Procedural Elements for Computer Graphics (Second Edition)

计算机图形学的算法基础

（原书第2版）

（美） David F. Rogers 著
石教英 彭群生 等译

机械工业出版社
China Machine Press
Mc Graw Hill Education

HZ BOOKS
华章教育

国外经典教材

Classical Texts From Top Universities

国外经典教材 Classical Texts From Top Universities

计算机图形学的算法基础

（原书第2版）

Procedural Elements for Computer Graphics

(Second Edition)

本书在很大程度上可以说是一次重写，本书对于学生、程序员和计算机图形学专业人员有特别的价值。新增加的部分主要包括图形用户界面、椭圆、图像压缩和线条反走样算法。除经典的Cohen-Sutherland裁剪算法、中点分割算法、Cyrus-Beck裁剪算法和Sutherland-Hodgman裁剪算法外，还增加了Liang-Barsky裁剪算法和Nicholl-Lee-Nicholl裁剪算法。可见面算法这一章在经典算法之外增加了Appel、晕线、A-buffer、二叉空间剖分(BSP)、八叉树和移动立方体等算法。原来的可见面光线跟踪算法也有很大扩充。绘制一章现在的内容包括基于物理的光照模型、透明材料、阴影和纹理以及锥光线跟踪、束光线跟踪、笔束光线跟踪和随机光线跟踪等。该版本还详细介绍了辐射度的基本知识，并增加了均匀颜色空间、Gamma矫正、彩色图像的量化和印刷介质上的颜色重现等内容。不过也保留了老版本的许多精雕细琢的例子并作了相应的补充。

作者简介

David F. Rogers，计算机图形学的开拓者兼航空工程师。他一共撰写了包括《计算机图形学的数学基础》在内的四部教材，同时还参与编撰了一些其他书籍。他的著作已经被翻译为六种语言。迄今他在计算机图形学和航空学领域已发表50多篇论文和技术报告。

ISBN 7-111-07582-X
9 787111 075820
封面设计 张静闻

华章图书

www.china-pub.com
北京市西城区百万庄南街1号 100037
购书热线：(010)68995259，8006100280 (北京地区)
ISBN 7-111-07582-X/TP · 1210
定价：55.00 元

HZ BOOKS
华章教育
北京华章图文信息有限公司

国外经典教材 ❀ Classical Texts From Top Universities

（原书第2版）

计算机图形学的算法基础

Procedural Elements for Computer Graphics (Second Edition)

（美） David F. Rogers 著
石教英 彭群生 等译

机械工业出版社
China Machine Press

本书从图形学最基础的光栅扫描、区域填充、画直线和圆弧等算法讲起，详细介绍了线裁剪和面裁剪、凸区域裁剪和凹区域裁剪的异同、景物空间消隐算法和图像空间消隐算法的差别，具体讲述了二叉空间剖分(BSP)、八叉树等图形学中常用的数据结构。新版本增加了图形用户界面、椭圆、图像压缩和线条反走样算法等，还增加了Liang-Barsky裁剪算法和Nicholl-Lee-Nicholl裁剪算法。新版本大大扩充了可见面光线跟踪算法。在绘制这一章中新增了基于物理的光照明模型，透明效果，阴影生成，纹理映射，以及锥光束、平面光束、笔形光束和随机光线跟踪算法；详细讨论了光辐射度基础，统一颜色空间，彩色图象量化和印刷介质上的颜色重现等新内容。本书列举了90个例子，具体描述了各类算法的执行细节。这对初学者体会算法的基本原理，比较各类算法执行时的细微差别大有裨益。具备大学数学基础和高级程序设计语言知识的人均可自学本书。

本书适合专业程序员、工程师及科研人员使用。非常适用于作为为高年级本科生和一年级研究生开设的重点讲授绘制技术的计算机图形学课程的教科书。

本书版权登记号：图字：01-1999-2369

图书在版编目(CIP)数据

计算机图形学的算法基础/（美）罗杰斯（Rogers, D. F.）著；石教英等译. －北京：机械工业出版社，2002.1

(国外经典教材)

书名原文：Procedural Elements for Computer Graphics (2E)

ISBN 7-111-07582-x

I. 计… II. ①罗… ②石… III. 计算机图形学—算法理论 IV.TP391.41

中国版本图书馆CIP数据核字(2001)第07669号

机械工业出版社(北京市西城区百万庄大街22号 邮政编码 100037)

责任编辑：陈贤舜

北京第二外国语学院印刷厂印刷·新华书店北京发行所发行

2002年1月第1版第1次印刷

787mm × 1092mm 1/16 · 36.75印张

印数：0 001-5 000册

定价：55.00元

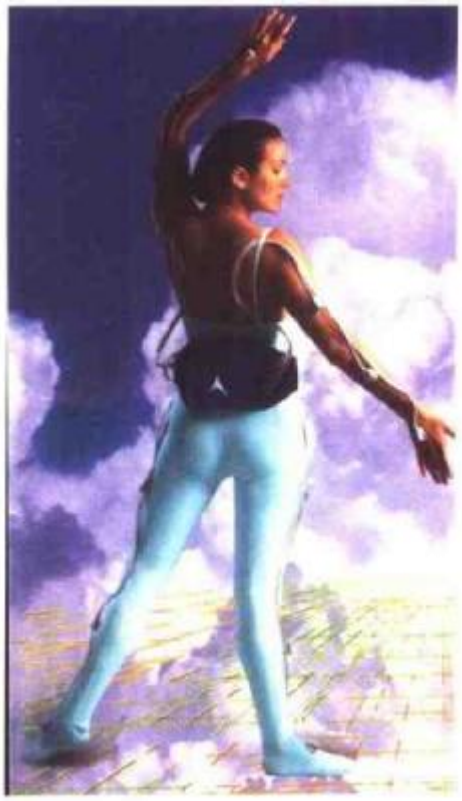


图1 运动捕获。一无线的三维运动状态捕获系统（参见第1.9节）（由Ascension Technology公司提供）

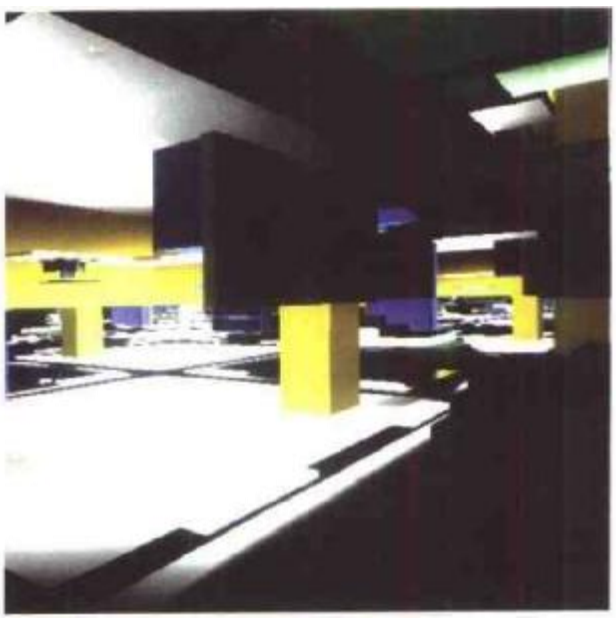


图2 办公室场景。采用层次z-buffer算法绘制（参见第4.9节）
（由纽约理工学院计算机图形实验室N. Greene提供）

图3 1983雪佛兰汽车，D.Warn采用Watkins算法和第5.8节所述的具有特殊效果的光照模型绘制（由通用汽车公司研究院提供）

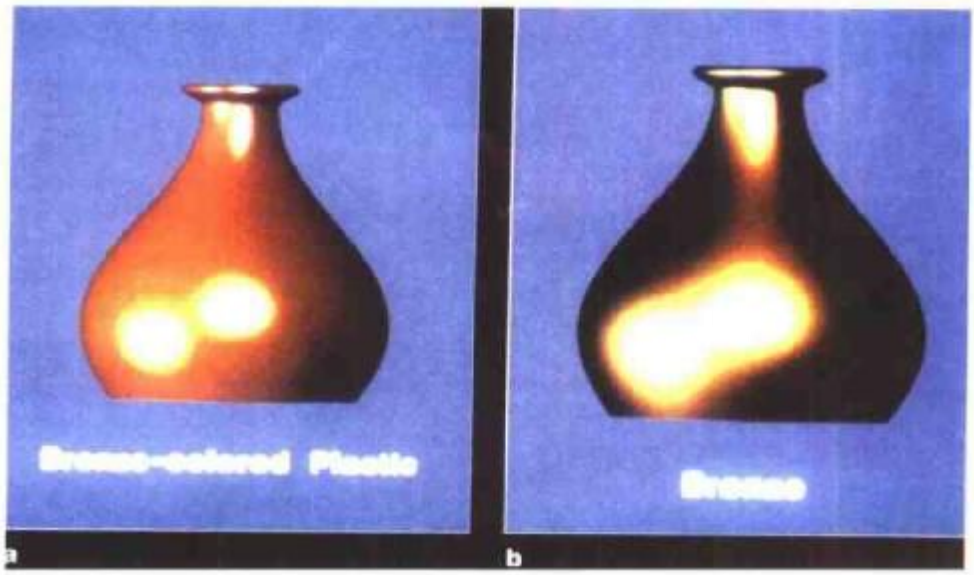


图4 古铜花瓶

a）由古铜色的塑料制成 b）金属铜花瓶每个花瓶都受到两个光源的照射，并采用第5.9节所介绍的Cook-Torrance光照模型绘制而成

（由R. Cook 和Cornell大学计算机图形实验室提供）

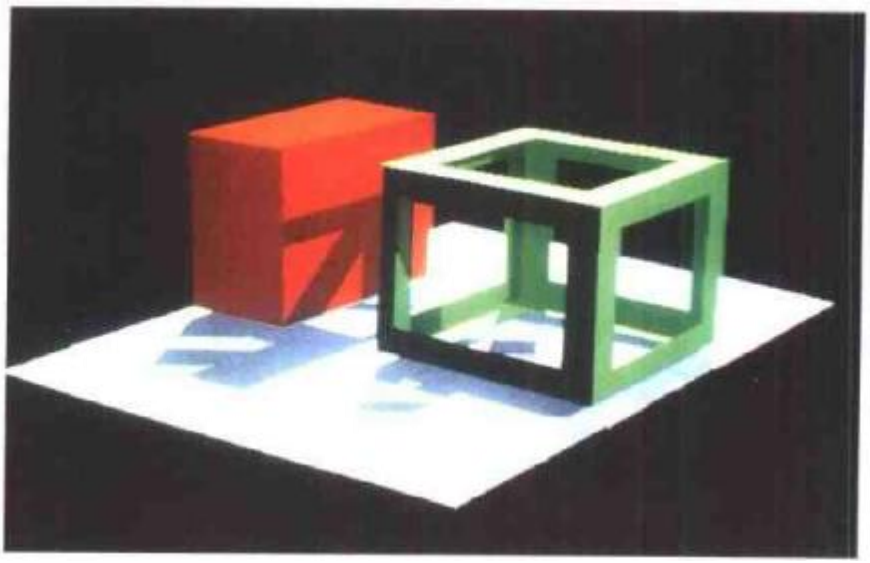


图5　阴影。采用第5.11节所介绍的Weiler-Atherton算法计算该场景在两个光源照射下的阴影
（由P. Atherton 和 Cornell大学计算机图形实验室提供）

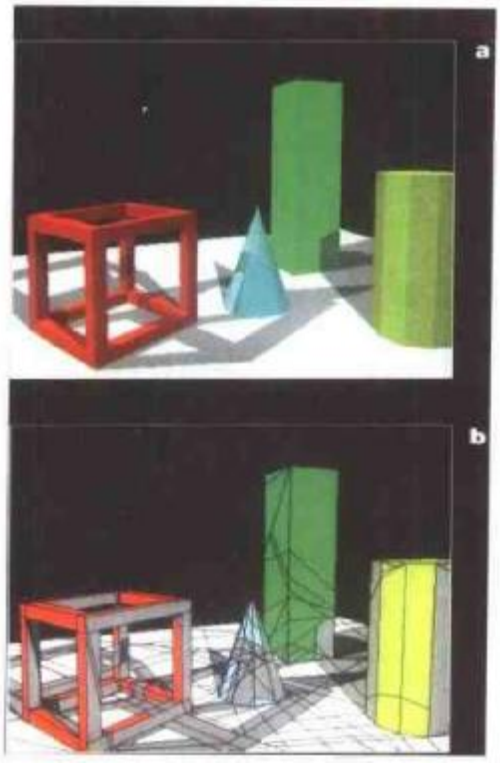


图6　阴影(SVBSP)
a) 场景由二个光源照明　b) 场景中的明暗显示出由BSP树和阴影体SVBSP树生成的阴影多边形区域，阴影区域的颜色由可照射到该区域的光源数目确定
（参见第5.11节）（由SGI公司的N. Chin提供）

