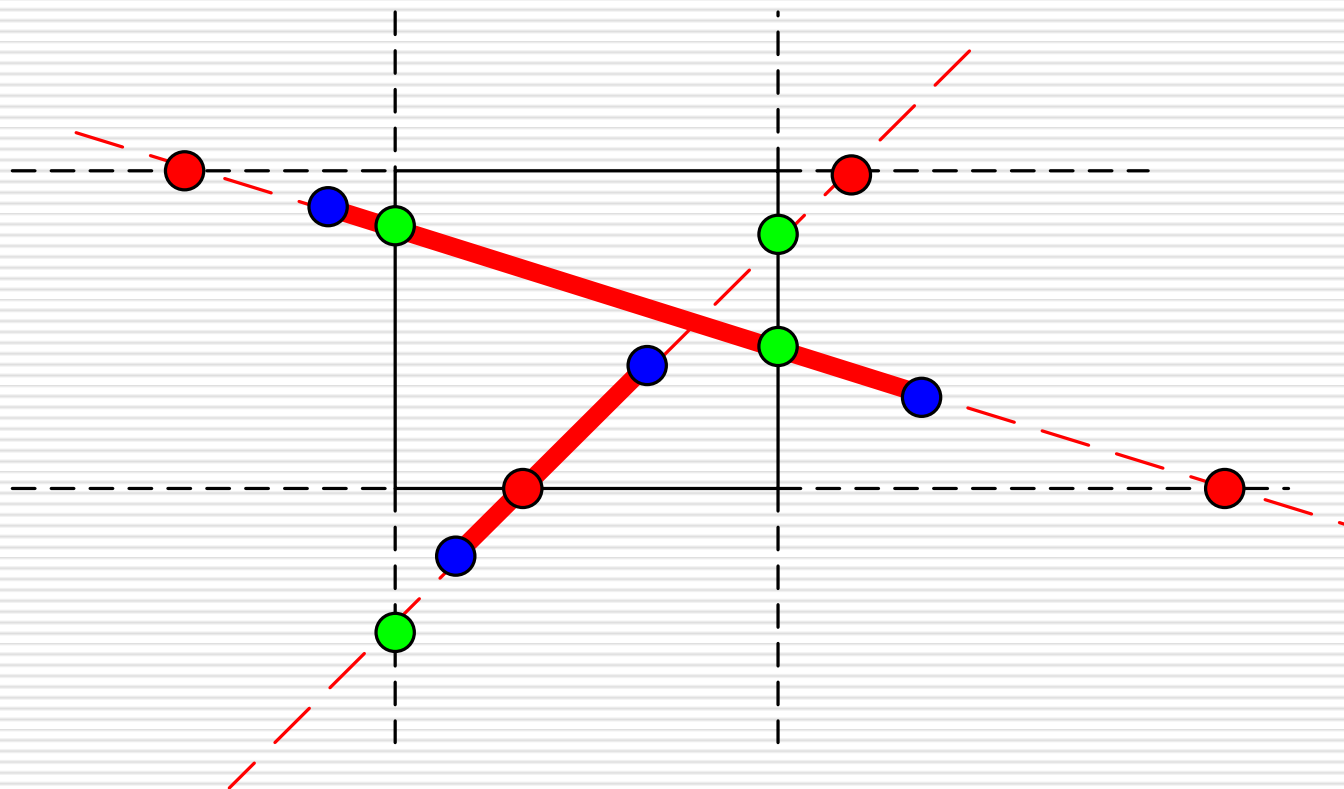


Cohen-Sutherland算法

- 用编码方法实现了对完全可见和不可见直线段的快速接受和拒绝;
- 求交过程复杂, 有冗余计算, 并且包含浮点运算, 不利于硬件实现。

Liang-Barsky算法

分析



Liang-Barsky算法

直线的参数方程

$$\begin{aligned}x &= x_1 + u \cdot (x_2 - x_1) \\ y &= y_1 + u \cdot (y_2 - y_1)\end{aligned} \quad 0 \leq u \leq 1$$