图7 阴影体。线光源照明所生成的半影和本影（参见第5.11节）（由Nishita教授和Nakamae教授提供）

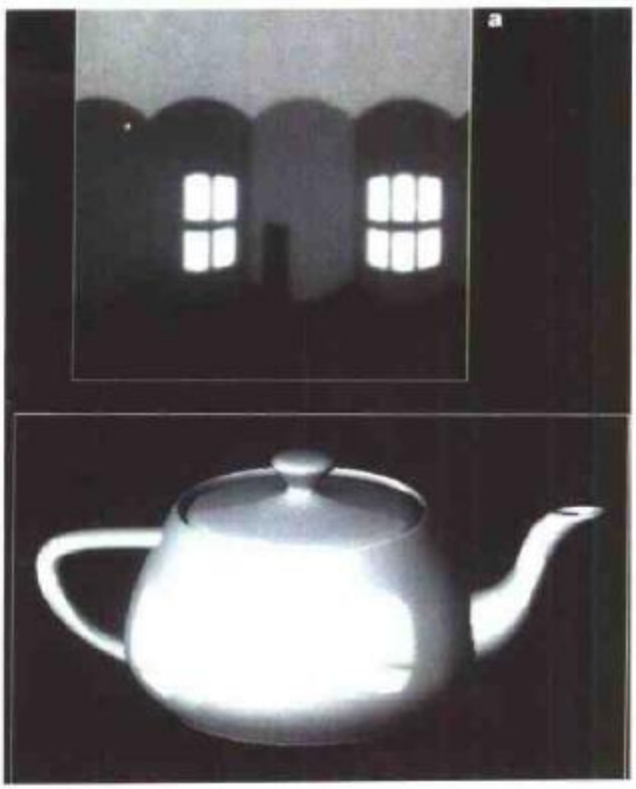


图8 茶壶。茶壶上有窗户的映像
a）环境映照 b）茶壶上的反射（参见第5.12节）（由Jim Blinn 提供）

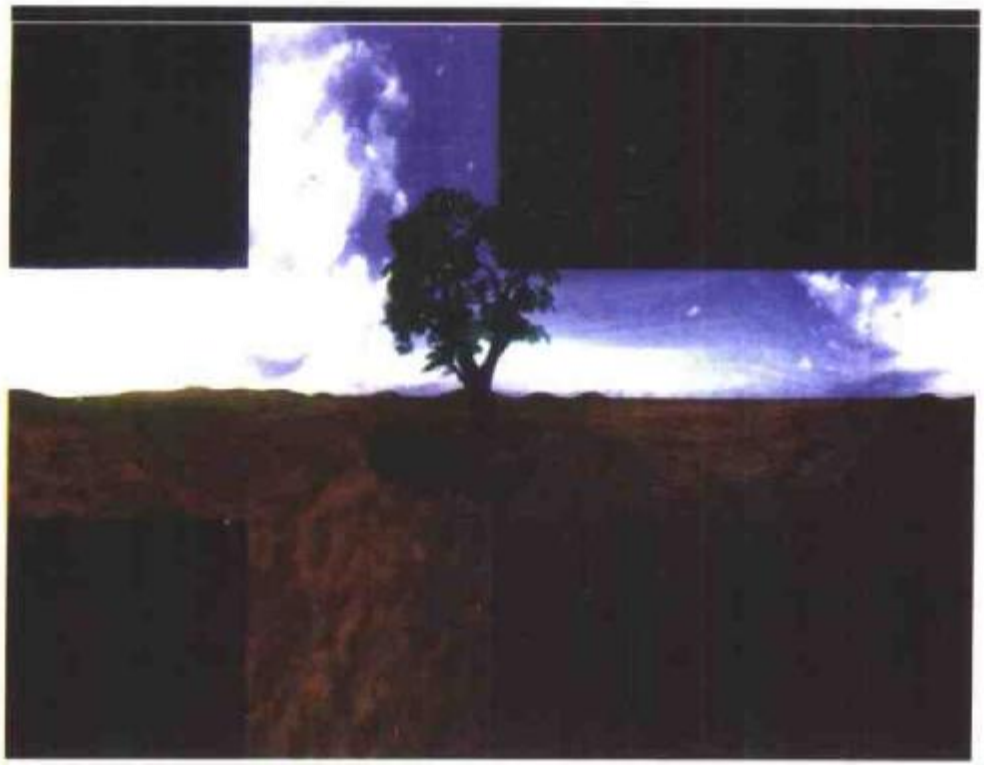


图9 环境映照。三维环境映照立方体的表面展开图（参见第5.12节）
（由纽约理工学院计算机图形实验室的N. Greene提供）

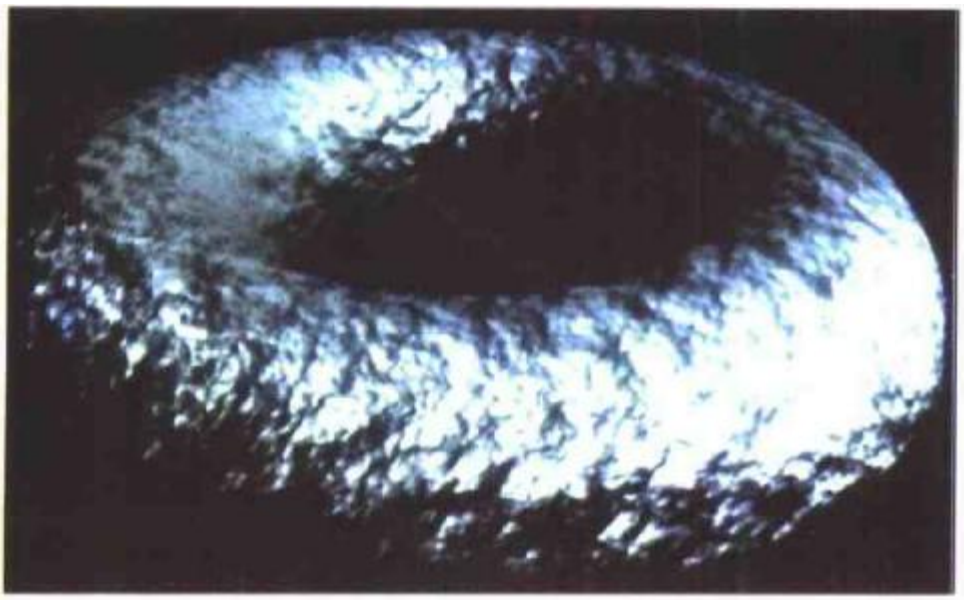


图10 过程纹理。采用过程纹理和Dnoise函数生成的具有凹凸感的面包圈
（参见5.12节）（由K. Perlin提供）

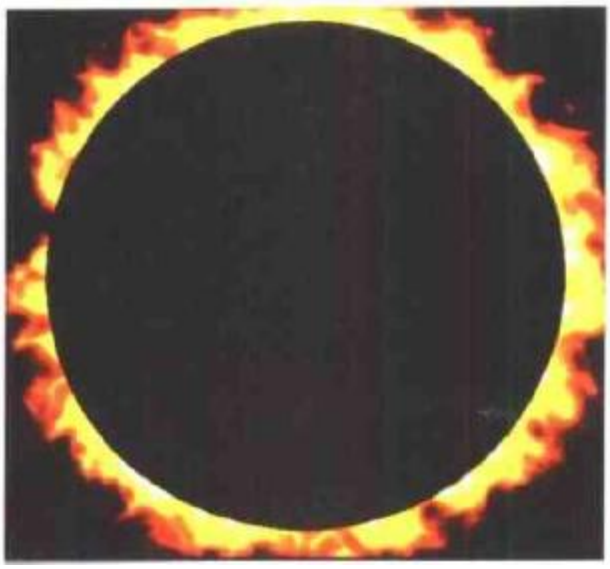


图11　日冕。采用过程纹理模拟的日冕图像（参见第5.12节）（由K.Perlin提供）

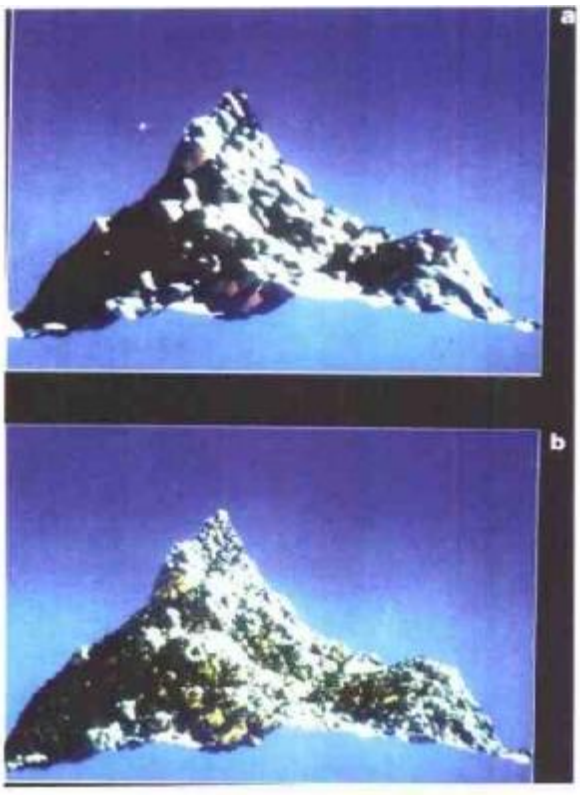


图12　分形山脉图。a）中的山脉图像由16 384个分形三角形组成，图b中的山脉图像由262 144个分形三角形组成。注意在右侧光源的照射下，山体表面呈现的自身阴影（参见第5.13节）（由加州理工学院J. Kajiya提供）

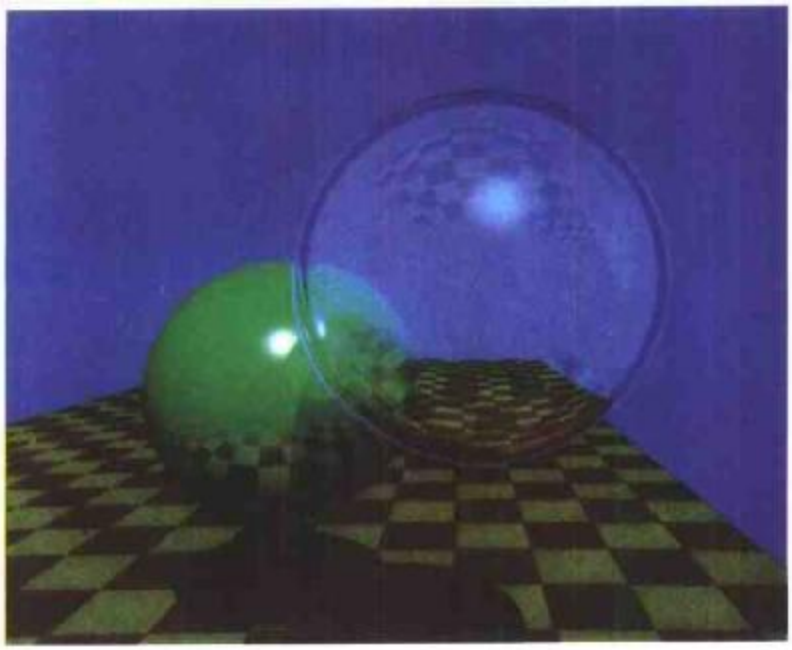


图13　位于红—黄棋盘上的球。采用光线跟踪算法和包含反射、阴影和带折射的透明等效果的整体光照模型绘制（参见第5.14节）（由贝尔实验室的T. Whitted提供，原载于《Communication of the ACM》，1980年6月第23期，1980年。版权：1980年，ACM，获准转载）

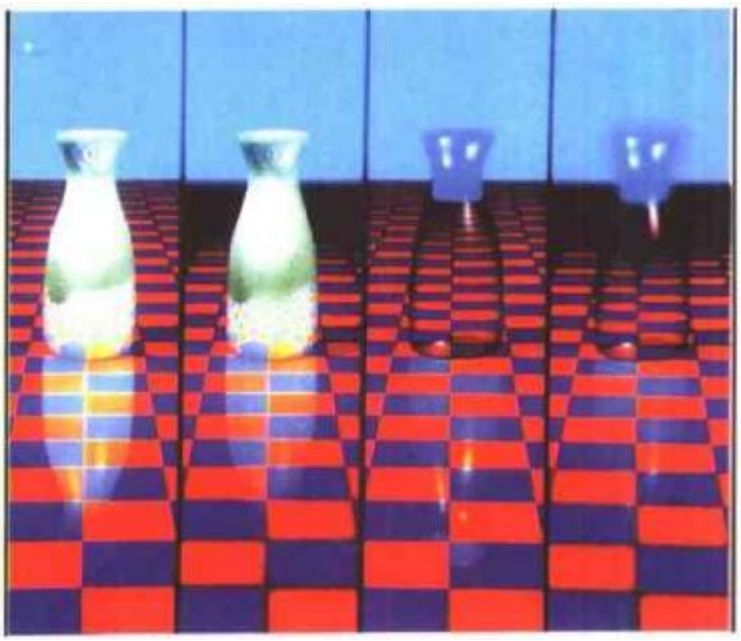


图14　花瓶。图中透明和不透明的花瓶均采用光线跟踪算法绘制。每一种花瓶的右侧图像均含有景深效果（参见第5.14节）（由M. Potmesil和Rensselaer工业大学图像处理实验室提供）

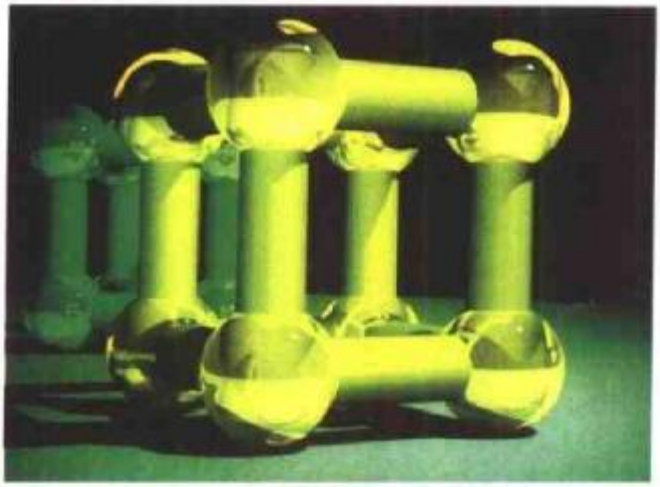


图15 带反射和折射的球和圆柱，采用Whitted光线跟踪算法及其光照模型绘制（参见第5.12节）（由Raster Technologies 公司的A. Barr提供）

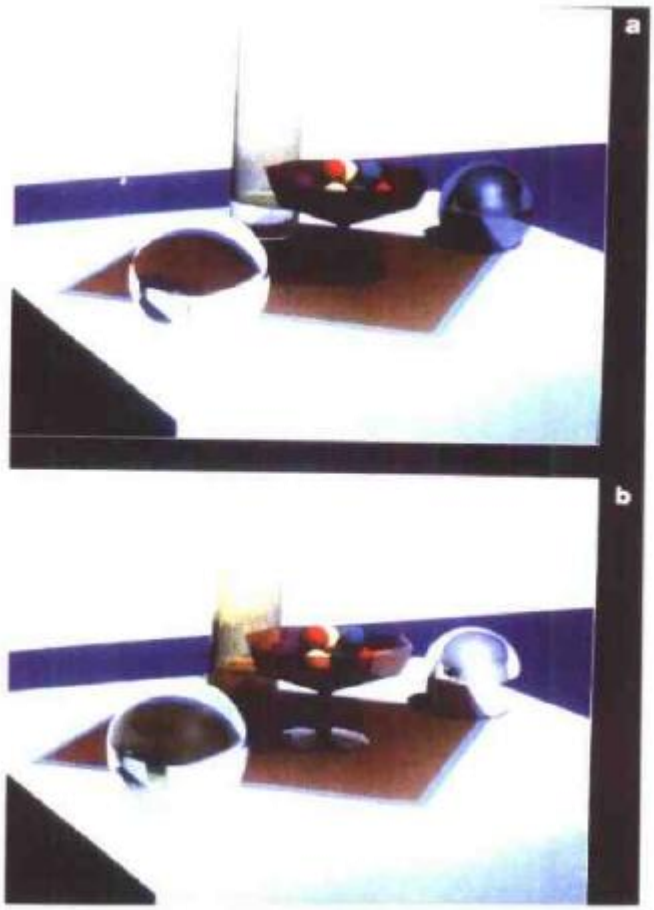


图16 静态环境。
a）由Whitted光照模型生成 b）由Hall光照模型生成（参见第5.15节）
注意图中球上光照效果的差别（由R. A. Hall和Cornell大学计算机图形实验室提供）

图17　光束跟踪。一个纹理递归映射的高反射率立方体（参见第5.16节）（由P. Heckbert提供）

图18　采用随机采样技术生成运动模糊的半影效果，注意镜面映像和阴影中亦体现出运动模糊（参见第5.16节）（版权：1984年，PIXAR，由T. Porter提供）

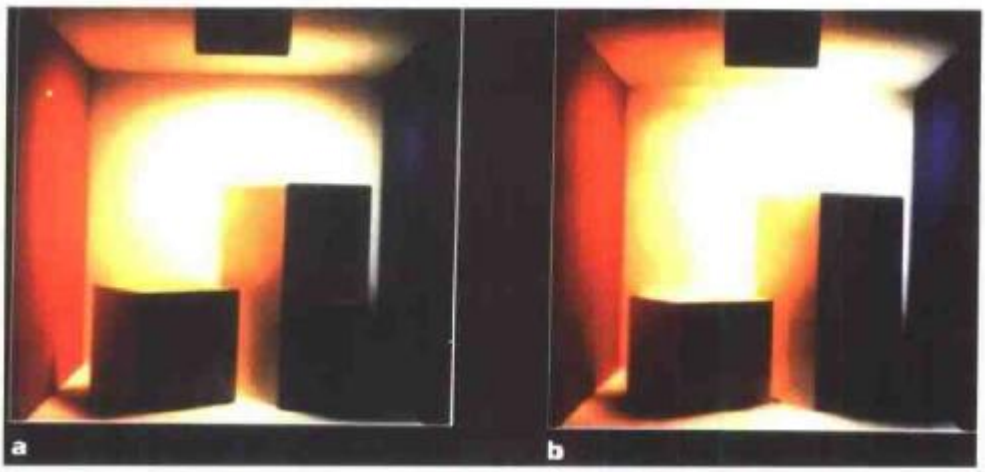


图19 辐射度。真实场景图像（图a）与计算机生成图像的比较（由G. Meyer、H. Rushmeier、M. T. Cohe、D.Greenberg、K. Torrance 及Cornell大学计算机图形实验室提供）

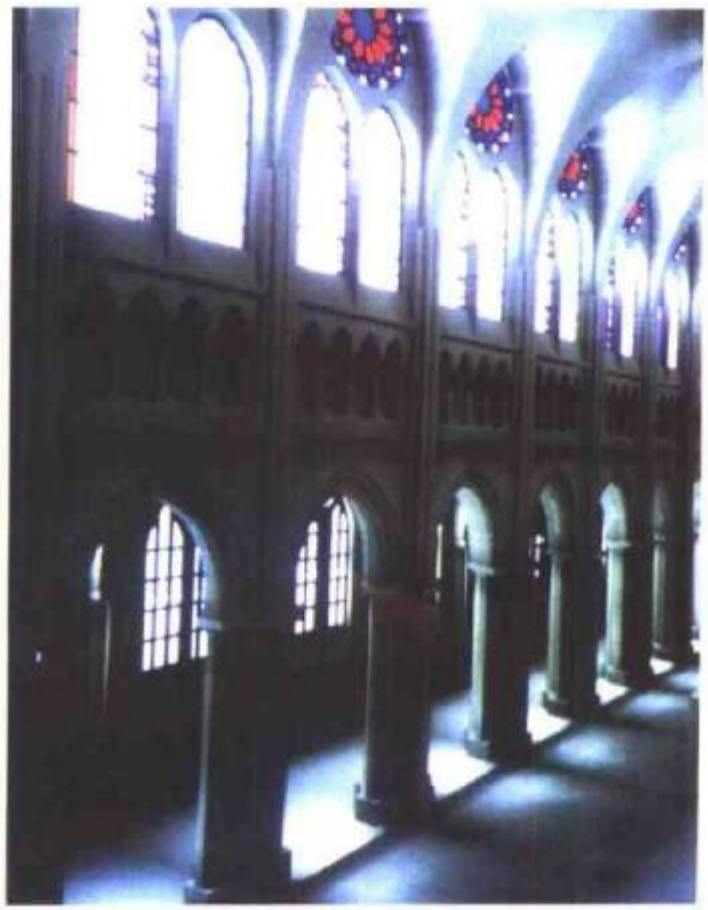


图20 Chartres大教堂。采用辐射度方法生成的Chartres大教堂的侧廊（参见第5.17节）（由3DEyc公司的J.Wallace、R. Elmquist和E. Haines提供）

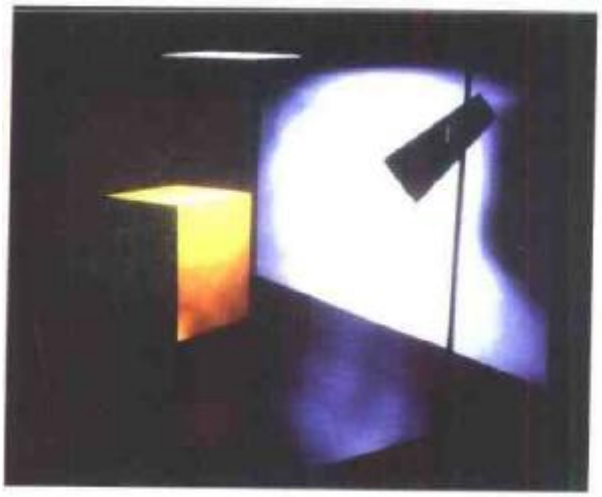


图21 包含整体漫反射和非漫反射效果的辐射度解(参见第5.17节)(由D. Immel, M. Cohen, D. Greenberg 和Cornell 大学计算机图形实验室提供)

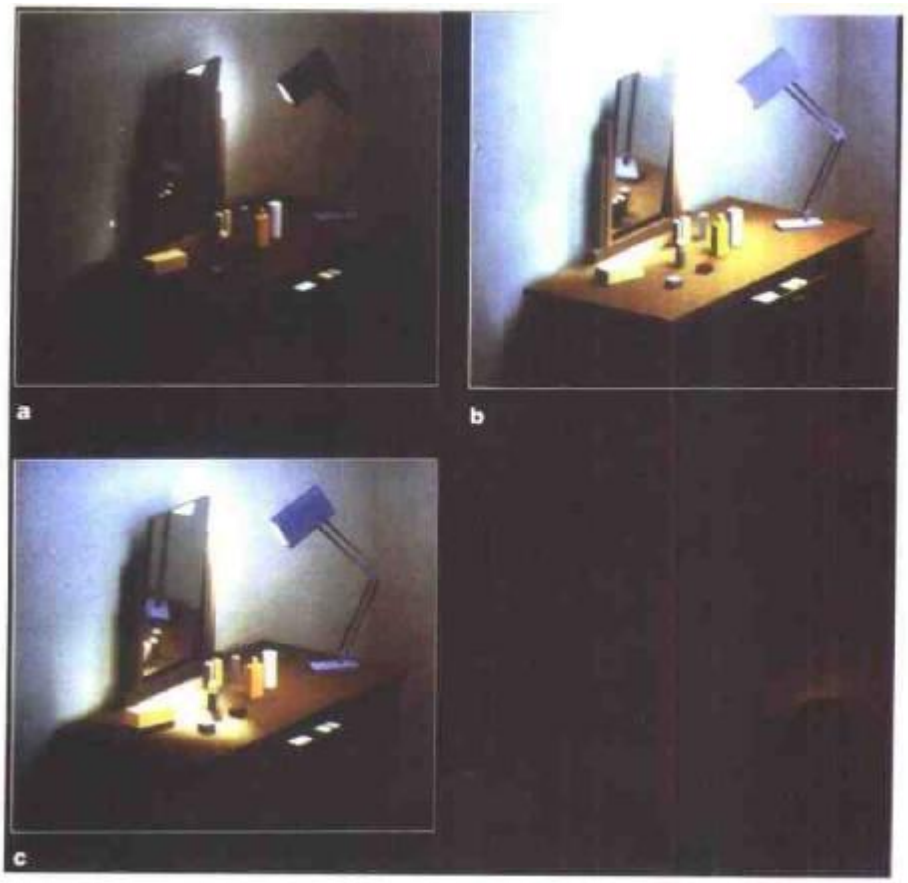


图22 两步法光照解

a) 直接光照效果 b) 直接光照加漫射光至漫射光的光能传递 c) 完整的解(参见第5.18节)
(由J. Wallace、M. Cohen、D. Greenberg 和Cornell 大学计算机图形实验室提供)

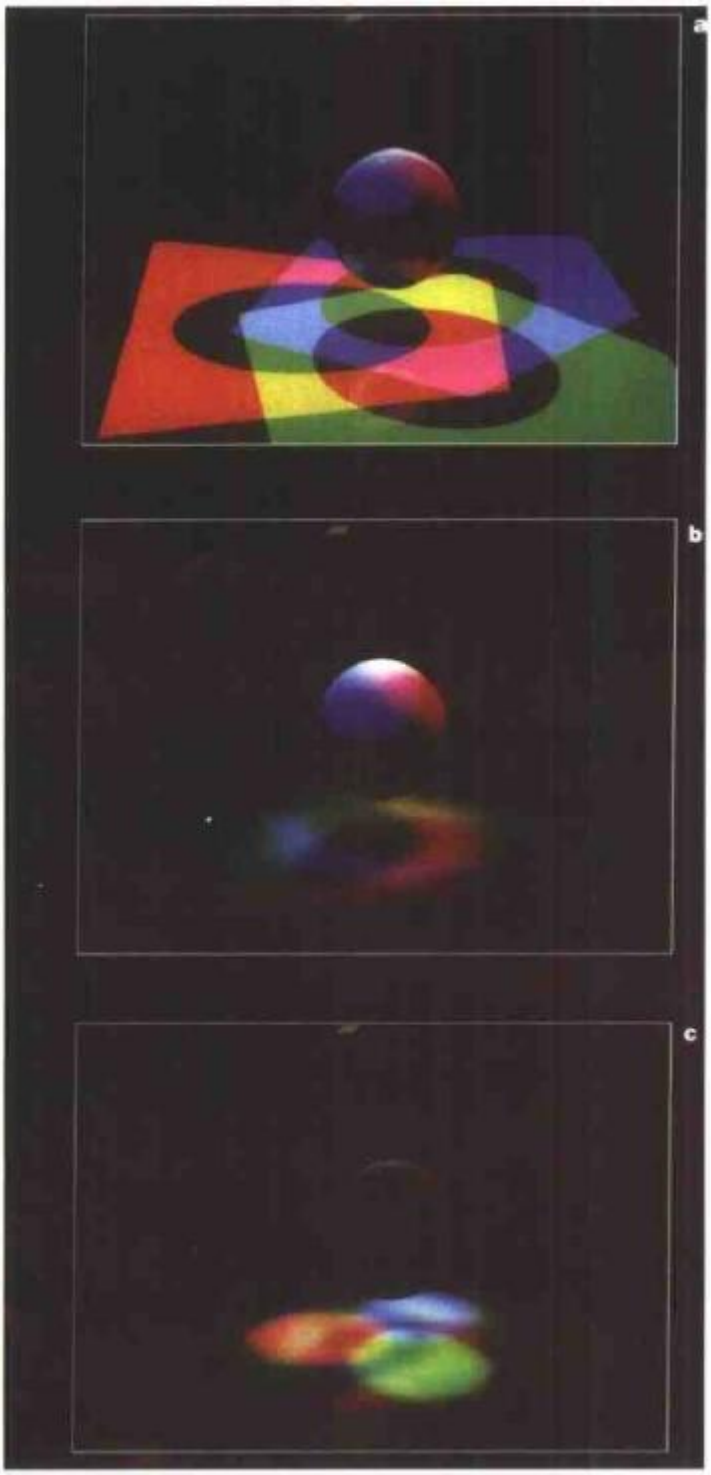


图23 折射效果，采用结合辐射度和光线跟踪的两步法算法绘制

a）传统光线跟踪方法生成的画面 b）经典辐射度方法生成的画面 c）结合辐射度方法和光线跟踪算法生成的画面，折射率取为1:1（参见第5.18节）（由IMAGIS-GRAVIR CNRS研究所的F.Sillion 提供）

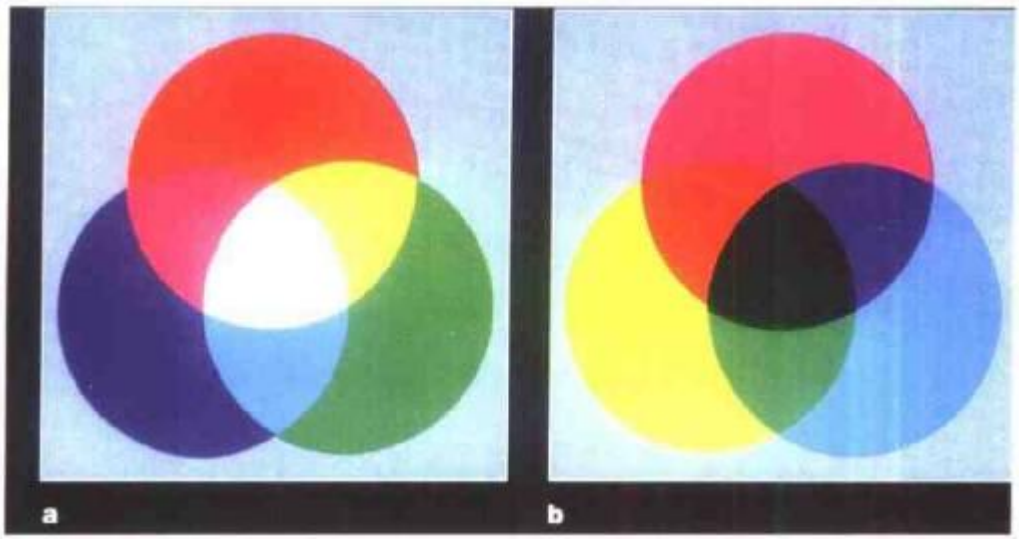


图24 颜色系统

a）RGB加色系统 b）CMY减色系统（参见第5.19节）

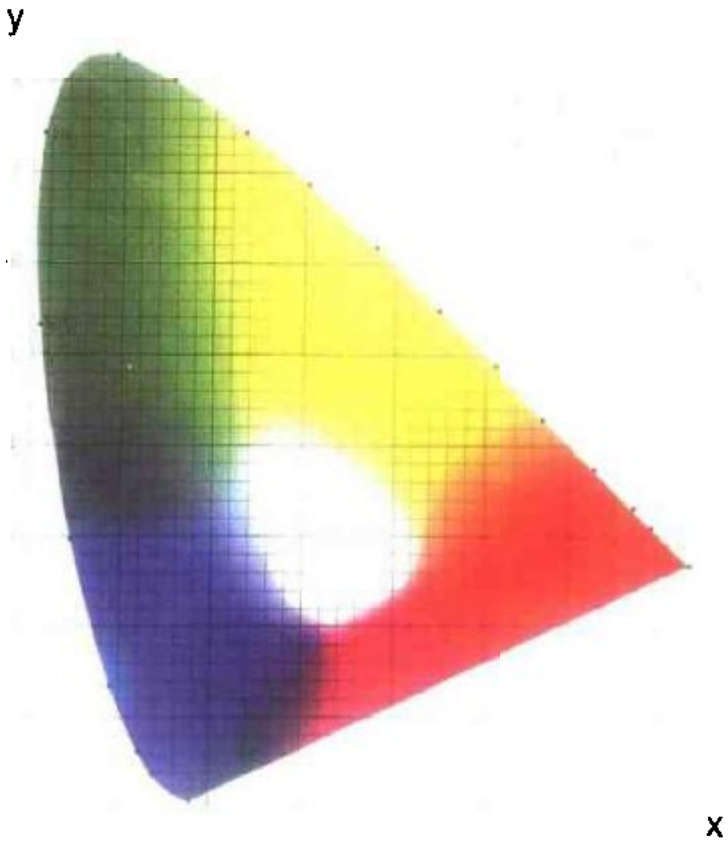


图25 1931 CIE xy色度图（参见第5.19节）（由Minolta公司的仪器系统部提供）

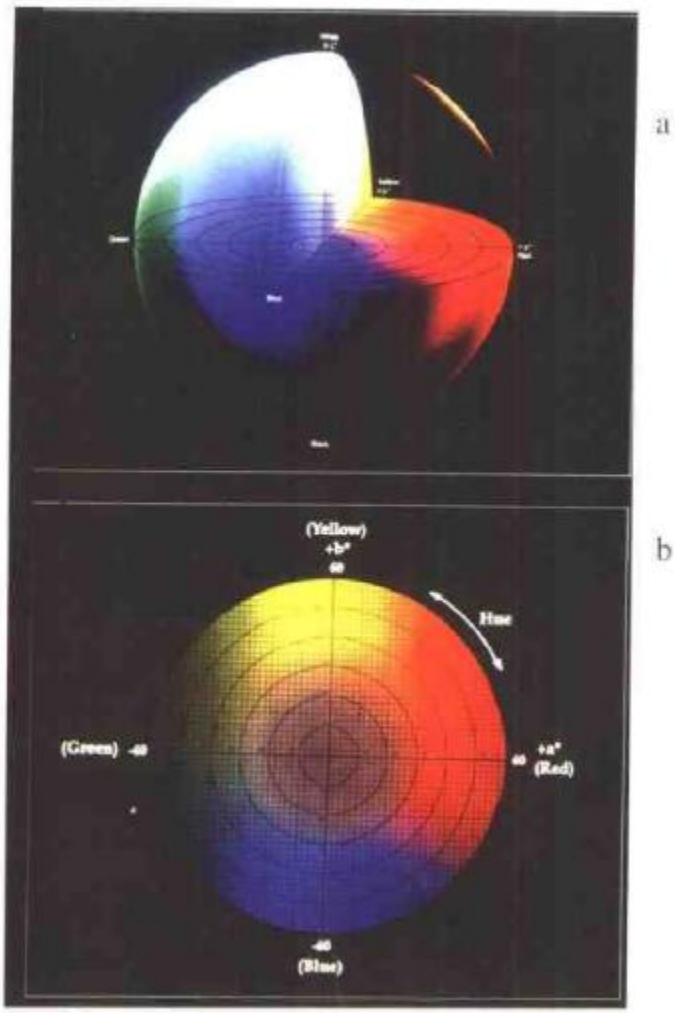


图26　CIELAB颜色空间

a）将L*a*b*颜色空间表示成一实体　b）L*a*b*颜色实体的一个截面对应一张a*b*色度图（参见第5.19节）（由Minolta公司仪器系统部提供）

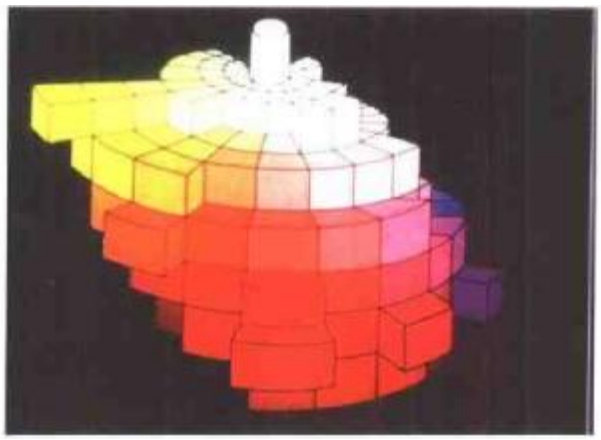


图27　颜色实体（参见第5.19节）（由Minolta公司仪器系统部提供）

图28 颜色量化

a）原始24位彩色图像 b）采用流行色算法量化 c）采用中分截断算法量化
d）采用八叉树算法量化 e）采用顺序标量算法量化
（参见第5.20节，图e）由C. Bouman提供）

图29　沙地。花木根据画家Alvy Ray Smith的算法从单个网格开始生长而成，该算法基于Paulien Hogeweg提出的数学原理。草丛由Bill Reeves运用粒子系统绘制。隐藏面消除软件由Loren Carpenter提供。图像合成软件由Tom Porter提供（参见第5.22节）（由Alvy Ray Smith提供，版权：Lucasfilm 公司）

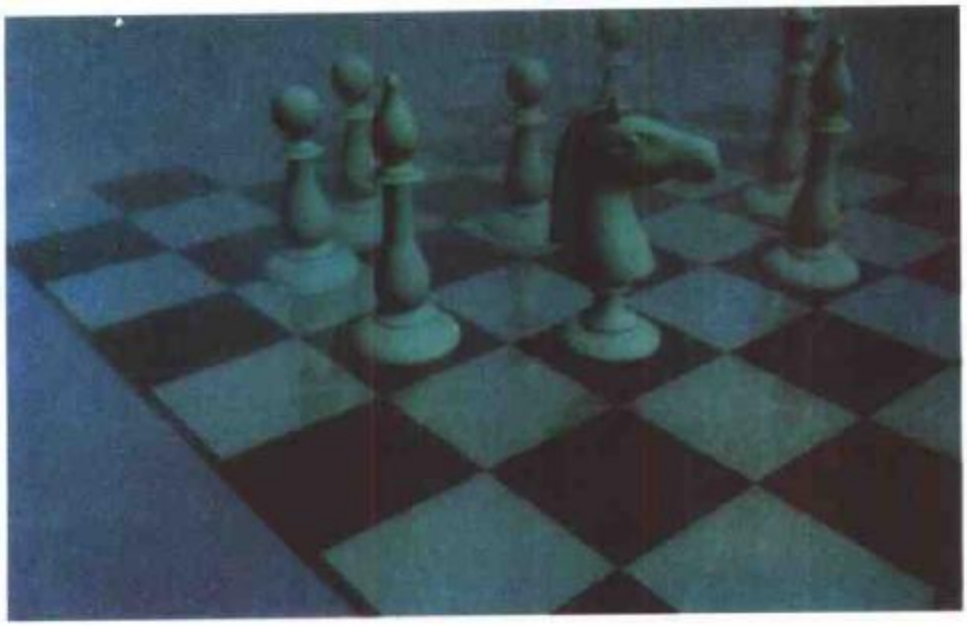


图30　朦胧中的棋子，采用扫描线算法绘制，并由数幅棋子、棋子的倒影和棋盘图像拼合而成，（由贝尔实验室的T.Whitted提供，原载于ACM TOG期刊，1982年1月第1期。

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall, Addison-Wesley, McGraw-Hill, Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum, Stroustrup, Kernighan, Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及庋藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：针对本科生的核心课程，剔抉外版菁华而成“国外经典教材”系列；对影印版的教材，则单独开辟出“经典原版书库”；定位在高级教程和专业参考的“计算机科学丛书”还将保持原来的风格，继续出版新的品种。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

“国外经典教材”是响应教育部提出的使用外版教材的号召，为国内高校的计算机本科教学度身订造的。在广泛地征求并听取丛书的“专家指导委员会”的意见后，我们最终选定了这20

多种篇幅内容适度、讲解鞭辟入里的教材，其中的大部分已经被M.I.T.、Stanford、U.C. Berkley、C.M.U.等世界名牌大学采用。丛书不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

电子邮件：hzedu@hzbook.com
联系电话：（010）68995265
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

专家指导委员会

（按姓名笔画顺序）

尤晋元	王　珊	冯博琴	史忠植	史美林
张立昂	李伟琴	李师贤	李建中	杨冬青
周克定	周傲英	孟小峰	岳丽华	范　明
高传善	梅　宏	程　旭	程时端	谢希仁
石教英	吕　建	孙玉芳	吴世忠	吴时霖
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
郑国梁	施伯乐	钟玉琢	唐世渭	袁崇义
裘宗燕	戴　葵			

译 者 序

1985年春，美国佛罗里达大学教授、IEEE CG&A学术期刊时任主编John Staudhammer先生应邀来浙江大学讲学，向我们推荐了 David Rogers编写的图形学新著《Procedural Elements of Computer Graphics》。该书取材新颖，内容全面，便于自学，立即引起了大家的浓厚兴趣。当时国内计算机图形学研究正处于起步阶段，有关计算机图形学的中文教材和参考书甚少。因此我们立即着手组织翻译。中译本取名为《计算机图形学的算法基础》，由科学出版社出版。首批5000册不到3个月即销售一空，许多没有买到该书的人只好整本地复印。该书的出版，对我国计算机图形学的发展曾起了很好的作用。不少人以这本书为向导，走上了从事图形学研究的道路。

时隔14年，机械工业出版社购买了Rogers教授新版《Procedural Elements of Computer Graphics》的翻译版权，并委托我们翻译。我们很高兴接受了这一任务。与第1版相比，新版的篇幅几乎增加了一倍。从80年代到90年代，正是计算机图形学蓬勃发展的黄金时期，光线跟踪和光能辐射度方法逐渐发展成为真实感图形绘制的两大主流技术。新版收集整理了这一期间所涌现出来的大量优秀的算法，包括各种快速光线跟踪算法，带反走样的纹理映射技术以及逐步求精辐射度方法等。此外，新版继续保持了原书概念清晰、举例详细、便于自学的风格。本书从图形学最基础的光栅扫描、区域填充、画直线和圆弧等算法开始讲解，详细介绍了线裁剪和面裁剪、凸区域裁剪和凹区域裁剪的异同，景物空间消隐算法和图象空间消隐算法的差别，具体讲述了二叉空间剖分(BSP)、八叉树等图形学中常用的数据结构。每一章都增添了新的内容，反映了在该领域图形学的最新研究成果。特别需要指出的是，图形学是一门只有通过实践才能掌握的学科，本书列举了90个例子，具体描述了各类算法的执行细节。这对初学者体会算法的基本原理，比较各类算法执行时的细微差别大有裨益。具备大学数学基础和高级程序设计语言知识的人均可自学本书。

本书的作者David Rogers是一位资深的图形学专家。他长期任职于美国海军科学院，从事计算机图形学的教学与科研。曾出版学术著作4部，主编多部。他编著的《Procedural Elements of Computer Graphics》(第1版)，曾被评为美国Top-Five(前5名)图形学教科书，并先后被翻译成六种语言文字出版。

翻译如此一本学术名著实在是一件困难的事。尽管我们有翻译本书第1版的经验，翻译工作量仍然巨大。本书的前言、第1、2章仍由石教英负责，第3～5章仍由彭群生负责。张明敏、赵友兵参与了本书第1、2章的翻译，刘新国、陈莉、冯结青分别参与了第3～5章的翻译，全书由石教英、彭群生仔细校对、修改定稿。梁友栋教授校对了第3～5节清样，并提出了许多宝贵意见。在翻译过程中，我们发现原书的一些错误之处，其中大部分为排版和印刷错误，译文中已一一作了订正。我们也曾遇到一些国内尚未见到、尚无统一译法的名词，如Haloed Line、Pencil tracing、the popularity algorithm等，我们斟酌选用了最能表达其含义的译法，详见本书索引。这些译法是否恰当，仍有待读者认可。由于译者水平和学识有限，译本中翻译不尽妥当之处在所难免，恳请读者批评指正。

译 者

2000年11月

序

我在第1版的序言中曾说过“计算机图形学已经成为成熟的学科了”，当时我们很少有人能预见到它这20余年来的发展。那个时候，光线跟踪还只是一个活跃的研究方向——但现在即便是个人电脑也可随意使用这类程序了；那时，辐射度的研究才刚刚起步，但今天的商品化绘制软件已经普遍使用了这项技术；那时，纹理是软件实现的，但现在硬件纹理加速已经很普遍了；那时，彩色图像的量化算法只在计算机图形学界中得到有限使用，但今天即便从万维网上下载图片也要用到图像量化技术。还可以找到一长串图形技术进步的例子。总之，今天的计算机图形学已经紧密地与日常生活结合在一起，它所涉及的领域遍及广告、娱乐、医学、教育、科学、工程、航空，等等。大部分计算机软件，包括使用频率最高的操作系统，都使用了图形化的用户界面。