对于直线上一点 (x, y) , 若它在窗口内则有

$$wxl \leq x_1 + u \cdot (x_2 - x_1) \leq wxr$$

$$wxb \leq y_1 + u \cdot (y_2 - y_1) \leq wyt$$

$$u \cdot (x_1 - x_2) \leq x_1 - wxl$$

$$u \cdot (x_2 - x_1) \leq wxr - x_1$$

$$u \cdot (y_1 - y_2) \leq y_1 - wyb$$

$$u \cdot (y_2 - y_1) \leq wyt - y_1$$

$$p_1 = -(x_2 - x_1) \quad q_1 = x_1 - wxl$$

$$p_2 = x_2 - x_1 \quad q_2 = wxr - x_1$$

$$p_3 = -(y_2 - y_1) \quad q_3 = y_1 - wyb$$

$$p_4 = y_2 - y_1 \quad q_4 = wyt - y_1$$

则有 $u \cdot p_k \leq q_k$

- 任何平行于剪切边界之一的直线 $p_k=0$,其中 k 对应于该剪切边界 ($k=1,2,3,4$ 对应于左、右、下、上边界)。如果还满足 $q_k<0$,则线段完全在边界之外,因此舍弃该线段。如果 $q_k\geq 0$,则线段位于边界之内。
- 当 $p_k<0$,线段从剪切边界延长线的外部延长到内部。当 $p_k>0$,线段从剪切边界延长线的内部延长到外部。当 $p_k\neq 0$,可以计算出线段与边界 k 的延长线的交点的 u 值:

$$u = \frac{q_k}{p_k}$$

Liang-Barsky算法

$$\text{由 } u \cdot p_k \leq q_k$$

$$\left\{ \begin{array}{l} \frac{q_k}{p_k} (p_k < 0) \leq u \leq \frac{q_k}{p_k} (p_k > 0) \quad k = 1, 2 \\ \frac{q_k}{p_k} (p_k < 0) \leq u \leq \frac{q_k}{p_k} (p_k > 0) \quad k = 3, 4 \\ 0 \leq u \leq 1 \end{array} \right.$$

特殊处理:

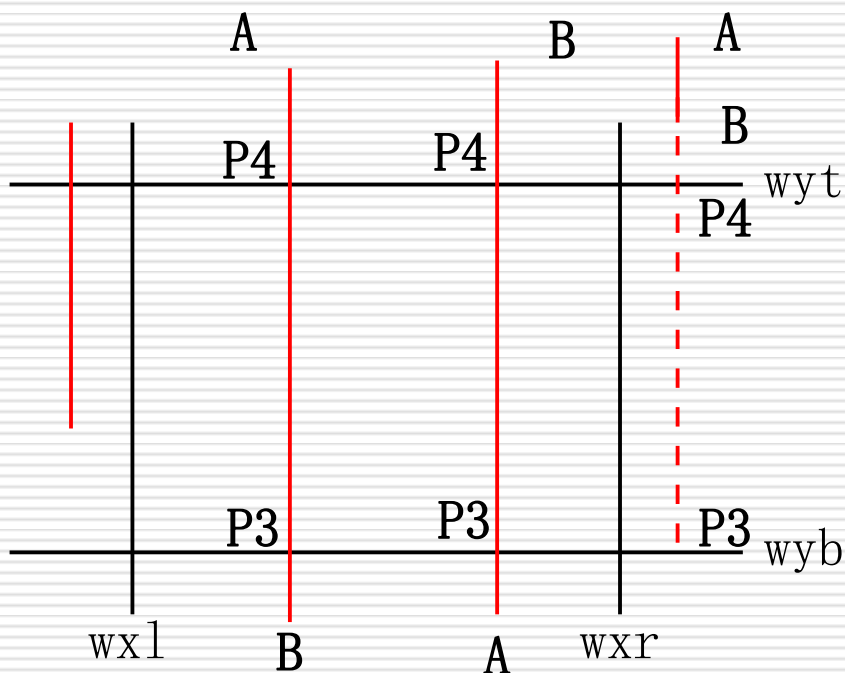
$$p_3 = -(y_2 - y_1) \quad q_3 = y_1 - wyb$$

$$p_4 = y_2 - y_1 \quad q_4 = wyt - y_1$$

求出参数值:

$$u_3 = q_3/p_3, \quad u_4 = q_4/p_4$$

$$u_A = 0, \quad u_B = 1,$$



(a) 直线段与窗口边界
 wxl 和 wxr 平行的情况

$$u_{\max} = \max(0, u_k \mid p_k < 0)$$

$$u_{\min} = \min(u_k \mid p_k > 0, 1)$$

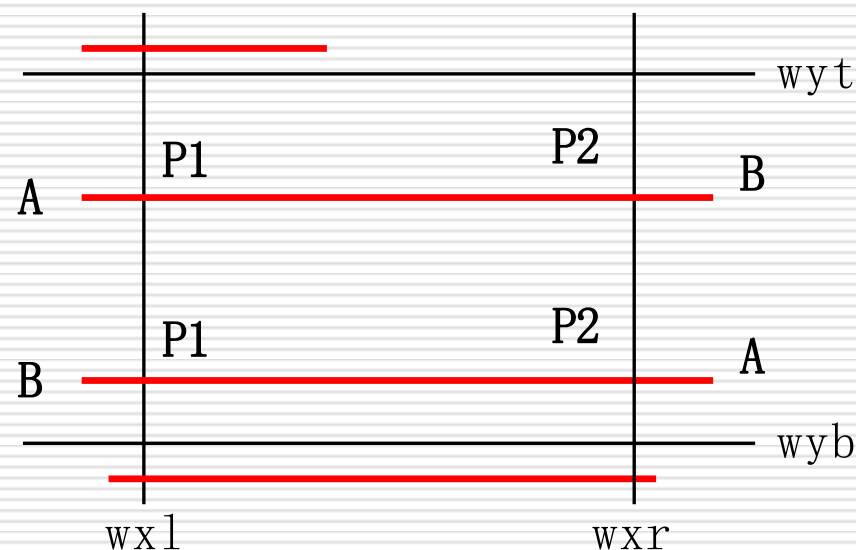
$$p_1 = -(x_2 - x_1) \quad q_1 = x_1 - wxl$$

$$p_2 = x_2 - x_1 \quad q_2 = wxr - x_1$$

求出参数值:

$$u_1 = q_1/p_1, \quad u_2 = q_2/p_2$$

$$u_A = 0, \quad u_B = 1,$$



(b) 直线段与窗口边界
wyb和wyt平行的情况

$$u_{\max} = \max(0, u_k \mid p_k < 0)$$

$$u_{\min} = \min(u_k \mid p_k > 0, 1)$$

$$p_1 = -(x_2 - x_1)$$

$$q_1 = x_1 - wxl$$

$$p_2 = x_2 - x_1$$

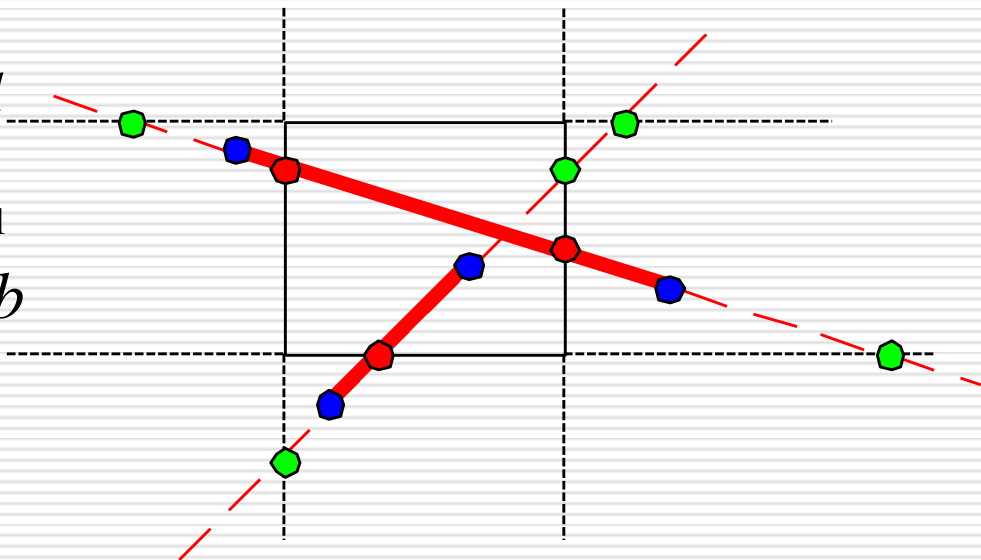
$$q_2 = wxr - x_1$$

$$p_3 = -(y_2 - y_1)$$

$$q_3 = y_1 - wxb$$

$$p_4 = y_2 - y_1$$

$$q_4 = wyt - y_1$$



一般情况:

$$u_{\max} = \max(0, u_k \mid p_k < 0, u_k \mid p_k < 0)$$

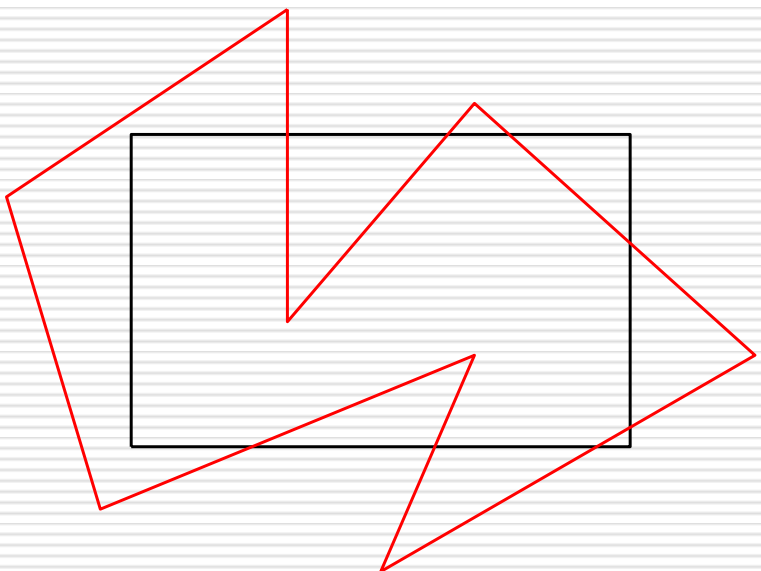
$$u_{\min} = \min(u_k \mid p_k > 0, u_k \mid p_k > 0, 1)$$