在这一版中，我改写了第1版的很大一部分内容，这样做的结果是新版本的篇幅差不多是原来的两倍。增加的内容主要集中于图形用户界面以及直线、圆、椭圆的生成和图像压缩等算法的扩充。新版本还给出了直线和多边形裁剪的新算法，特别是在介绍经典的Cohen-Sutherland裁剪算法、中点分割算法、Cyrus-Beck裁剪算法和Sutherland-Hodgman裁剪算法的同时，重点讨论了Liang-Barsky裁剪算法和Nicholl-Lee-Nicholl裁剪算法。

可见面算法这一章在讨论二叉空间剖分(Binary Space Partitioning, BSP)、八叉树和移动立方体等算法的同时，还用专门的篇幅介绍了Appel、晕线和A—buffer等算法。可见面光线跟踪算法也有很大扩充。

绘制这一章也增加了不少内容。对基于物理的光照明模型、透明性、阴影和纹理等处理方法进行了扩充。在讨论辐射度基本理论的同时，我们还介绍了光线跟踪的最新进展，如锥光线跟踪、束光线跟踪、笔束光线跟踪和随机光线跟踪等。颜色一节补充了均匀颜色空间，并对Gamma矫正进行了更为详细的讨论。还专门讨论了彩色图像的量化和在印刷介质上的颜色再现问题。

本书适合专业程序员、工程师及科研人员使用。本书也非常适用于为高年级本科生和一年级研究生开设的重点讲授绘制技术的计算机图形学课程的教科书。如果与本书的姊妹篇《Mathematical Elements for Computer Graphics》(计算机图形学的数学基础)配合使用，课程内容就会扩大到包含各种可操纵变换、曲线和曲面等。在保留第1版例子的基础上，这一版还给出了一些新例子，共有90个。只要具有大学数学和程序设计语言基本知识的人都可以使用本书。

任何不给出算法的计算机图形学书籍都将是不完整的。本书介绍的算法采用三种方法来描述。第一种是用列表的形式给出的语言描述；第二种是详细算法的过程描述；第三种是更为形式化的伪代码描述。尽管现在许多书籍用C语言描述算法，但我不赶这个时髦。我相信只有真正实现一个算法才能对其有深刻的理解，并对算法的细枝末节有所体会，这是书本所不能涵盖的。而且只有在实现算法时才能领会实现语言所特有的效率。实现那些用伪码表示的算法比实现其他两类算法相对要更直观些。

如果没有他人的帮助，要完成一本书是不可能的。衷心感谢阅读了手稿各个部分的同事们。

其中，JohnDill和他的学生阅读了有关裁剪的第3章，并提出了许多有价值的建议。PaulHeckbert审阅了彩色图像量化和纹理这两节，他的建议为这两部分增色不少。Maureen Stone在颜色再现方面给了我很多帮助。Eric Haines对光线跟踪部分提出了相当多的建议。我还从后来的讨论中受益匪浅。John Wallace阅读了有关辐射度的部分，并就关键几点为我指点迷津。如果仍有不当之处的话，那是我的错。

我要向在法国Grenoble从事iMAGIS项目的同事François Sillion和Peter Kipfer表示特别的感谢，他们应用自行开发的层次光线跟踪软件，在极短的时间内制作了封面图像。他们热情地完成了我所要求的修改，和他们合作我感到十分愉快。

还要提一下和我合作25年之久的编辑B. J. Clark，虽然他现在已经不再从事这一方面的工作了，但想想当年我还只是一个有志于在计算机图形学方面出一本书的年轻学者时，如果没有他对我的信任，以及多年来他无微不至的鼓励和关怀，就不会有我今天的一切。还要感谢Fine Line Illustrations公司的Fred Eckardt及其同事，他们在绘制本书插图方面帮了我大忙，他们甚至惠允我使用部分原始文件。McGraw-Hill出版公司的Kari Geltemeyer. Laurie Entringer和Heather Burbridge等也做出了很大贡献。

最后要提一下我的妻子Nancy，把她放在最后当然不意味着她的贡献小。我要向她致以特别的谢意。她不仅在我写作过程中表现出了极大的耐心，而且还做了大量的誊写、编辑、校对和打字工作。我想Nancy现在绝对是一个TEX编辑高手。

David F. Rogers

第1版序

计算机图形学已成为一门成熟的学科。目前作为商品的图形软件和硬件已能方便地生成各种线画图形和自然景物的真实感图像。十年前，这种硬件和软件价格在几十万美元左右，如今只需几万美元就能买到性能优异的设备。至于性能稍低但在许多情况下仍然适用的设备则只需几千美元。所有现代科学和工程领域几乎都采用计算机图形以加强信息的传递和理解，因此当今的科学家和工程师都需具备计算机图形学的基本知识。计算机图形学也在深入到商业、医疗、广告以及娱乐等行业中。用计算机图形设备制作演讲用的幻灯片以及一般商业用的幻灯片已是司空见惯。在医疗上根据CAT扫描所采集的数据重建三维图像已变得日益普及。在电视和其他媒介上制作广告也经常利用计算机图形以及计算机动画技术，在娱乐行业中已将计算机图形应用于电子游戏和拍摄长篇故事片。从本书所提供的一些照片可以看到，计算机图形学甚至已深入到艺术领域。

本书的姐妹篇《计算机图形学的数学基础》出版已有十年之久。在此期间，光栅扫描图形学取得了巨大进展。本书将着重介绍这方面的内容。第1章是计算机图形学硬件导论，重点介绍阴极射线管显示器以及交互设备的基本概念。以后各章分别讨论光栅扫描图形学，包括直线和圆的绘制，多边形填充以及反走样（antialiasing）算法；二维和三线裁剪，包括对任意凸体的裁剪；隐藏线和隐藏面算法，包括光线跟踪算法；最后讨论绘制——生成真实感图形的“艺术”，包括建立局部和整体照明模型以及对纹理、阴影、透明以及色彩等效果的处理。本书采用的叙述方式与其姐妹篇相同。对每一问题作系统讨论之后一般都给出详细的算法和实例。

本书可作为高年级本科生或研究生第一门正式的计算机图形学课程的教材，可供讲授一个学期，重点放在光栅扫描图形学上。如果已讲授过《计算机图形学的数学基础》，那么接着讲授本书就再好不过了。作者就是这样组织教学的。如果需要在一个学期中讲授更多的内容，那么也可以将这两本书结合起来使用。具体安排如下：先讲两本书中的第1章，接着讲《计算机图形学的数学基础》一书的第2、第3两章和第4章中的几节，然后从本书的其他各章中选择一些内容进行讲授，例如第2章的2.1～2.5、2.7、2.15～2.19、2.22、2.23和2.28节，第3章中的3.1、3.2、3.4～3.6、3.9、3.11、3.15和3.16节，第4章的4.1、4.2节中有关背面剔除部分、4.3、4.4、4.7、4.9、4.11和4.13节以及第5章的5.1～5.3、5.5、5.6、5.14节等。本书也可作为专业程序员、工程师和科学家们的参考书。详细的算法和实用的例题使本书特别适合于各种水平的人员进行自学。只要具有大学数学和高级程序设计语言基础知识的人即可阅读，如有一定数据结构知识则更好，但不是必需的。

本书中用两种方式来陈述算法。第一种用叙述方式详细说明算法的每一个步骤。第二种较为形式化，采用一种算法“语言”来陈述。由于计算机图形学的应用十分广泛，在选择一种适当的“算法语言”时，作者再三斟酌，征求过许多同事的意见，但他们的看法均不一致。计算机科学界的同行们一般喜欢使用PASCAL，但也有相当一部分更偏爱C语言。工业界则一般爱用FORTRAN语言，这样易于和已有的软件兼容。作者本人喜欢用BASIC，因为它的

用法简单。为了求同存异，本书中的详细算法均用伪代码写成。使用伪代码是基于作者对很多班级讲授计算机图形学课程的经验得到的，因为这些班级中的学生并不掌握一种共同的程序设计语言。伪代码易于翻译成任何一种通用程序设计语言，关于伪代码的介绍详见附录。本书中给出的所有伪代码算法或者直接写成并验证，或者从已运行的程序导出。这些算法均用通用程序设计语言编写，并在Apple IIe上用BASIC、在IBM 3400上用PL1以及在规模介于这两者之间的的各种机器上用不同语言实现过。作者可提供一套演示程序。

简单介绍本书的产生过程是饶有兴趣的。它采用计算机排版，所用的TEX排版系统是由弗吉尼亚州雷斯顿市的TYX公司提供的。书稿直接由手写本经计算机处理后得到。长条校样和供编辑以及页面排版用的两套版面校样均由激光打印机印出。最后供插入的艺术图片的复制本是在照排机上产生的。TYX公司的Jim Gauthier和Mark Hoffman为解决在使用TEX系统过程中所发现的许多细小问题作出了很大努力，在此表示深切的谢意。对Louise Bohrer和Beth Lessels在整理手稿时所做的出色工作深表谢意。McGraw-Hill出版公司的出版工作一贯是高质量的，本书由David Damstra和Sylvia Warren负责编辑。

一本书的出版总是在很多人的帮助下完成的。本书是根据1978年以来在Johns Hopkins大学应用物理实验室中心为研究生课程所准备的材料编写而成。感谢听过我这门课和其他课程的一些学生，我从他们那里学到很多东西。感谢阅读本书原始提纲并提出有价值建议的Turner Whitted。感谢我的同事Pete Atherton、Brian Barsky、Ed Catmull、Rob Cook、John Dill、Steve Hansen、Bob Lewand、Gary Meyer、Alvy Ray Smith、Dave Warn以及Kevin Weiler，他们仔细地阅读了手稿的一些章节，提出了一些意见和建议，从而使本书更为完善。感谢我的同事Linda Rybak和Linda Adlum，他们读完全部手稿并检查了所有例子。感谢我的3位学生：Bill Meier实现了Roberts算法，Gary Boughan首先提出凸性检查（见3.7节），Norman Schmidt首先提出多边形分割技术（见3.8节）。感谢Mark Meyerson实现了这一分割算法并给出算法的数学基础。特别感谢Lee Billow和John Metcalf准备了所有线画插图。

深切感谢Steve Satterfield全部读完800页手稿并提出有益的意见。

还要特别感谢我的长子Stephen，他实现了第4章中的所有隐藏面算法以及本书其他许多算法。我们之间多次活跃的讨论使很多重要观点更为清晰。

最后，要特别提一提我的妻子Nancy以及另外两个孩子Karen和Ransom，他们面对自己的丈夫和父亲在长达一年半之久的时间里几乎每晚和周末都在办公室里度过面毫无怨言，这是一种真正的支持！谢谢！

David F. Rogers

目　录

第3章 裁　剪

裁剪是从数据集合中抽取所需信息的过程，它是计算机图形学中许多重要问题的基础。裁剪最典型的用途是确定场景或画面中位于给定区域之内的部分。这一区域称为裁剪窗口。除了图形裁剪外，第2章中已用裁剪方法处理图形反走样问题。在以后各章中将陆续看到裁剪在消除隐藏线、消除隐藏面、阴影、纹理等算法中也是有用的工具。值得一提的是用本章中介绍的裁剪概念和方法，还可以构造更高级的裁剪算法，即实现多面体对多面体的裁剪。这类算法可以用来在简单实体造型系统中执行布尔运算，例如在简单立方体和二次曲面体之间求并和求交。此外裁剪还可用于复制、移动或删除画面中某一部分。在窗口系统中，这一功能称之为剪贴。由于篇幅所限，对这些应用本书中不作详述。

裁剪算法有二维的和三维的。裁剪对象和裁剪区域可以是规则的，也可以是不规则的。裁剪算法可以用硬件实现，也可以用软件实现。由于软件实现常达不到实时应用环境提出的速度要求，因此二维和三维裁剪算法通常由相应的硬件或固件实现。其中裁剪区域通常只限于规则区域或规则体。但采用超大规模集成电路(*VLSI*)硬件可以实时地完成对更为一般规则的或不规则的区域和体的实时裁剪。

3.1 二维裁剪

图3-1给出一幅二维画面和一个规则裁剪窗口。规则裁剪窗口为由左(L)、右(R)、上(T)、下(B)四边定义的矩形，它们分别与物体空间(object space)或显示设备的坐标轴平行。图形裁剪旨在决定画面中哪些点、线段或部分线段位于裁剪窗口之内。上述点、线段或部分线段被保留用于显示，而其他的则被抛弃。在一个典型的场景之中，需要对大量的点、线段进行裁剪，因此裁剪算法的效率十分重要。注意在许多情形中，绝大多数的点和线段要么完全位于窗口之内，要么完全位于窗口之外，因此快速接受图3-1中的线段ab、点p或抛弃线段ij、点q对提高裁剪算法的效率是至关重要的。

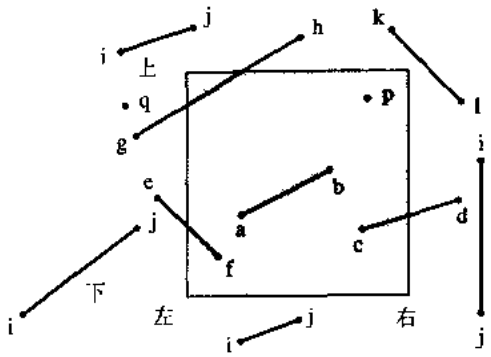


图3-1 二维裁剪窗口

3.1.1 简单可见性判别算法

点位于窗口之内的等价条件是:

$$x_L \leqslant x \leqslant x_R \quad 和 \quad y_B \leqslant y \leqslant y_T$$

其中等号表示窗口边界上的点被认定为属于窗口区域之内。

如果一线段的两端点均位于窗口内，则该线段也位于窗口内并可见，如图3-1中的线段ab。但是当线段两端点均在窗口之外时，该线段不一定全部在窗口的外面，例如图3-1中的gh。若线段的两端点同时位于窗口左侧、右侧、上面或下面，则该线段必定全部在窗口之外，因而不可见。通过这种检查可剔除图3-1中所有标记为ij的线段，但不能剔除部分可见线段gh，也不能剔除完全不可见线段kl。

设a、b为一线段的两端点，则一个能判别完全可见线段和绝大多数不可见线段的算法可表述如下。

```
simple visibility algorithm
a and b are the end points of the line, with components x and y
    for each line
    Visibility = True
    check for totally invisible lines
    if both end points are left, right, above or below the window, the line
      is trivially invisible
    if x_a < x_L and x_b < x_L then Visibility = False
    if x_a > x_R and x_b > x_R then Visibility = False
    if y_a > y_T and y_b > y_T then Visibility = False
    if y_a < y_B and y_b < y_B then Visibility = False
    if Visibility <> False then    avoid the totally visible calculation
        check if the line is totally visible
        if any coordinate of either end point is outside the window, then
          the line is not totally visible
        if x_a < x_L or x_a > x_R then Visibility = Partial
        if x_b < x_L or x_b > x_R then Visibility = Partial
        if y_a < y_B or y_a > y_T then Visibility = Partial
        if y_b < y_B or y_b > y_T then Visibility = Partial
    end if
    if Visibility = Partial then
        the line is partially visible or diagonally crosses a corner invisibly
        determine the intersections and the visibility of the line
    end if
    if Visibility = True then
        line is totally visible — draw line
    end if
    line is invisible
    next line
finish
```

在上述算法中，x_L、x_R、y_T、y_B分别为裁剪窗口左、右、上、下四边的x和y坐标值。执行可见或不可见诸项检查的先后顺序是无关紧要的。有些线段需进行全部四项检查才能确定其为可见或完全不可见，有的只需作一步检查就够了。在算法实现中先执行完全可见线段检查还是先执行完全不可见线段检查，同样也是无关紧要的。但是，线段与窗口边界的求交运算

量较大，因此要安排在最后阶段执行。

3.1.2 端点编码

完全可见线段检查和上面给出的完全不可见线段的区域检查可采用编码方法实现。这一方法由Dan Cohen 和Ivan Sutherland提出。它采用四位数码来标识线段的端点位于九个区域中的哪一个区域内。四位数码方法如图3-2所示。最右边的位是第一位，编码规则如下:

第一位置1——如果线段端点位于窗口左侧

第二位置1——如果线段端点位于窗口右侧

第三位置1——如果线段端点位于窗口下面

第四位置1——如果线段端点位于窗口上面

否则相应位置零。由编码规则可知，若线段两端点的编码均为零，即两端点均在窗口之内，故线段可见。完全不可见线段也可容易地由两端点的编码判定。考虑下面的逻辑“与”操作的真值表:

True and False	→	False		1 and 0	→ 0
False and True	→	False	False = 0 →	0 and 1	→ 0
False and False	→	False		0 and 0	→ 0
True and True	→	True	True = 1	1 and 1	→ 1

将线段两端点的编码逐位取逻辑“与”，若结果非零，则该线段必为完全不可见，因而可立即抛弃。表3-1中所示的例子可以进一步验证。注意，在表3-1中，当两端点编码逻辑“与”非零时，线段固然完全不可见，但当逻辑“与”为零时，线段可能完全或部分可见，也可能完全不可见，因此要判别完全可见线段时，还需要对两端点的编码分别进行检查。

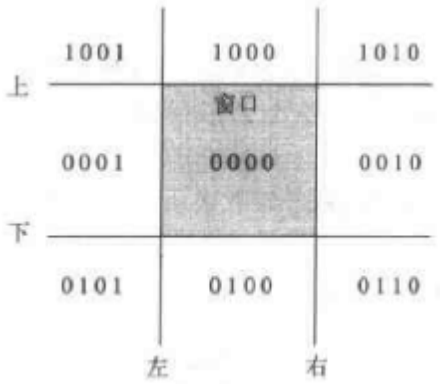


图3-2 线段端点的区域编码

表3-1 端点编码

线段(见图3-1)	端点编码(见图3-2)	逻 辑 与	注 释
ab	0000 0000	0000	完全可见
ij	0010 0110	0010	完全不可见
ij	1001 1000	1000	完全不可见

（续）

线段(见图3-1)	端点编码(见图3-2)	逻 辑 与	注　释
ij	0101 0001	0001	完全不可见
ij	0100 0100	0100	完全不可见
cd	0000 0010	0000	部分可见
ef	0001 0000	0000	部分可见
gh	0001 1000	0000	部分可见
kl	1000 0010	0000	完全不可见

当机器提供位操作子程序时，端点编码检查很容易实现。当不使用位操作时，编码检查也可用下面的算法实现。

end point code algorithm

P_1 *and* P_2 *are the end points of the line*

x_L, x_R, y_T, y_B *are the left, right, top and bottom window coordinates*

calculate the end point codes

put the codes for each end into 1×4 *arrays called* P1code *and* P2code

first end point: P_1

if $x_1 < x_L$ then P1code(4) = 1 else P1code(4) = 0
if $x_1 > x_R$ then P1code(3) = 1 else P1code(3) = 0
if $y_1 < y_B$ then P1code(2) = 1 else P1code(2) = 0
if $y_1 > y_T$ then P1code(1) = 1 else P1code(1) = 0

second end point: P_2

if $x_2 < x_L$ then P2code(4) = 1 else P2code(4) = 0
if $x_2 > x_R$ then P2code(3) = 1 else P2code(3) = 0
if $y_2 < y_B$ then P2code(2) = 1 else P2code(2) = 0
if $y_2 > y_T$ then P2code(1) = 1 else P2code(1) = 0

finish

如果先已判定完全可见线段和显然不可见线段，则剩下线段两端点编码的逻辑与均为零，然后转入求交处理，以决定是否是部分可见线段。当然，求交程序应能判别其中可能出现的完全不可见线段。