算法步骤:

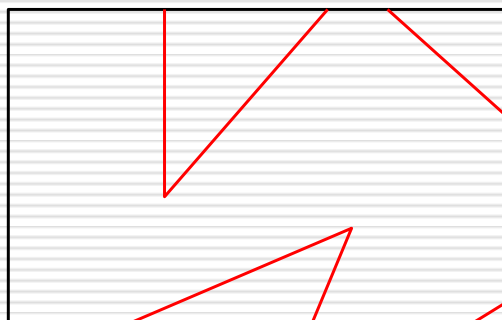
- (1) 输入直线段的两端点坐标: (x_1, y_1) 和 (x_2, y_2) , 以及窗口的四条边界坐标: wyt 、 wyb 、 wxl 和 wxr 。
- (2) 若 $\Delta x = 0$, 则 $p_1 = p_2 = 0$ 。此时进一步判断是否满足 $q_1 < 0$ 或 $q_2 < 0$, 若满足, 则该直线段不在窗口内, 算法转(7)。否则, 满足 $q_1 > 0$ 且 $q_2 > 0$, 则进一步计算 u_1 和 u_2 。算法转(5)。
- (3) 若 $\Delta y = 0$, 则 $p_3 = p_4 = 0$ 。此时进一步判断是否满足 $q_3 < 0$ 或 $q_4 < 0$, 若满足, 则该直线段不在窗口内, 算法转(7)。否则, 满足 $q_1 > 0$ 且 $q_2 > 0$, 则进一步计算 u_1 和 u_2 。算法转(5)。
- (4) 若上述两条均不满足, 则有 $p_k \neq 0$ ($k=1, 2, 3, 4$)。此时计算 u_1 和 u_2 。
- (5) 求得 u_1 和 u_2 后, 进行判断: 若 $u_1 > u_2$, 则直线段在窗口外, 算法转(7)。若 $u_1 < u_2$, 利用直线的参数方程求得直线段在窗口内的两端点坐标。
- (6) 利用直线的扫描转换算法绘制在窗口内的直线段。算法结束。
- (7) 算法结束。

多边形的裁剪

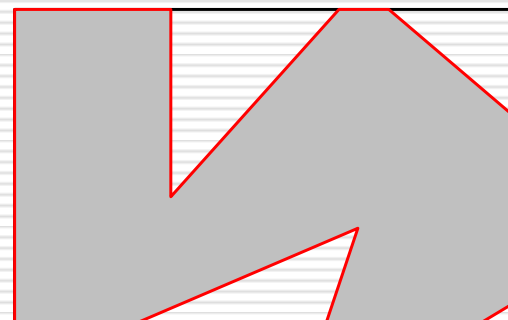
问题的提出:



(a) 裁剪前



(b) 直接采用直线段
裁剪的结果



(c) 正确的裁剪结果

Sutherland-Hodgeman 多边形裁剪

- 基本思想：将多边形的边界作为一个整体，每次用窗口的一条边界对要裁剪的多边形进行裁剪，体现分而治之的思想。

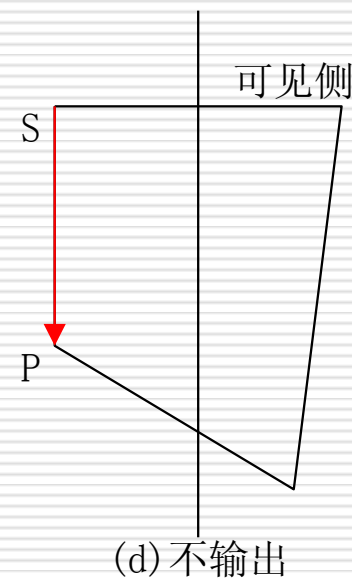
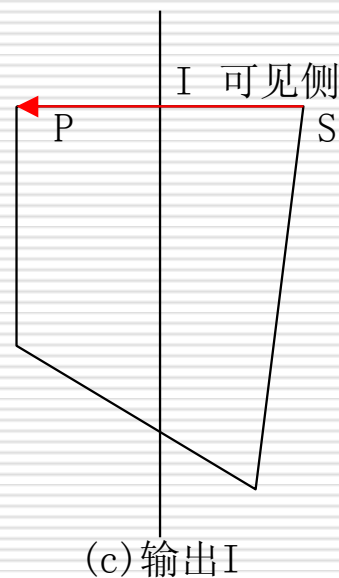
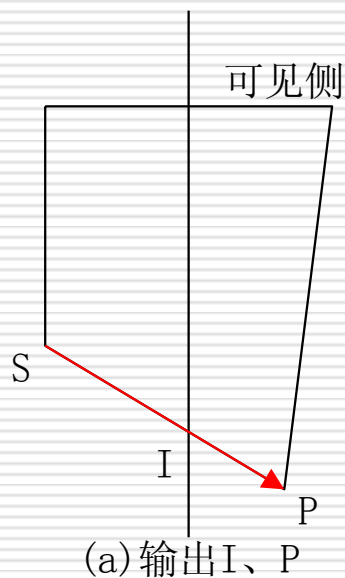
Sutherland-Hodgeman 多边形裁剪

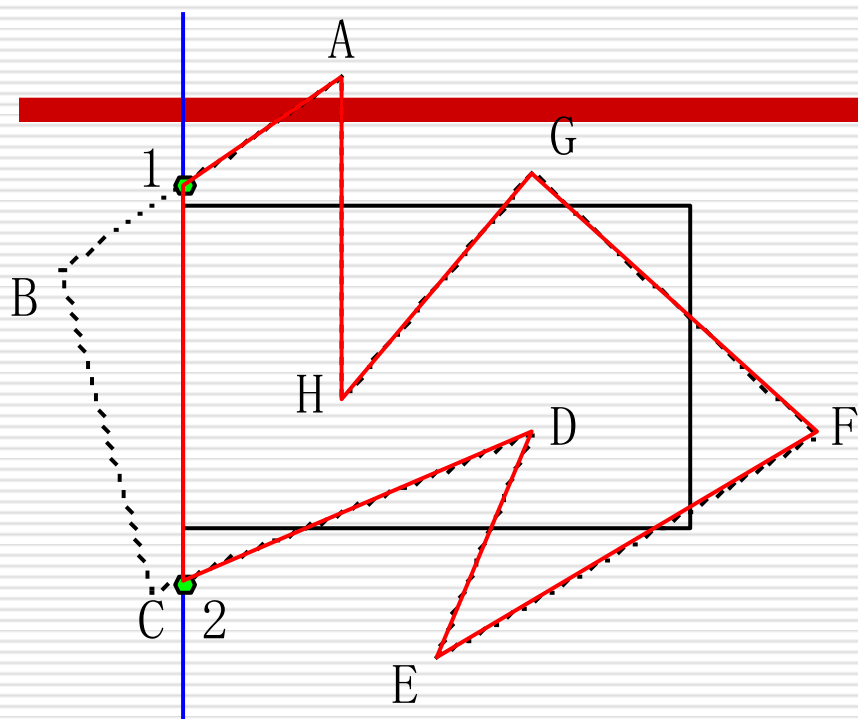
□ 算法实施策略:

- 为窗口各边界裁剪的多边形存储输入与输出顶点表。在窗口的一条裁剪边界处理完所有顶点后，其输出顶点表将用窗口的下一条边界继续裁剪。
- 窗口的一条边以及延长线构成的裁剪线把平面分为两个区域，包含窗口区域的区域称为可见侧；不包含窗口区域的域为不可见侧。