两线段的求交可采用参数表示形式或非参数表示形式。通过$P_1(x_1, y_1)$和$P_2(x_2, y_2)$两点的直线的显式方程为

$$y = m(x-x_1) + y_1 \quad \text{或} \quad y = m(x - x_2) + y_2$$

其中

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

为直线的斜率。直线与窗口诸边的交点为

左: x_L, $y = m(x_L - x_1) + y_1$, $m \neq \infty$

右: x_R, $y = m(x_R - x_1) + y_1$, $m \neq \infty$

上: y_T, $x = x_1 + (1/m)(y_T - y_1)$, $m \neq 0$

下: y_B, $x = x_1 + (1/m)(y_B - y_1)$, $m \neq 0$

例3.1说明了直线和各窗口边的显式求交方法。

例3.1 直线和窗口显式求交。

考虑图3-3所示的裁剪窗口和被裁剪线段，其中通过$P_1(-\frac{3}{2}, \frac{1}{6})$和$P_2(\frac{1}{2}, \frac{3}{2})$的直线的斜率为

$$m=\frac{y_2-y_1}{x_2-x_1}=\frac{\frac{3}{2}-\frac{1}{6}}{\frac{1}{2}-(-\frac{3}{2})}=\frac{2}{3}$$

与窗口各边的交点为

左边: $x=-1 \quad y=\frac{2}{3}[-1-(-\frac{3}{2})]+\frac{1}{6}=\frac{1}{2}$

右边: $x=1 \quad y=(\frac{2}{3})[1-(-\frac{3}{2})]+\frac{1}{6}=1\frac{1}{6}$

上边: $y=1 \quad x=-\frac{3}{2}+\frac{3}{2}[1-\frac{2}{6}]=-\frac{1}{4}$

下边: $y=-1 \quad x=-\frac{3}{2}+\frac{3}{2}[-1-\frac{1}{6}]=-1\frac{3}{4}$

注意此直线段和窗口右边界的交点P_2''超出了直线段的端点。

类似地，对通过$P_3(-\frac{3}{2},-1)$和$P_4(\frac{3}{2}, 2)$的直线有

$$m=\frac{y_2-y_1}{x_2-x_1}=\frac{2-(-1)}{\frac{3}{2}-(-\frac{3}{2})}=1$$

和

左边: $x=-1 \quad y=(1)[-1-(-\frac{3}{2})]+(-1)=-\frac{1}{2}$

右边: $x=1 \quad y=(1)[1-(-\frac{3}{2})]+(-1)=\frac{3}{2}$

上边: $y=1 \quad x=-\frac{3}{2}+(1)[1-(-1)]=\frac{1}{2}$

下边: $y=-1 \quad x=-\frac{3}{2}+(1)[-1-(-1)]=-\frac{3}{2}$ □

在设计一个有效的裁剪算法时，需要考虑一些特殊情形的处理。若直线的斜率为无穷大，则直线平行于窗口的左边和右边，故仅需检查直线与上、下两边的交点；同样，若直线斜率

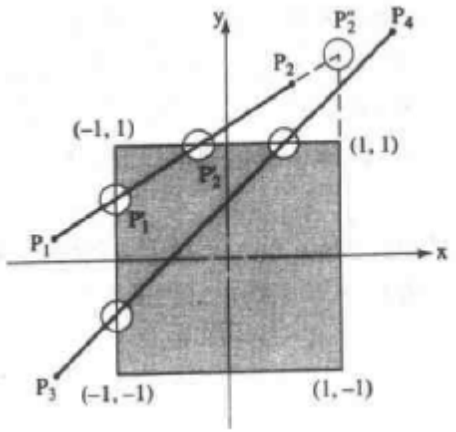


图3-3 参数形式的二维裁剪

为零，则它平行于窗口的上、下两边，这时仅需检查直线与左、右两边的交点。最后，若直线上有一个端点的编码为零，则该点位于裁剪窗口之内，此时直线与窗口只有一个交点。Rogers [Roge85]依此提出了一个有效的显式裁剪算法。

3.2 Cohen-Sutherland线段细分裁剪算法

Dan Cohen 和 Ivan Sutherland所设计的裁剪算法利用端点编码性质简单地决定是否接受或抛弃一条线段。一条线段如果不能被简单地接受或抛弃，便在窗口的边界处被分割为两段。这个算法并不检查交点是否位于窗口的边界内，而是根据每一段的端点编码决定它们的取舍。观察图3-3中的P_1P_2，立即发现这一简单规则实现起来有困难。当用窗口左边对P_1P_2进行裁剪时，得到两段P_1P_1'和$P_1'P_2$。由每段两端点的编码可推知它们均可能部分可见，因此既不能将任一段作为不可见线段而抛弃，也无法将任一段作为可见线段而接受。Cohen-Sutherland算法的关键在于总是要得知位于窗口之外的一个端点。这样位于此端点至交点之间的区段必为不可见，故可抛弃。然后此算法继续处理线段被裁剪后的剩余部分，此时取交点来代替被裁剪线段的一个端点。Cohen-Sutherland算法可简述如下。

对于每条窗口边：

检查线段P_1P_2是否为完全可见段或可以抛弃的显然不可见段。

若P_1在窗口外，继续执行算法；否则交换P_1和P_2。

用P_1P_2和窗口边的交点取代P_1点。

例3.2详细说明了算法的实现过程。

例3.2　Cohen-Sutherland裁剪算法。

仍考虑图3-3所示线段P_1P_2和裁剪窗口，线段端点$P_1(-\frac{3}{2}, \frac{1}{6})$和$P_2(\frac{1}{2}, \frac{3}{2})$的编码分别为(0001)和(1000)。两端点编码不全为0，逻辑与结果为0。因此该线段既非完全可见，也不是显然不可见。比较两端点编码的第一位可以发现，该线段跨越窗口的左边界，并且端点P_1位于窗外。

线段与窗口左边($x=-1$)的交点为$P_1'(-1, \frac{1}{2})$。用P_1'取代P_1得到新线段$P_1(-1, \frac{1}{2})P_2(\frac{1}{2}, \frac{3}{2})$。

现在端点P_1、P_2的编码分别为(0000)和(1000)。新线段仍非完全可见或显然不可见。比较端点编码的第二位，发现线段并不跨越窗口右边界，转而考察窗口底部边界。

端点P_1和P_2的编码仍然为(0000)和(1000)，此线段既非完全可见，也不是完全不可见。比较端点编码的第三位，发现线段并不跨越窗口底部边界，转而考察窗口顶部边界。

端点P_1和P_2的编码仍然为(0000)和(1000)，此线段既非完全可见，也不是完全不可见。比较端点编码的第四位，发现该线段跨越窗口顶部边界，P_1不在窗口外，交换P_1和P_2得到一新线段$P_1(\frac{1}{2}, \frac{3}{2})P_2(-1, \frac{1}{2})$。

此线段同窗口顶边界($y=1$)的交点是$P_1'(-\frac{1}{4}, 1)$。用P_1'取代P_1得到新线段$P_1(-\frac{1}{4}, 1)P_2(-1, \frac{1}{2})$。

端点P_1和P_2的编码分别为(0000)和(0000)。该线段完全可见。

裁剪过程结束。

画出线段(图3-3中的$P_1'P_2'$)。　□

最初实现的Cohen-Sutherland算法十分简洁有效（见[Newm73,79]）。后来Duvanenko

进一步研究了该算法，经过优化和代码重写，将算法的效率提高了50~80% [Duva90a,90b,93]。改进主要是基于以下的观察结果：

裁剪后的线段和未裁剪前的线段具有相同的斜率，因此斜率及其倒数仅需在主循环的外面计算一次。

如果P_1不位于窗口外，交换P_1、P_2及其编码，线段与窗口边界的交点成为新的P_1，但P_2保持不变。所以仅有端点P_1的编码需重新计算。

图3-2表明，无论线段的端点位于九个区域中哪两个区域，在确定它是否完全可见或完全不可见时，至多需要比较端点编码中的两位。并且，如果一条部分可见的线段已经为窗口某一边界分割，那么在判断该线段是否跨越别的窗口边界时，就不必再去比较对应该边界的那一位编码，而仅需再比较一位就可以判断该线段是否跨越别的边界。编码的上述性质使显然不可见线段的测试更为简便。

下面列出一个用伪代码描述的优化二维裁剪算法。为了使算法描述较为简洁，其中只采用了部分优化技术，余下部分将留给学生作为练习。由于部分过程重复出现，因此算法描述中采用了子程序。

Cohen–Sutherland two-dimensional clipping algorithm

Window *is a* 1×4 *array containing the edges* x_L, x_R, y_B, y_T, *respectively*
P_1 *and* P_2 *are the line end points with components* P_1x, P_1y, P_2x, P_2y
P1code *and* P2code *are* 1×4 *arrays containing the end point codes*
Iflag *classifies the slope;* -1 *vertical,* 0 *horizontal,* $+1$ *other*

```
call Endpoint(P1, Window; P1code)
call Endpoint(P2, Window; P2code)
call Sum(P1code; Sum1)
call Sum(P2code; Sum2)
call Visible(P1code, P2code, Sum1, Sum2; Vflag)
find the trivially visible and invisible lines immediately and exit
if Vflag = yes then  draw line and exit
if Vflag = no then  exit without drawing line
the line is partially visible
calculate the slope which does not change as the line is clipped
Iflag = 1        initialize Iflag
check for vertical and horizontal lines
if P2x = P1x then
    Iflag = −1        vertical line
else if P2y = P1y then
    Iflag = 0        horizontal line
else
    Slope = (P2y − P1y)/(P2x − P1x)        calculate slope
end if
clip a partially visible line
while Vflag = partial
    for each window edge
    for i = 1 to 4
```

```
            check to see if the line crosses the edge
            if P1code(5 − i) <> P2code(5 − i) then
                if P1 is inside, swap the end points, Pcodes and Sums
                if P1code(5 − i) = 0 then
                    Temp = P1
                    P1 = P2
                    P2 = Temp
                    Tempcode = P1code
                    P1code = P2code
                    P2code = Tempcode
                    Tempsum = Sum1
                    Sum1 = Sum2
                    Sum2 = Tempsum
                end if
                find intersections with the window edges
                select the appropriate intersection routine
                check for a vertical line
                if Iflag <> −1 and i ≤ 2 then        left and right edges
                    P1y = Slope*(Window(i) − P1x) + P1y
                    P1x = Window(i)
                    call Endpoint(P1, Window; P1code)
                    call Sum(P1code; Sum1)
                end if
                if Iflag <> 0 and i > 2 then        bottom and top edges
                    if Iflag <> −1 then        not vertical line
                        P1x = (1/Slope)*(Window(i) − P1y) + P1x
                    end if
                    P1y = Window(i)      if vertical line P1x is unchanged
                    call Endpoint(P1, Window; P1code)
                    call Sum(P1code; Sum1)
                end if
                call Visible(P1code, P2code, Sum1, Sum2; Vflag)
                if Vflag = yes then  draw line and exit
                if Vflag = no then  exit without drawing line
            end if
        next i
    end while
finish
subroutine module to determine the visibility of a line segment
subroutine Visible(P1code, P2code, Sum1, Sum2; Vflag)
P1code and P2code are 1 × 4 arrays containing the end point codes
Sum1 and Sum2 are the sums of the elements of the end point codes
Vflag is a flag set to no, partial, yes as the line segment is totally
  invisible, partially visible, or totally visible
    assume the line is partially visible
    Vflag = partial
    check if the line is totally visible
    if Sum1 = 0 and Sum2 = 0 then
        Vflag = yes
```

```
    else
        check if the line is trivially invisible
        call Logical(P1code, P2code; Inter)
        if Inter <> 0 then Vflag = no
    end if
    the line may be partially visible
return
subroutine module to calculate the end point codes
subroutine Endpoint(P, Window; Pcode)
Px, Py are the x and y components of the point P
Window is a 1 × 4 array containing the edges xL, xR, yB, yT, respectively
Pcode is a 1 × 4 array containing the end point code
    determine the end point codes
    if Px < xL then Pcode(4) = 1 else  Pcode(4) = 0
    if Px > xR then Pcode(3) = 1 else  Pcode(3) = 0
    if Py < yB then Pcode(2) = 1 else  Pcode(2) = 0
    if Py > yT then Pcode(1) = 1 else  Pcode(1) = 0
return
subroutine to calculate the sum of the endpoint codes
subroutine Sum(Pcode; Sum)
Pcode is a 1 × 4 array containing the end point code
Sum is the element-by-element sum of Pcode
    calculate the sum
    Sum = 0
    for i = 1 to 4
        Sum = Sum + Pcode(i)
    next i
return
subroutine module to find the logical intersection
subroutine Logical(P1code, P2code; Inter)
P1code and P2code are 1 × 4 arrays containing the end point codes
Inter is the sum of the bits for the logical intersection
    Inter = 0
    for i = 1 to 4
        Inter = Inter + Integer((P1code(i) + P2code(i))/2)
    next i
return
```

Duvanenko、Gyurcsik和Robbins(见[Duva93])证明，判断一条线段完全可见并不需要做4次比较，而只需3次就够了。方法是不直接比较端点和窗口边界的值，而先在两个端点之间做比较，然后再和窗口边界做比较。算法如下：

```
Duvanenko et al.'s trivial visible algorithm for the left
  and right edges.
A similar algorithm is required for the bottom and top edges.
P1x and P2x are the line end point x components
```

```
x_L, x_R are the left and right window edges
    if P1x < P2x then
        if P1x < x_L then return not trivially visible
        if P2x > x_R then return not trivially visible
    else
        if P2x < x_L then return not trivially visible
        if P1x > x_R then return not trivially visible
    end if
    if get here then trivially visible
finish
```

Duvanenko、Gyurcsik 和Robbins(见[Dnva93])还证明，判断一条显然不可见的线段，只需要做两次比较。方法是调整两端点对窗口边界进行比较的次序。具体算法如下:

```
Duvanenko et al.'s trivial invisible algorithm for the left
  and right edges.
A similar algorithm is required for the bottom and top edges.
P1x and P2x are the line end point x components
x_L, x_R are the left and right window edges
    if P1x < x_L then
        if P2x < x_L then return trivially invisible
    else
        if P2x > x_R then
            if P1x > x_R then return trivially invisible
        end if
    end if
finish
```

他们称在大多数情况下，这一算法的效率高于先取P_1x和x_R进行比较，再取P_2x进行比较的做法。

3.3 中点分割算法

在Cohen-Sutherland算法中，需要计算裁剪线段和窗口各边的交点。如果不断地在中点处将线段一分为二，则上述直接求交点的计算过程可以用二分查找取代。这一算法是Sproull 和 Sutherland[Spro86] 为便于硬件实现而提出的，是Cohen-Sutherland算法实现的一种特例。用硬件实现中点分割既快又有效，这是因为整个过程可以并行处理，而且用硬件执行加法和除2运算非常快。事实上，用硬件除2不过是将数码右移一位而已。例如十进制数6可表示成四位二进制数0110，右移一位后得0011，它对应于十进制数3 = $6/2$ 。尽管该算法是为了硬件实现而设计的，但如果软件系统提供移位操作，该算法也可以用软件予以实现。

算法采用线段端点编码和相应的检查方法，首先判定完全可见线段和显然不可见线段，如图3-4中的线段 a 和线段 b。不能用上述方法立即判定的线段，如图3-4中的线段 $c-g$，被分割成相等的两段。然后对每一小段重复上述检查，直至找到每段与窗口边的交点或分割子段的长度充分小，可以视为一点时为止，例如图3-4中的线段 f，然后决定该点的可见性。实际上，这相当于采用对数查找法求交，分割次数最多不超过线段端点的表示精度(机器位数)。

为了进一步说明这一方法，考虑图3-4中的线段 c 和线段 f。尽管线段 f 不可见，但由于它跨过窗口一角，故不能立即判为显然不可见段。将线段 f 在中点P_{m_1}处一分为二，则立即可判定$P_{m_1}P_2$不可见并将其抛弃；但另外一段$P_{m_1}P_1$仍跨窗口角点而不能直接判定。再取其中点P_{m_2}，分割$P_{m_1}P_1$，可知$P_{m_2}P_1$不可见并将其抛弃。继续分割线段剩余部分$P_{m_1}P_{m_2}$，直至在给定精度下求得它与窗口右边延伸段的交点。检查此交点，得知其不可见。因此整个线段 f 均不可见。

对于图3-4中线段c，从它的端点编码可知它既不是完全可见段，也不是可立即抛弃的显然不可见段。在中点P_{m_1}处将线段c一分为二，两条子线段的情形仍然相同。暂不考虑$P_{m_1}P_1$，在P_{m_2}处进一步分割$P_{m_1}P_2$。从图中可看出，子段$P_{m_1}P_{m_2}$完全可见，而$P_{m_2}P_2$部分可见。因此$P_{m_1}P_{m_2}$可立即画出。然而这样处理将导致线段的可见部分被划分成一系列的可见小段，并以分段方式画出。这样绘制的效率很低。解决方法是将P_{m_2}作为当前离P_1最远的可见点保存起来。

继续分割$P_{m_2}P_2$。每次分割时，若其中分点可见，则它被当做当前离P_1最远的可见点予以保存，直至在一个给定精度下求得线段与窗口一边的交点为止。该交点即是离P_1最远的可见点。对另一半段$P_{m_1}P_1$可同样处理。对于图3-4中的线段c，离P_2最远的可见点是它与窗口左边的交点。然后画出线段P_1P_2位于两交点间的可见部分。

对于图3-4中的线段c和线段d，中点分割算法需做两遍对数查找，以分别求得离线段两个端点最远的两个可见点，即线段与窗口边的交点。每一次对分都是对交点的一次逼近。在处理线段e和g时，由于其中的一个端点可见，故仅需做一遍对数查找。用软件实现时，将顺序进行两遍对数查找；但若采用硬件实现方式，查找可并行处理。该算法可分为三步[Newm73,79]。

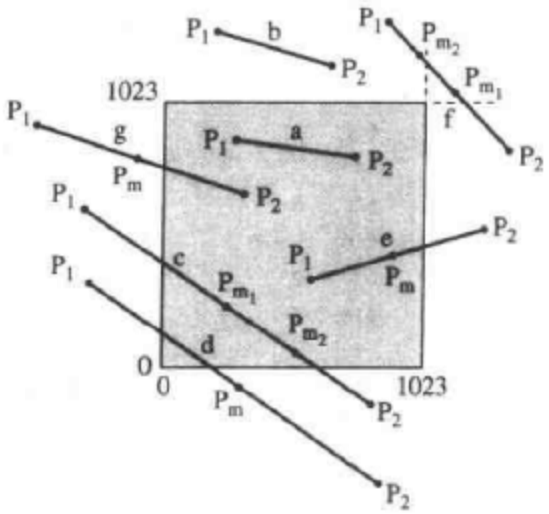


图3-4 中点分割

对线段的每一端点:

若该端点可见，则它为离另一个端点最远的可见点，处理结束。否则继续。

若线段为显然不可见段，则无输出，处理结束。否则继续。

在中点P_m处将线段P_1P_2一分为二，并以P_m作为最远可见点的初始估计，对所得的子线段P_1P_m和P_mP_2分别做上述检查。若P_mP_2为显然不可见段，则抛弃，这时中点P_m为最远可见点的过估计，继续处理P_1P_m。否则，中点P_m为最远可见点的欠估计，继续处理P_mP_2。如

果子线段非常短以至于它的中点已达到机器表示端点的精度，则计算此点的可见性，处理结束。

为了更好地说明算法的实现过程，下面举一个例子。

例3.3 中点分割。

考虑图3-4所示的窗口，其左、右、下、上边的屏幕坐标分别为0、1023、0、1023。线段c的两端点的屏幕坐标为$P_1(-307, 631)$和$P_2(820, -136)$。P_1的端点编码为(0001)，P_2的端点编码为(0100)。两端点编码均非零，因此线段c不是完全可见段；两端点编码的逻辑与为(0000)，故线段也不是显然不可见段，需求它与窗口边的交点。

采用整数运算，求得线段c的中点为

$$x_m=\frac{x_2+x_1}{2}=\frac{820-307}{2}=256.5=256$$

$$y_m=\frac{y_2+y_1}{2}=\frac{-136+631}{2}=247.5=247$$

中点的编码为(0000)。两半段P_1P_m和P_2P_m均非完全可见段或显然不可见段。首先寻找距离P_1最远的可见点，然后寻找距离P_2最远的可见点，具体分割过程如表3-2所示。线段P_1P_2与窗口边实际交于(0, 422)和(620, 0)。误差是由于运算过程中截断取整造成的。 □

表3-2 中点分割过程

P_1	P_2	P_m	注　释
-307, 631	820, -136	256, 247	保存P_1，继续处理P_mP_2
256, 247	820, -136	538, 55	继续处理P_mP_2
538, 55	820, -136	679, -41	抛弃P_mP_2，用P_m代替P_2，继续处理P_mP_2
538, 55	679, -41	608, 7	继续处理P_mP_2
608, 7	679, -41	643, -17	抛弃P_mP_2，用P_m代替P_2，继续处理P_mP_2
608, 7	643, -17	625, -5	抛弃P_mP_2，用P_m代替P_2，继续处理P_mP_2
608, 7	625, -5	616, 1	继续处理P_mP_2
616, 1	625, -5	620, -2	抛弃P_mP_2，用P_m代替P_2，继续处理P_mP_2
616, 1	620, -2	618, -1	抛弃P_mP_2，用P_m代替P_2，继续处理P_mP_2
616, 1	618, -1	617, 0	成功回到保存的P1，用P_1代替P_2，用P_m代替P_1
617, 0	-307, 631	155, 315	继续处理P_mP_2
155, 315	-307, 631	-76, 473	抛弃P_mP_2，用P_m代替P_2，继续处理P_mP_2
155, 315	-76, 473	39, 394	继续处理P_mP_2
39, 394	-76, 473	-19, 433	抛弃P_mP_2，用P_m代替P_2，继续处理P_mP_2
39, 394	-19, 433	10, 413	继续处理P_mP_2
10, 413	-19, 433	-5, 423	抛弃P_mP_2，用P_m代替P_2，继续处理P_mP_2
2, 418	-5, 423	2, 418	继续处理P_mP_2
2, 418	-5, 423	-2, 420	抛弃P_mP_2，用P_m代替P_2，继续处理P_mP_2
2, 418	-2, 420	0, 419	继续处理P_mP_2
0, 419	-2, 420	-1, 419	抛弃P_mP_2，用P_m代替P_2，继续处理P_mP_2
0, 419	-1, 419	-1, 419	成功

下面是该算法的伪代码描述，其中用到3.2节Cohen-Sutherland算法中的Endpiont，Sum和Logical子程序。

Midpoint subdivision clipping algorithm

subroutine mid($P_1, P_2, x_L, x_R, y_B, y_T$)

Window *is a* 1×4 *array containing the window edges* x_L, x_R, y_B, y_T
P_1 *and* P_2 *are the line end points with components* P_1x, P_1y, P_2x, P_2y
P1code *and* P2code *are* 1×4 *arrays containing the end point codes*

set up for visibility checks
call Endpoint(P1, Window; P1code)
call Endpoint(P2, Window; P2code)
call Sum(P1code; Sum1)
call Sum(P2code; Sum2)
call Visible(P1code, P2code, Sum1, Sum2; Vflag)
check for trivially visible line
if Vflag = yes **then**
draw visible line
exit subroutine
end if
check for trivially invisible line
if Vflag = no **then**
trivially invisible
exit subroutine
end if
the line is partially visible or crosses a corner invisibly
Error = 1
for i = 1 **to** 2
if P_1 *or* P_2 *is inside the window, only go through the loop once*
if i = 1 **and** Sum1 = 0 **then** P_1 *inside — switch* P_1, P_2
$SaveP_2 = P_1$
$Temp = P_2$
$P_2 = P_1$
$P_1 = Temp$
i = 2
end if
if i = 1 **and** Sum2 = 0 **then** P_2 *inside — switch* P_1, P_2
$Temp = P_2$
$P_2 = P_1$
$P_1 = Temp$
$SaveP_2 = P_1$
i = 2
end if
$SaveP_1 = P_1$
TempSum1 = Sum1
SaveP1code = P1code
while Abs($P_2x - P_1x$) > Error **or Abs**($P_2y - P_1y$) > Error
P_m = **Integer**(($P_2 + P_1$)/2)
$Temp = P_1$ *temporarily save* P_1

```
            P1 = Pm      assume Pm is the far end of the line
            call Endpoint(P1, Window; P1code)
            call Sum(P1code; Sum1)
            call Visible(P1code, P2code, Sum1, Sum2; Vflag)
            if Vflag = no then      reject PmP2
                P1 = Temp      backup to P1
                P2 = Pm      Pm is now the far end of the line
                P2code = P1code      switch the end point codes and sums
                Sum2 = Sum1
                call Endpoint(P1, Window; P1code)
                call Sum(P1code; Sum1)
            end if
        end while
        if i = 1 then
            SaveP2 = Pm
            P1 = Pm
            P2 = SaveP1
            Sum2 = TempSum1
                P2code = SaveP1code
            else
                P1 = Pm
                P2 = SaveP2
            end if
            call Endpoint(P1, Window; P1code)
            call Endpoint(P2, Window; P2code)
            call Sum(P1code; Sum1)
            call Sum(P2code; Sum2)
            call Visible(P1code, P2code, Sum1, Sum2; Vflag)
        next i
        the partially visible line is found — check for invisible point
        call Logical(P1code, P2code; Inter)
        if Inter <> 0 then
            draw P1P2
        end if
    return
```

3.4 凸区域的二维参数化线段裁剪

以上算法中均假设裁剪窗口为一规则的矩形窗口。然而，在许多应用中，裁剪窗口并非规则矩形。例如，若将矩形窗口在坐标系内旋转一个角度，如图3-5所示，则以前所述算法将无一适用。Cyrus和Beck提出任意凸区域内的裁剪算法（见[Cyru78]）。

在专门介绍Cyrus-Beck算法之前，先考虑对一条用参数定义的直线段的裁剪，从P_1到P_2的直线段的参数方程为:

$$P(t) = P_1 + (P_2 - P_1)t \qquad 0 \leqslant t \leqslant 1 \tag{3-1}$$

其中t为参数。当t的取值范围为$0 \leqslant t \leqslant 1$时，上式代表一条线段而不是无限长的直线。注意线段的参数表示与坐标系选择无关。由于这一性质，在线段同任意凸多边形边求交计算时，参数表示形式显得特别方便。下面先以规则矩形窗口为例说明这一方法。

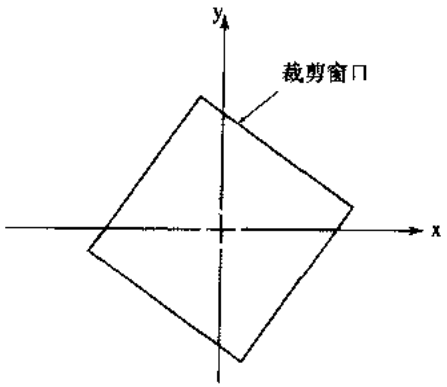


图3-5 旋转后的裁剪窗口

在二维笛卡儿坐标系中，方程(3-1)对应两个参数方程，每个方程对应一个坐标分量，即

$$x(t) = x_1 + (x_2 - x_1)t \quad 0 \leqslant t \leqslant 1 \tag{3-2a}$$

$$y(t) = y_1 + (y_2 - y_1)t \quad 0 \leqslant t \leqslant 1 \tag{3-2b}$$

对于规则矩形裁剪窗口，由于直线段与窗口任一边交点的一个坐标分量已经知道，故只需计算交点的另一坐标分量。由式(3-1)可知线段上任一点的参数t为

$$t = \frac{P(t) - P_1}{P_2 - P_1}$$

由式(3-2)可求出直线段与窗口边交点的参数值分别为

$$\text{左边：} \quad t = \frac{x_L - x_1}{x_2 - x_1} \qquad 0 \leqslant t \leqslant 1$$

$$\text{右边：} \quad t = \frac{x_R - x_1}{x_2 - x_1} \qquad 0 \leqslant t \leqslant 1$$

$$\text{顶边：} \quad t = \frac{y_T - y_1}{y_2 - y_1} \qquad 0 \leqslant t \leqslant 1$$

$$\text{底边：} \quad t = \frac{y_B - y_1}{y_2 - y_1} \qquad 0 \leqslant t \leqslant 1$$

其中x_L、x_R、y_T、y_B分别为窗口左、右、上、下各边的坐标值。如果由上面各式求得的t值超出了范围$0 \leqslant t \leqslant 1$，则表明相应交点位于直线段的两端之外，可以舍去。现举一条部分可见线段的裁剪例子来帮助理解参数化线段裁剪的基本概念。

部分可见线段

例3.4 简单的部分可见线段。

考虑由 $P_1(-3/2,-3/4)$ 到 $P_2(3/2, 1/2)$ 的部分可见线段对窗口(−1,1, −1,1)的裁剪，括号内值分别对应窗口的x_L、x_R、y_B、y_T，如图3-6所示。P_1P_2与窗口各边交点的参数值分别为

$$左边:\ t=\frac{x_L-x_1}{x_2-x_1}=\frac{-1-(-3/2)}{3/2-(-3/2)}=\frac{1/2}{3}=\frac{1}{6}$$

$$右边:\ t=\frac{x_R-x_1}{x_2-x_1}=\frac{1-(-3/2)}{3/2-(-3/2)}=\frac{5/2}{3}=\frac{5}{6}$$

$$底边:\ t=\frac{y_B-y_1}{y_2-y_1}=\frac{-1-(-3/4)}{1/2-(-3/4)}=\frac{-1/4}{5/4}=-\frac{1}{5}$$

由于此值小于零，故相应交点应舍去。

$$顶边:\ t=\frac{y_T-y_1}{y_2-y_1}=\frac{1-(-3/4)}{1/2-(-3/4)}=\frac{7/4}{5/4}=\frac{7}{5}$$

此值大于1，同样应舍去。故线段的可见部分位于 $1/6 \leqslant t \leqslant 5/6$ 之间。

交点的x、y坐标值可由参数方程求出，将$t=1/6$代入 (3-2)式得到

$$x\left(\frac{1}{6}\right)=-\frac{3}{2}+\left[\frac{3}{2}-\left(-\frac{3}{2}\right)\right]\left(\frac{1}{6}\right)=-1$$

显然这是已知的$x=-1$，表示交点位于窗口左边上。交点的y坐标为

$$y\left(\frac{1}{6}\right)=-\frac{3}{4}+\left[\frac{1}{2}-\left(-\frac{3}{2}\right)\right]\left(\frac{1}{6}\right)=-\frac{13}{24}$$

类似地，可求出$t=5/6$所对应交点的坐标值:

$$\left[x\left(\frac{5}{6}\right)\ \ y\left(\frac{5}{6}\right)\right]=\left[-\frac{3}{2}\ \ -\frac{3}{4}\right]+\left[\frac{3}{2}-\left(-\frac{3}{2}\right)\ \ \frac{1}{2}-\left(-\frac{3}{4}\right)\right]\left(\frac{5}{6}\right)$$
$$=\left[1\ \ \frac{7}{24}\right]$$

此处对x和y坐标的计算合并在一起进行。同样，由于该交点是直线段与窗口右边的交点，故参数值为$5/6$时的x坐标为已知。 □

由上例可以看出参数表示方法简单直观。不过，下例将很好地说明应用这一方法会遇到某些问题。

例3.5 部分可见线段。

考虑由 $P_3(-5/2,-1)$ 到 $P_4(3/2,2)$的直线段对窗口(−1,1,−1,1)的裁剪，如图3-6所示。直线段与窗口各边的交点的参数值分别为

$$t_L=\frac{3}{8}\quad t_R=\frac{7}{8}\quad t_B=0\quad t_T=\frac{2}{3}$$

注意这四个t值均在$0 \leqslant t \leqslant 1$之内。 □

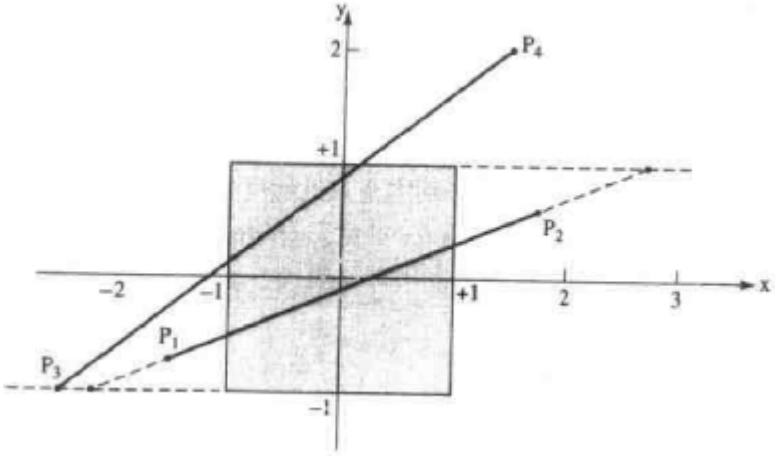


图3-6 部分可见线段参数裁剪

众所周知，当一条直线段与一个凸多边形相交最多存在两个交点，因此上例求得的四个参数值中，只有两个是所求交点的参数值。按升序对这四个参数值排序得t_B、t_L、t_T、t_R，由图3-6可知：$t_L=\frac{3}{8}$和$t_T=\frac{2}{3}$为所求交点的参数值，它们分别对应交点$(-1, \frac{1}{8})$和$(\frac{1}{6}, 1)$。不难发现，这两个t值是所有t值中最大的最小值和最小的最大值。一般来说，t值的确定是一简单的经典线性规划问题，在3.5节中将给出一个确定t值的算法。

对任何裁剪算法，迅速地判别和分离完全可见线段和完全不可见线段是十分重要的。下面的两例将说明应用上述方法可能遇到的问题。

例3.6 完全可见线段。

考虑图3-7中完全可见线段$P_1(-\frac{1}{2}, \frac{1}{2})$ $P_2(\frac{1}{2}, -\frac{1}{2})$对窗口(-1,1, -1,1)的裁剪。线段与窗口各边的交点的参数值为

$$t_L=-\frac{1}{2} \quad t_R=\frac{3}{2} \quad t_B=\frac{3}{2} \quad t_T=-\frac{1}{2}$$

所有这些值均在$0<t<1$之外。 □

由上例似乎已找到判别完全可见线段的方法。然而，例3.7却提供了一个反例。

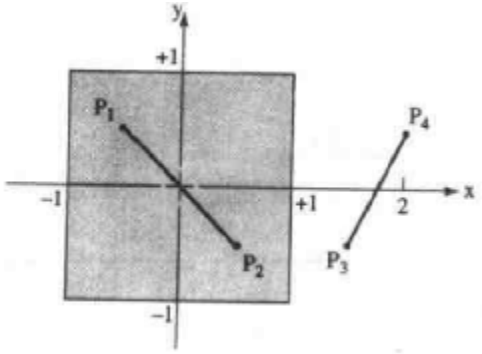


图3-7 可见与不可见线段参数裁剪

例3.7 完全不可见线段。

考虑图3-7中完全不可见线段$P_3(\frac{3}{2}, -\frac{1}{2})$ $P_4(2, \frac{1}{2})$对窗口(-1,1,-1,1)的裁剪。线段与

窗口各边的交点的参数值为

$$t_L=-5 \quad t_R=-1 \quad t_B=-\frac{1}{2} \quad t_T=\frac{3}{2}$$

同样，上述t值均位于$0<t<1$之外。　□

例3.7的结果中t值的分布与前例导出的判别完全可见线段的条件相同，然而与前例中的线段P_1P_2不同的是，此处P_3P_4是不可见线段。由此可见，对于参数表示的直线段的裁剪，并没有简单统一的方法来判别完全可见线段或完全不可见线段。要解决这一问题，需要更为确切的方法。

3.5 Cyrus-Beck算法

为了获得一个可靠的裁剪算法，必须先找到一个可靠的方法，用以判定线段上的一点是在裁剪窗口之内、之外或边界上。Cyrus-Beck算法利用法矢量的概念来解决这一问题。

考虑一个凸的裁剪区域R。尽管R并不限于二维区域，但下面的例子中将暂时假定R是二维区域。R可为任一凸的平面多边形，但不能是凹多边形。R边界上任一点a处的内向法矢量$\mathbf{n}$，满足：

$$\mathbf{n}\cdot(\mathbf{b}-\mathbf{a})\geqslant 0$$

其中b为R边界上的任一其他点。为了证实这一点，考虑矢量$\mathbf{V}_1$和$\mathbf{V}_2$的点积：

$$\mathbf{V}_1\cdot\mathbf{V}_2=|\mathbf{V}_1|\ |\mathbf{V}_2|\cos\theta$$

其中θ为$\mathbf{V}_1$、$\mathbf{V}_2$所夹的锐角。当 $\theta=\pi/2$时，$\cos\theta=0$，$\mathbf{V}_1\cdot\mathbf{V}_2=0$。即当两矢量的点积为零时，两矢量相互垂直。图3-8所示为一个凸的裁剪区域R。在边界上一点a处画出了该边界的内法线矢量$\mathbf{n}_i$和外法线矢量$\mathbf{n}_o$。图中还列出了几个从a点到边界上其他点的矢量。不难发现，$\mathbf{n}_i$与这些矢量之间的夹角总是在 $-\pi/2<\theta<\pi/2$ 之间，在此范围内$\cos\theta$恒为正，故点积也恒为正。但是外法线与上面任一矢量的夹角为$\pi-\theta$，且$\cos(\pi-\theta)=-\cos\theta$，恒取负值。为了进一步说明这一点，可考虑下面的例子。