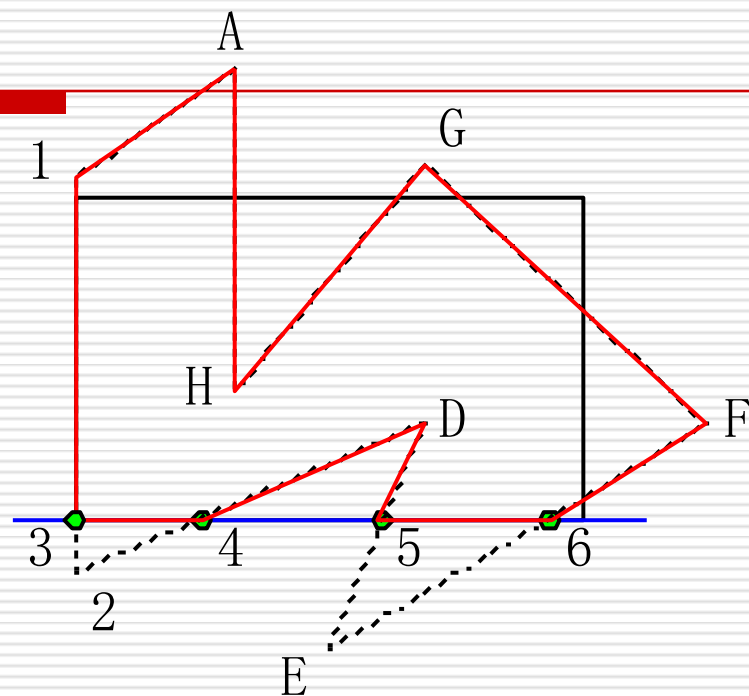
Sutherland-Hodgeman 多边形裁剪

■ 沿着多边形依次处理顶点会遇到四种情况：

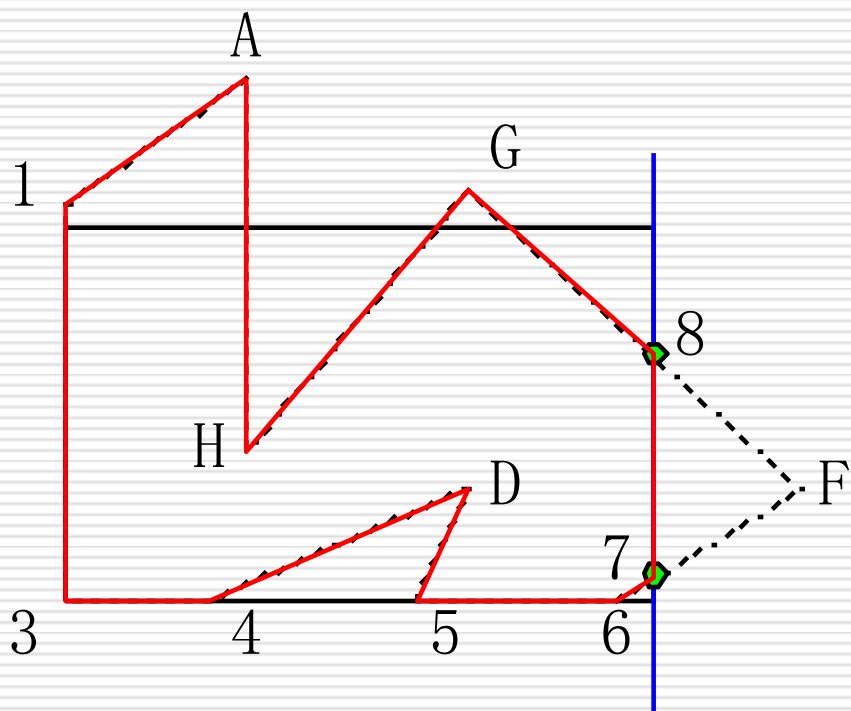




输入: ABCDEFGH
 输出: 12DEFGHA
 (a) 用左边界裁剪

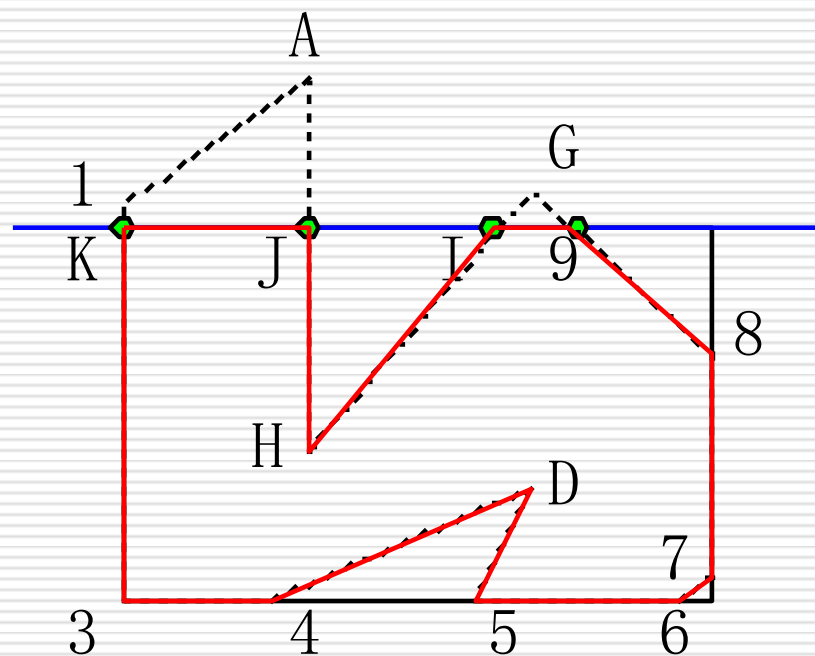


输入: 12DEFGHA
 输出: 34D56FGHA1
 (b) 用下边界裁剪



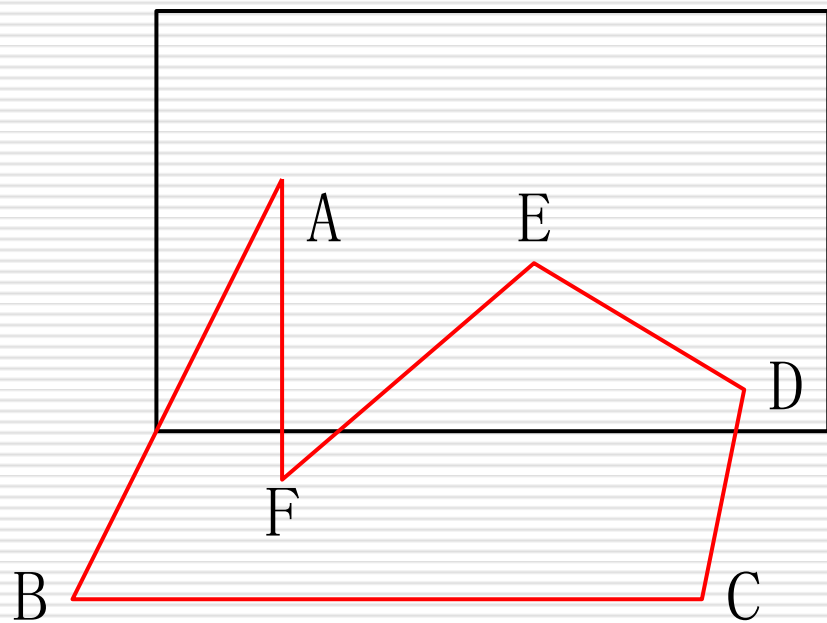
输入：34D56FGHA1

(c) 用右边界裁剪

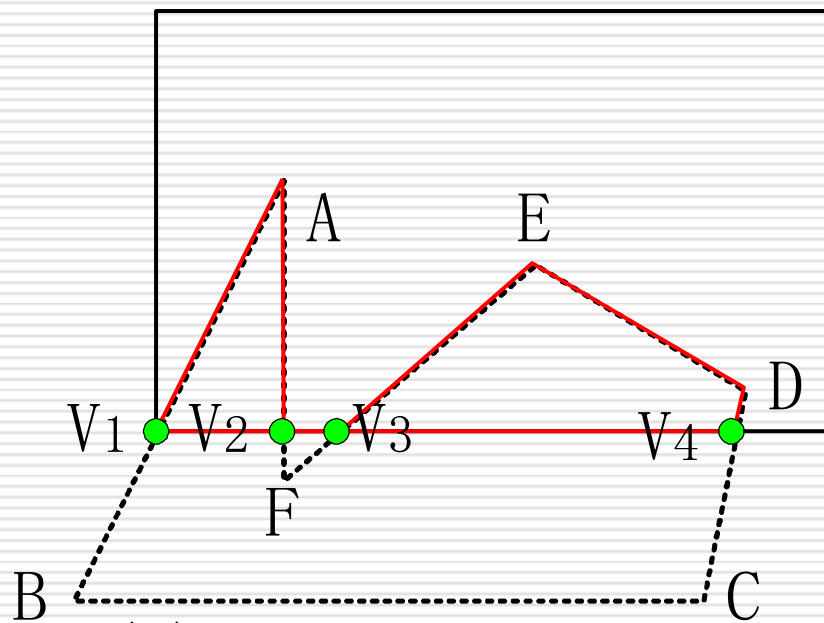


(d) 用上边界裁剪

特点



(a) 裁剪前



(b) Sutherland-Hodgeman
算法的裁剪结果

Weiler-Atherton 多边形裁剪

□ 假定按顺时针方向处理顶点，且将用户多边形定义为 P_s ，窗口矩形为 P_w 。算法从 P_s 的任一点出发，跟踪检测 P_s 的每一条边，当 P_s 与 P_w 相交时（实交点），按如下规则处理：