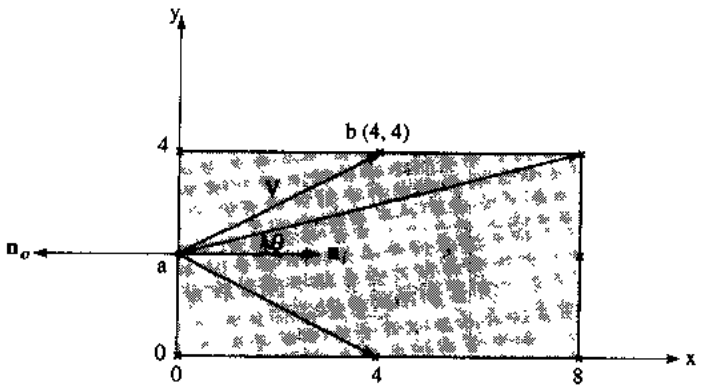


图3-8　内法线和外法线

例3.8　内法线和外法线。

考虑图3-8所示的矩形区域，其中a处的内法线和外法线分别为 $\mathbf{n}_i=\mathbf{i}$ 和 $\mathbf{n}_o=-\mathbf{i}$，$\mathbf{i}$为沿x方向的单位矢量。表3-3中列出了a点处的内、外法矢量与从a点指向矩形边界上的不同点b的矢量的点积。作为一个具体例子，注意a处的内法矢量为

$$\mathbf{n}_i = \mathbf{i}$$

从a (0, 2) 到b (4, 4) 的矢量为

$$\mathbf{b} - \mathbf{a} = 4\mathbf{i} + 2\mathbf{j}$$

点积为

$$\mathbf{n}_i \cdot (\mathbf{b}-\mathbf{a}) = \mathbf{i} \cdot (4\mathbf{i}+2\mathbf{j})=4$$

表 3-3

a	b	$\mathbf{n}_i\cdot(\mathbf{b}-\mathbf{a})$	$\mathbf{n}_o\cdot(\mathbf{b}-\mathbf{a})$
(0,2)	(0,4)	0	0
	(4,4)	4	−4
	(8,4)	8	−8
	(8,2)	8	−8
	(8,0)	8	−8
	(4,0)	4	−4
	(0,0)	0	0

表3-3中的零表明所示矢量与a点处边界的内法矢量和外法矢量垂直。 □

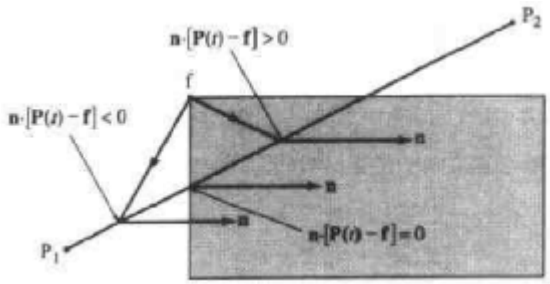


图3-9 矢量的方向

回到求取直线段与一窗口边的交点。仍考虑从P_1到P_2的直线段的参数表示

$$P(t) = P_1 + (P_2 - P_1)\,t, \qquad 0 \leqslant t \leqslant 1$$

若f是凸区域R边界上的一点，而$\mathbf{n}$是该点处边界的内法线矢量，则对于线段P_1P_2上的任一点$P(t)$:

$$\mathbf{n} \cdot [\,\mathbf{P}(t) - \mathbf{f}\,] < 0$$

表明$\mathbf{P}(t)-\mathbf{f}$指向R的外部;

$$\mathbf{n} \cdot [\,\mathbf{P}(t) - \mathbf{f}\,] = 0$$

表明$\mathbf{P}(t)-\mathbf{f}$平行于一个通过f且与法矢量垂直的平面;

$$\mathbf{n} \cdot [\,\mathbf{P}(t) - \mathbf{f}\,] > 0$$

表明$\mathbf{P}(t)-\mathbf{f}$指向R的内部，见图3-9。上述条件同时表明，若凸区域R是封闭的，即为二维情形中一个封闭的凸多边形，则一条无限长的直线与R仅有两个交点，而且这两个交点不会位于凸区域R的同一边界上。因此

$$\mathbf{n} \cdot [\,\mathbf{P}(t) - \mathbf{f}\,] = 0$$

仅有一个解。若f位于边界面或边上，$\mathbf{n}$为 f 处的内法矢量，那么线段$P(t)$上满足上式的点即是直线段与R边界的交点。

3.5.1 部分可见线段

我们用一个例子来说明部分可见线段的裁剪技术。

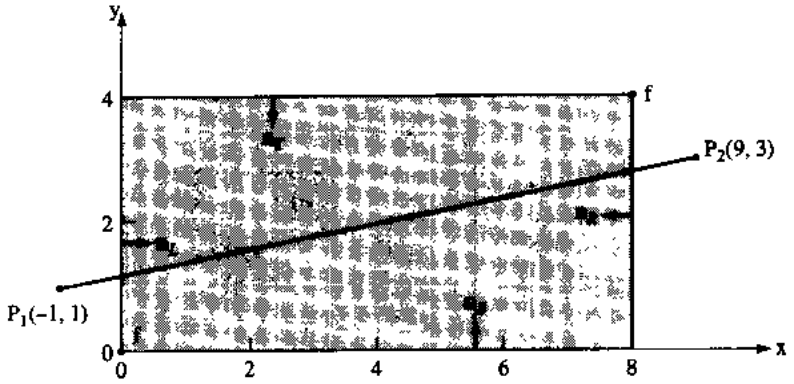


图3-10　Cyrus-Beck 裁剪——部分可见线段

例3.9　Cyrus-Beck裁剪——部分可见线段。

考虑从$P_1(-1,1)$到$P_2(9,3)$的直线段对一个矩形区域内的裁剪，如图3-10所示。直线段P_1P_2的方程为$y=(x+6)/5$，与窗口交于$(0, \frac{6}{5})$和$(8, \frac{14}{5})$。直线P_1P_2的参数表示为

$$\begin{aligned} P(t) &= P_1 + (P_2-P_1)\,t = [-1\ \ 1] + [10\ \ 2]t \\ &= (10t-1)\mathbf{i} + (2t+1)\mathbf{j} \qquad 0 \leqslant t \leqslant 1 \end{aligned}$$

其中**i**，**j**分别为沿x轴和y轴的单位矢量，矩形区域边界的内向法矢量分别为

$$\begin{aligned} &\text{左边：}\ \mathbf{n}_L = \mathbf{i} \\ &\text{右边：}\ \mathbf{n}_R = -\mathbf{i} \\ &\text{下边：}\ \mathbf{n}_B = \mathbf{j} \\ &\text{上边：}\ \mathbf{n}_T = -\mathbf{j} \end{aligned}$$

对于左边界，取$f=(0,\ 0)$，得

$$\mathbf{P}(t)-\mathbf{f} = (10t-1)\mathbf{i} + (2t+1)\mathbf{j}$$

和

$$\mathbf{n}_L \cdot [\,\mathbf{P}(t)-\mathbf{f}] = 10t-1 = 0$$

即

$$t = \frac{1}{10}$$

此即P_1P_2与裁剪窗口左边的交点：

$$P\left(\frac{1}{10}\right) = [-1\ \ 1] + [10\ \ 2]\left(\frac{1}{10}\right) = \left[0\ \ \frac{6}{5}\right]$$

这与用显式方程计算所得的结果是一致的。

对于右边界，取$f=(8,\ 4)$，得

$$\mathbf{P}(t)-\mathbf{f} = (10t-9)\ \mathbf{i} + (2t-3)\mathbf{j}$$

和

$$\mathbf{n}_R \cdot [\,\mathbf{P}(t)-\mathbf{f}\,] = -(10t-9) = 0$$

即

$$t = \frac{9}{10}$$

此即线段P_1P_2与窗口右边的交点：

$$P\left(\frac{9}{10}\right) = [-1 \quad 1] + [10 \quad 2]\left(\frac{9}{10}\right) = \left[8 \quad \frac{14}{5}\right]$$

这与显式计算的结果也是一致的。

对于下边界，仍取$f = (0, 0)$，得

$$\mathbf{n}_B \cdot [\mathbf{P}(t) - \mathbf{f}] = (2t+1) = 0$$

即

$$t = -\frac{1}{2}$$

由于它位于$0 \leqslant t \leqslant 1$之外，故抛弃。

对于上边界，同样取$f = (8, 4)$，得

$$\mathbf{n}_T \cdot [\mathbf{P}(t) - \mathbf{f}] = -(2t-3) = 0$$

即

$$t = \frac{3}{2}$$

这一t值在$0 \leqslant t \leqslant 1$之外，也被抛弃。因此线段$P_1P_2$经图3-10所示矩形区域裁剪后，其可见区段为$1/10 \leqslant t \leqslant 9/10$，即从$\left(0, 6/5\right)$到$\left(8, 14/5\right)$之间的线段。

上例说明，应用本方法可以方便地求取交点。 □

3.5.2 完全可见线段

接下来的三个例子将说明如何判别完全可见线段和完全不可见线段。

例3.10 Cyrus-Beck 裁剪——完全可见线段。

考虑从$P_1(1, 1)$到$P_2(7, 3)$的直线段对如图3-11所示矩形区域的裁剪。线段P_1P_2的参数表示为:

$$P(t) = [1 \quad 1] + [6 \quad 2]t$$

采用例3.9中的矩形区域边界点和内法线矢量，结果列于表3-4中。

表 3-4

边	n	f	$\mathbf{P}(t) - \mathbf{f}$	$\mathbf{n} \cdot [\mathbf{P}(t) - \mathbf{f}]$	t
左边界	i	(0, 0)	$(1+6t)\mathbf{i} + (1+2t)\mathbf{j}$	$1+6t$	$-1/6$
右边界	-i	(8, 4)	$(-7+6t)\mathbf{i} + (-3+2t)\mathbf{j}$	$7-6t$	$-7/6$
下边界	j	(0, 0)	$(1+6t)\mathbf{i} + (1+2t)\mathbf{j}$	$1+2t$	$-1/2$
上边界	-j	(8, 4)	$(-7+6t)\mathbf{i} + (-3+2t)\mathbf{j}$	$3-2t$	$3/2$

所有交点的t值均在$0 \leqslant t \leqslant 1$之外，故线段$P_1P_2$完全可见。

3.5.3 完全不可见线段

在下面两个例子中考察了两类不可见线段。其中一条线段完全位于窗口的左侧，可以用3.1节讨论过的端点编码方法来判定并抛弃。第二条线段跨越位于窗外的一个角域，故无法由端点编码方法确定为不可见线段。

例3.11 Cyrus-Beck裁剪——显然不可见线段。

考虑从$P_3(-6,-1)$到$P_4(-1,4)$的直线段对矩形窗口的裁剪，如图3-11所示。P_3P_4为不可见线段，它的参数表示为

$$P(t) = [-6\ \ -1] + [\,5\ \ 5\,]t$$

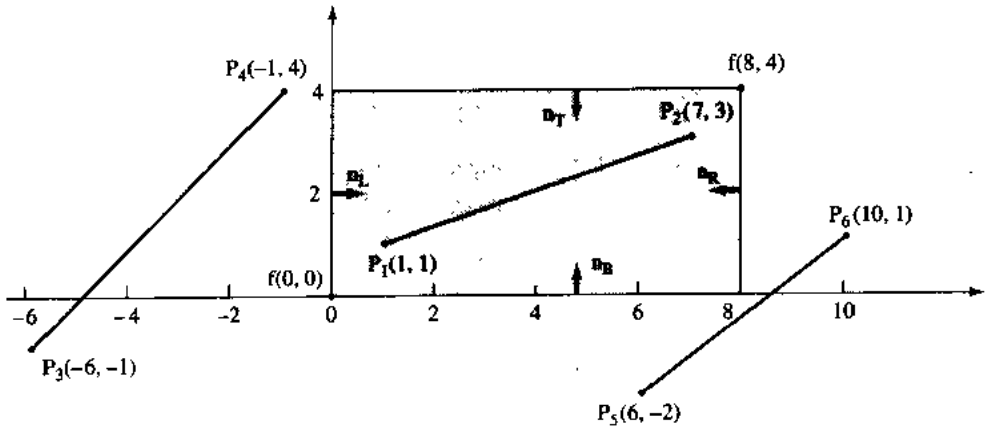


图3-11　Cyrus-Beck 裁剪——可见和不可见线段

表　3-5

边	n	f	$\mathbf{P}(t)-\mathbf{f}$	$\mathbf{n}\cdot[\mathbf{P}(t)-\mathbf{f}]$	t
左边界	i	(0,0)	$(-6+5t)\mathbf{i}+(-1+5t)\mathbf{j}$	$-6+5t$	$6/5$
右边界	−i	(8,4)	$(-14+5t)\mathbf{i}+(-5+5t)\mathbf{j}$	$-(-14+5t)$	$14/5$
下边界	j	(0,0)	$(-6+5t)\mathbf{i}+(-1+5t)\mathbf{j}$	$-1+5t$	$1/5$
上边界	−j	(8,4)	$(-14+5t)\mathbf{i}+(-5+5t)\mathbf{j}$	$-(-5+5t)$	1

采用前面两例中用过的矩形边界点和边界内法线矢量，结果列于表3-5中。可以看出，线段P_3P_4与窗口左、右两边交点的t值位于$0 \leqslant t \leqslant 1$之外，但是它与上、下两边交点的$t$值均位于$0 \leqslant t \leqslant 1$之内。由此似乎可认为线段$P_3P_4$在$\frac{1}{5} \leqslant t \leqslant 1$之间可见。然而，进一步考察$P_3P_4$与窗口左、右两边的交点时，发现两交点的参数值均大于1，这表示窗口完全位于线段P_3P_4的右侧，故线段P_3P_4不可见。　□

在上例中，若交换P_3和P_4的位置，将导出显示窗口完全位于线段左侧的结论，因此，在作出线段不可见的结论时，被裁剪线段的方向是重要的。下一例将进一步说明这一点。

例3.12　Cyrus-Beck裁剪——非显然不可见线段。

考虑从$P_5(6,-2)$到$P_6(10,1)$的直线段在矩形窗口内的裁剪，如图3-11所示。线段的参数表示为

$$P(t) = [6\ \ -2] + [4\ \ 3]t$$

采用前面两例中用过的矩形边界点和边界内法线矢量，结果列于表3-6中。可以看出，线段P_5P_6同窗口左、上两边的交点超过了t的取值范围。然而它与右、下两边的交点恰在t的取值范围内。但直线段方向是由P_5至P_6，显然它不可能先交窗口右边于$t = \frac{1}{2}$，然后再交下面边于$t = \frac{2}{3}$并贯穿窗口R。因此，线段P_5P_6为不可见线段。　□

从上面这些例子中可以看出，欲正确判别显然可见线段，还需考虑线段的方向。以上观察结果都体现在下面的Cyrus-Beck算法中。

3.5.4 Cyrus-Beck 算法的形式化描述

为了使算法形式统一，线段仍采用参数方程表示：

表 3-6

边	n	f	P(t) − f	n·[P(t) − f]	t
左边界	i	(0,0)	(6+4t)i+(−2+3t)j	6+4t	−3/2
右边界	−i	(8,4)	(−2+4t)i+(−6+3t)j	−(−2+4t)	1/2
下边界	j	(0,0)	(6+4t)i+(−2+3t)j	−2+3t	2/3
上边界	−j	(8,4)	(−2+4t)i+(−6+3t)j	−(−6+3t)	2

$$P(t) = P_1 + (P_2 - P_1)t \qquad 0 \leqslant t \leqslant 1 \tag{3-3}$$

这时连接线段上一点至边界上其他任一点的矢量与内法线矢量的点积为

$$\mathbf{n}_i \cdot [\mathbf{P}(t) - \mathbf{f}_i] \qquad i = 1, 2, 3, \cdots \tag{3-4}$$

当所取参数直线段上的点位于区域之内、区域边界上或区域之外时，点积的值分别为大于零、等于零和小于零，这一关系式适用于区域的每一边界面或边i。合并(3-3)式和(3-4)式得到直线上的点位于区域边界上的条件，即直线与窗口交点的条件：

$$\mathbf{n}_i \cdot [\mathbf{P}_1 + (\mathbf{P}_2 - \mathbf{P}_1)t - \mathbf{f}_i] = 0 \tag{3-5}$$

改写为

$$\mathbf{n}_i \cdot [\mathbf{P}_1 - \mathbf{f}_i] + \mathbf{n}_i \cdot [\mathbf{P}_2 - \mathbf{P}_1]t = 0 \tag{3-6}$$

注意矢量$\mathbf{P}_2 - \mathbf{P}_1$定义了直线段的方向，而矢量$\mathbf{P}_1 - \mathbf{f}_i$与线段端点至区域边界点之间的距离成正比。令

$$\mathbf{D} = \mathbf{P}_2 - \mathbf{P}_1$$

表示直线段的方向矢量。并令

$$\mathbf{w}_i = \mathbf{P}_1 - \mathbf{f}_i$$

表示一因子。于是式（3-6）变成：

$$t(\mathbf{n}_i \cdot \mathbf{D}) + \mathbf{w}_i \cdot \mathbf{n}_i = 0 \tag{3-7}$$

解得t为

$$t = \frac{\mathbf{w}_i \cdot \mathbf{n}_i}{\mathbf{D} \cdot \mathbf{n}_i} \qquad \mathbf{D} \neq 0 \quad i = 1, 2, 3, \cdots \tag{3-8}$$

若$\mathbf{D} \cdot \mathbf{n}_i$为零，则或者$\mathbf{D}=0$，即$P_2 = P_1$，线段化为一点；或者$\mathbf{D}$和$\mathbf{n}_i$相互垂直。如果$\mathbf{D}$和$\mathbf{n}_i$相互垂直，则直线不可能和窗口的边界相交。如果$P_2 = P_1$，那么

当$\mathbf{w}_i \cdot \mathbf{n}_i < 0$时，点位于区域或窗口之外

当$\mathbf{w}_i \cdot \mathbf{n}_i = 0$时，点位于区域或窗口边界上

当$\mathbf{w}_i \cdot \mathbf{n}_i > 0$时，点位于区域或窗口之内

式(3-8)可用来计算直线段与窗口各边交点的t值。若所得t值位于$0 \leqslant t \leqslant 1$之外，则可抛弃。虽然一直线段与凸窗口最多交于两点，对应两个t值，但是由式(3-8)可以计算出几个位于$0 \leqslant t \leqslant 1$范围之内的$t$值，每一个$t$值对应于窗口中不平行于线段的一条边。这些$t$值可分成两组，一组为下限组，分布于线段起始点一侧；另一组为上限组，分布于线段的终点一侧。现需找出下限组中的最大t值和上限组中的最小t值。若$\mathbf{D}_i \cdot \mathbf{n}_i > 0$，则所求出$t$值靠近线段的起点，属于下限组；相反，若$\mathbf{D}_i \cdot \mathbf{n}_i < 0$，则所得$t$值靠近线段的终点，属于上限组。算法采用了上述