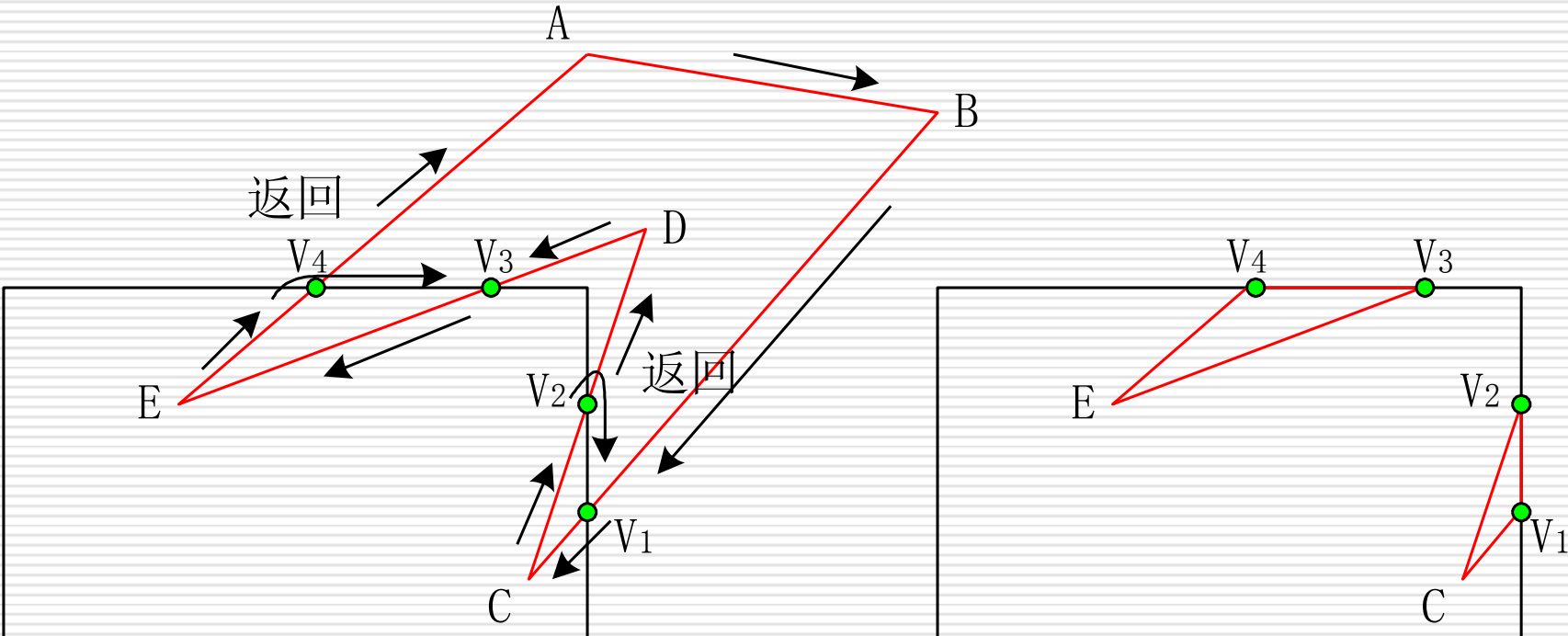
(1)若是由不可见侧进入可见侧，则输出可见直线段，转(3)；

Weiler-Atherton 多边形裁剪

(2)若是由可见侧进入不可见侧，则从当前交点开始，沿窗口边界顺时针检测 P_w 的边，即用窗口的有效边界去裁剪 P_s 的边，找到 P_s 与 P_w 最靠近当前交点的另一交点，输出可见直线段和由当前交点到另一交点之间窗口边界上的线段，然后返回处理的当前交点；

(3)沿着 P_s 处理各条边，直到处理完 P_s 的每一条边，回到起点为止。

□ 下图示了Weiler-Atherton算法裁剪凹多边形的过程和结果。



(a) 裁剪前

(b) Weiler-Atherton 算法的裁剪结果

图6-34 Weiler-Atherton算法裁剪凹多边形

其他裁剪

2. 文字裁剪

文字裁剪的策略包括几种：

- 串精度裁剪
- 字符精度裁剪
- 笔划、象素精度裁剪

3. 外部裁剪

保留落在裁剪区域外的图形部分、去掉裁剪区域内的所有图形，这种裁剪过程称为外部裁剪，也称空白裁剪。

6.6 OpenGL中的二维观察

- 指定矩阵堆栈
- 指定裁剪窗口
- 指定视区

指定矩阵堆栈

- 指定当前操作的是投影矩阵堆栈

glMatrixMode (GL_PROJECTION)

- 初始化，即指定当前操作的矩阵堆栈的栈顶元素为单位矩阵。

glLoadIdentity();

指定裁剪窗口

□ 定义二维裁剪窗口

`gluOrtho2D(xwmin, xwmax, ywmin, ywmax);`

□ 其中，双精度浮点数 **xwmin, xwmax, ywmin, ywmax** 分别对应裁剪窗口的左、右、下、上四条边界。

□ 默认的裁剪窗口，四条边界分别为 **wxl=-1.0, wxr=1.0, wyt=-1.0, wyb=1.0**。

指定裁剪窗口

□ 指定视区

`glViewport (xvmin, yvmin, vpWidth, vpHeight);`

□ **xvmin**和**yvmin**指定了对应于屏幕上显示窗口中的矩形视区的左下角坐标，单位为像素。

□ 整型值**vpWidth**和**vpHeight**则指定了视区的宽度和高度。

□ 默认的视区大小和位置与显示窗口保持一致。