条件来求解所构成的线性规划问题。图3-12是算法流程图。下面则是这一算法的伪代码描述。

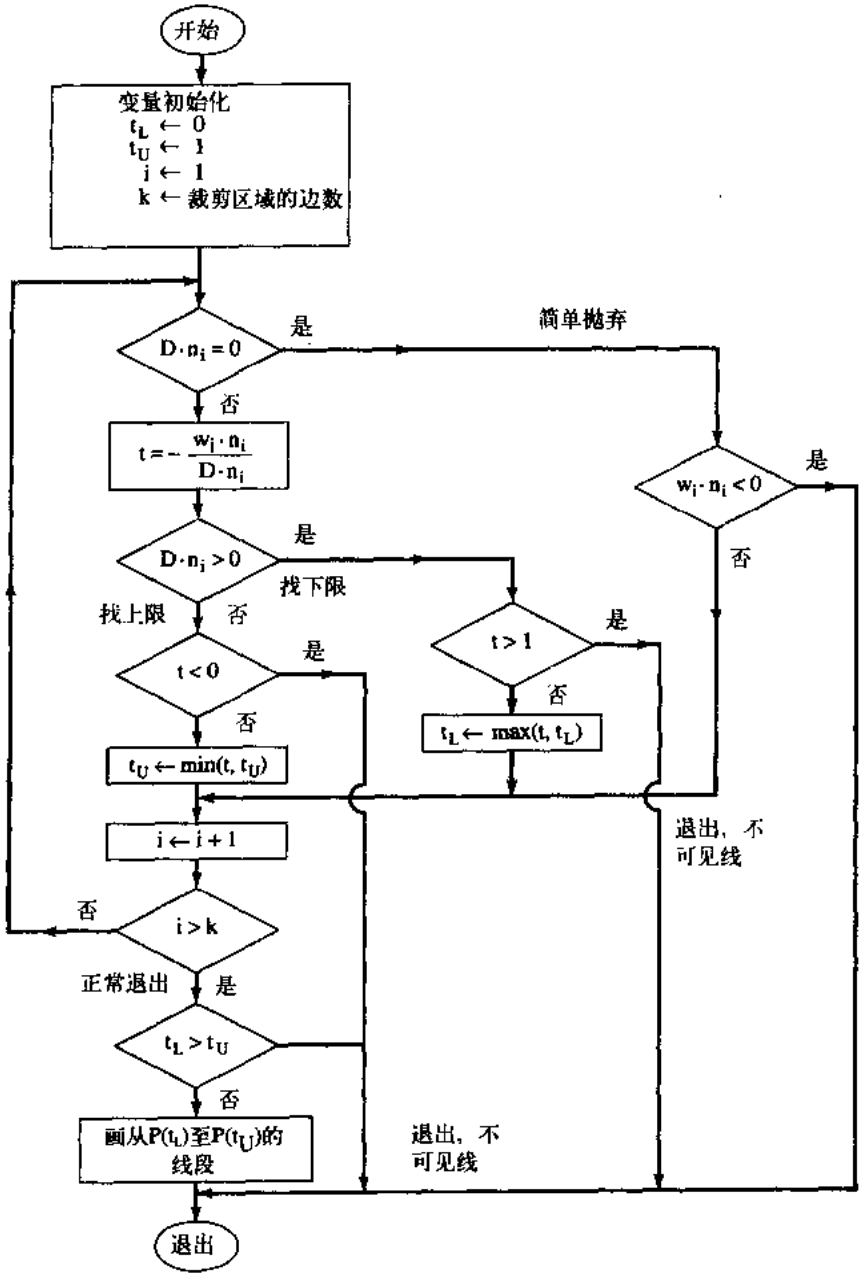


图3-12　Cyrus-Beck裁剪算法框图

Cyrus–Beck two-dimensional clipping algorithm

subroutine cb($\mathbf{P}_1, \mathbf{P}_2, \mathbf{n}_i, \mathbf{f}_i, \mathbf{k}$)

$\mathbf{P}_1$ *and* $\mathbf{P}_2$ *are the end points of the line*
the number of edges for the clipping region is $\mathbf{k}$
the $\mathbf{n}_i$ *are the* $\mathbf{k}$ *normal vectors*
the $\mathbf{f}_i$ *are the* $\mathbf{k}$ *boundary points, one in each edge*
$\mathbf{D}$ *is the direction of the line,* $\mathbf{P}_2 - \mathbf{P}_1$
$\mathbf{w}_i$ *is the weighting function,* $\mathbf{P}_1 - \mathbf{f}_i$

```
t_L, t_U are the lower and upper parameter limits
    initialize the parameter limits, assuming the entire line is visible
    t_L = 0
    t_U = 1
 calculate the directrix D
 D = P_2 − P_1
 start the main loop
 for i = 1 to k
     calculate w_i, D·n_i and w_i·n_i for this value of i
     w_i = P_1 − f_i
     call Dotproduct(D, n_i; Ddotn)
     call Dotproduct(w_i, n_i; Wdotn)
     is the line a point?
     if Ddotn <> 0 then
         the line is not a point, calculate t
         t = −Wdotn/Ddotn
         looking for the upper or the lower limit?
         if Ddotn > 0 then
             looking for the lower limit
             is t within the range 0 to 1?
             if t > 1 then              region entirely right of line
                 exit subroutine        line trivially invisible — exit
             else
                 t_L = Max(t, t_L)
             end if
         else    at this point Ddotn < 0
             looking for the upper limit
             is t within the range 0 to 1?
             if t < 0 then              region entirely left of line
                 exit subroutine        line trivially invisible — exit
             else
                 t_U = Min(t, t_U)
             end if
         end if
     else       here Ddotn = 0
         if Wdotn < 0 then
             the line is trivially invisible or an invisible point
             abnormal exit from the routine occurs
             exit subroutine
         end if
     end if
 next i
 a normal exit from the loop has occurred
```

```
    check if the line is in fact invisible
    if t_L <= t_U then      the = catches a visible corner point
        Draw line segment P(t_L) to P(t_U)
    end if
return
subroutine module to calculate the dot product
subroutine Dotproduct(Vector1,Vector2; Dproduct)
Vector1 is the first vector with components x and y
Vector2 is the second vector with components x and y
Dproduct is the dot or inner product
    Dproduct = Vector1x*Vector2x + Vector1y*Vector2y
return
```

3.5.5 非规则窗口

为了说明上述算法并不限于矩形裁剪窗口，考虑下面的例子。

例3.13 Cyrus-Beck裁剪——不规则窗口。

在图3-13中，裁剪窗口为一个八边形，被裁剪线段为$P_1(-1, 1)P_2(3, 3)$。表3-7列出了Cyrus-Beck算法的完整的结果。作为一个具体例子，考虑由V_5到V_6的多边形边。算法先计算

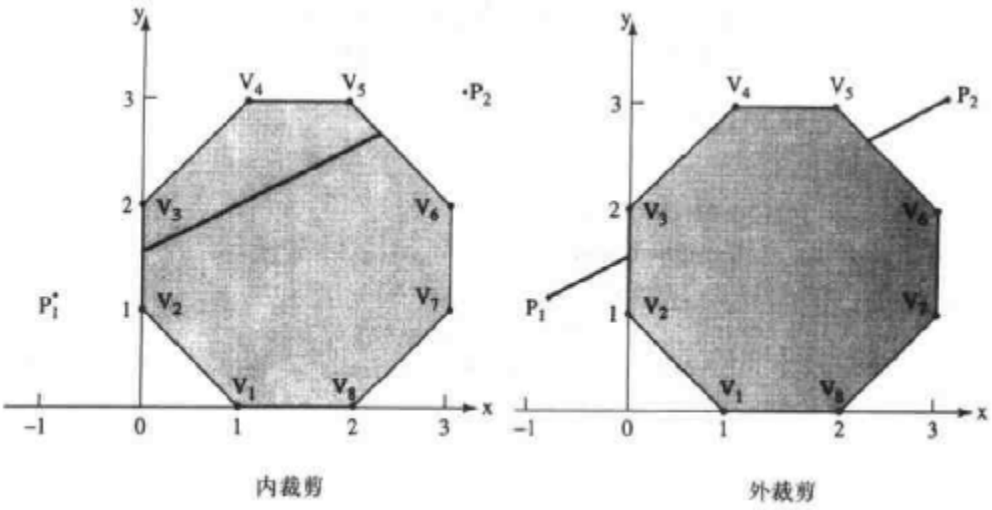


图3-13　Cyrus-Beck算法对多边形的内裁剪和外裁剪

$$\mathbf{D} = \mathbf{P}_2 - \mathbf{P}_1 = [3\ \ 3] - [-1\ \ 1] = [4\ \ 2]$$

取窗口边界点$f(2, 3)$，有

$$\mathbf{w} = \mathbf{P}_1 - \mathbf{f} = [-1\ \ 1] - [2\ \ 3] = [-3\ \ -2]$$

窗口边V_5V_6的内法矢量为

$$\mathbf{n} = [-1\ \ -1]$$

所以

$$\mathbf{D} \cdot \mathbf{n} = -6 < 0$$

故所求t值属上限值。由于

$$\mathbf{w} \cdot \mathbf{n} = 5$$

$$t_U = -\frac{5}{-6} = \frac{5}{6}$$

由表3-7可知，最大下限值为$t_L = \frac{1}{4}$，而最小上限值为$t_U = \frac{5}{6}$，故直线段P_1P_2在$\frac{1}{4} < t < \frac{5}{6}$内，即从$(0, \frac{3}{2})$到$(\frac{7}{3}, \frac{8}{3})$的部分可见，如图3-13所示。

表 3-7

边	n	f	w	w·n	D·n①	t_L	t_U
V_1V_2	[1 1]	(1,0)	[2 1]	−1	6	1/6	
V_2V_3	[1 0]	(0,2)	[−1 −1]	−1	4	1/4	
V_3V_4	[1 −1]	(0,2)	[−1 −1]	0	2	0	
V_4V_5	[0 −1]	(2,3)	[−3 −2]	2	−2		1
V_5V_6	[−1 −1]	(2,3)	[−3 −2]	5	−6		5/6
V_6V_7	[−1 0]	(3,1)	[−4 0]	4	−4		1
V_7V_8	[−1 1]	(3,1)	[−4 0]	4	−2		2
V_8V_1	[0 1]	(1,0)	[−2 1]	1	2	−1/2	

① $\mathbf{D} \cdot \mathbf{n} < 0$，为上限$t$值($t_U$)；$\mathbf{D} \cdot \mathbf{n} > 0$，为下限$t$值（$t_L$)。

3.6 Liang-Barsky 二维裁剪

本质上说，Liang-Barsky二维裁剪算法(见[Lian84])是一种特殊形式的Cyrus-Beck算法。(见3.5节)。虽然Liang-Barsky算法适用于任意的凸二维和三维裁剪区域，但通常只讨论与坐标轴平行的矩形裁剪区域(规则、方正的裁剪区域)。如图3-14所示，x_L、x_R、y_B、y_T分别表示裁剪区域或窗口的左、右、下、上四条边。

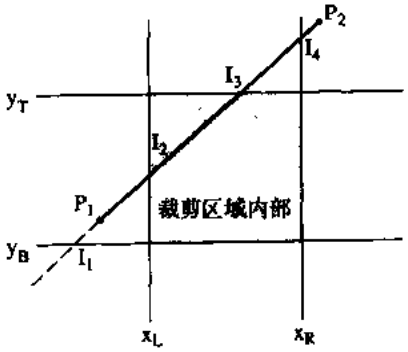


图3-14 Liang-Barsky 裁剪：部分可见线段

裁剪区域的内部可以表达为两个不等式：

$$x_L < x < x_R$$

$$y_B \le y \le y_T$$

将直线的参数方程$P(t) = P_1 + (P_2 - P_1)t$，$0 \le t \le 1$的两个分量，即

$$x(t) = x_1 + (x_2 - x_1)t = x_1 + t\Delta x \qquad 0 < t < 1$$

$$y(t) = y_1 + (y_2 - y_1)t = y_1 + t\Delta y \qquad 0 < t < 1$$

代入上面的不等式，得到裁剪区域内部的另一种表示：

$$-t\Delta x < x_1 - x_L \qquad -t\Delta y < y_1 - y_B$$

$$t\Delta x < x_R - x_1 \qquad t\Delta y < y_T - y_1$$

即

$$td_i < q_i \qquad i = 1, 2, 3, 4 \tag{3-9}$$

其中

$$d_1 = -\Delta x \qquad d_2 = \Delta x \qquad d_3 = -\Delta y \qquad d_4 = \Delta y$$

$$q_1 = x_1 - x_L \qquad q_2 = x_R - x_1 \qquad q_3 = y_1 - y_B \qquad q_4 = y_T - y_1$$

$i = 1, 2, 3, 4$分别对应于窗口的左、右、下、上四条边。

窗口的每一条边将裁剪平面划分为两个区域：一个是裁剪区域的内侧，另一个是外侧。例如，在图3-14中位于无限长直线x_L左侧的区域是裁剪区域的外侧，位于x_L右侧的区域是裁剪区域的内侧。不等式组(3-9) 式可以用来确定点或线段与裁剪区域的几何关系。

如果$q_i < 0$，那么P_1位于裁剪区域的第i条边的外侧(不可见侧)。要说明这一点，只需看到：

$q_1 = x_1 - x_L < 0 \quad \Rightarrow \quad x_L > x_1 \quad \Rightarrow$ P_1位于左边的左侧

$q_2 = x_R - x_1 < 0 \quad \Rightarrow \quad x_1 > x_R \quad \Rightarrow$ P_1位于右边的右侧

$q_3 = y_1 - y_B < 0 \quad \Rightarrow \quad y_B > y_1 \quad \Rightarrow$ P_1位于底边的下面

$q_4 = y_T - y_1 < 0 \quad \Rightarrow \quad y_1 > y_T \quad \Rightarrow$ P_1位于顶边的上面

其实这正是3.1节中确定Cohen-Sutherland端点编码的另一种方法。

类似地，如果$q_i \ge 0$，那么P_1位于裁剪区域的第i条边的内侧(可见侧)。在特殊情况下，即q_i=0时，P_1位于裁剪区域的边界上。如果d_i=0，那么直线平行于相应的边。因此，如果d_i = 0且$q_i < 0$，那么整个线段位于裁剪区域的外面，可以被简单抛弃。

再考虑式(3-9)，可以看到，参数直线$P(t)$与裁剪区域边界的交点为

$$t = \frac{q_i}{d_i}$$

假设$P(t)$不平行于窗口的边，那么$P(t)(-\infty < t < \infty)$与窗口的每一条边都有一个交点，交点记为$I_i$，如图3-14所示。然而对一条直线段$P(t)(0 \le t \le 1)$来说，任何参数值在$0 \le t \le 1$之外的交点都要舍去。

正如图3-14所示，即使舍去$0 \le t \le 1$之外的那些t值，得到的t值仍然可能多于两个。回忆一下在3.5节中对Cyrus-Beck算法的讨论，将交点的参数值分成上限组和下限组。寻找下限组的最大值和上限组的最小值，就可以确定正确的交点。Liang-Barsky二维裁剪算法的流程

图见于图3-15中。下面是Liang-Barsky算法的伪代码描述[Lian84]。

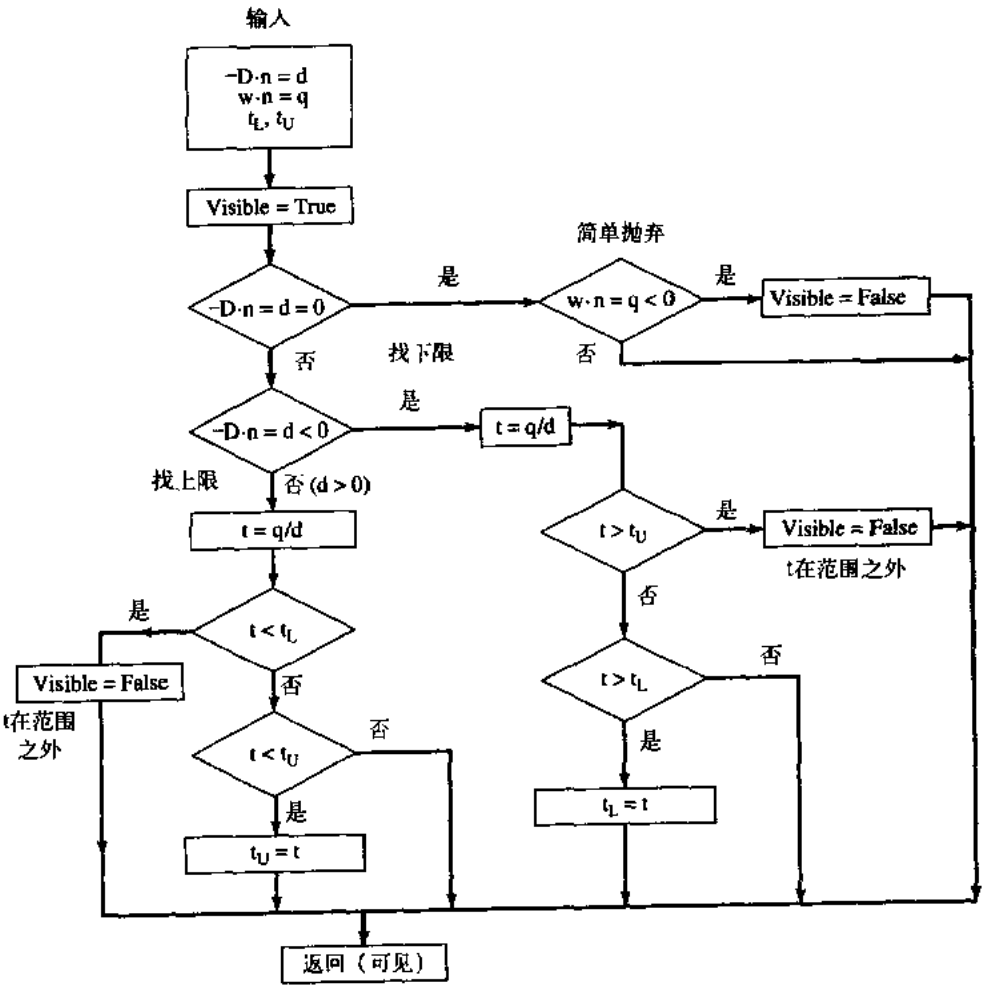


图3-15 Liang-Barsky 裁剪算法框图：图中给出了Cyrus-Beck 参数，$-\mathbf{D}\cdot\mathbf{n}=d$ 和 $\mathbf{w}\cdot\mathbf{n}=q$，以便与图3-12比较

Liang–Barsky two-dimensional clipping algorithm

P_1 *and* P_2 *are the line end points with components* x_1, y_1, x_2, y_2
t_L *and* t_U *are the lower and upper parametric limits*
x_L, x_R, y_B, y_T *are the left, right, bottom and top window edges*

```
function clipt(d, q; t_L, t_U)
    clipt performs trivial rejection tests and finds
    the maximum of the lower set of parameter values and
      the minimum of the upper set of parameter values
    visible = true
      if d = 0 and q < 0 then        line is outside and parallel to edge
          visible = false
      else if d < 0 then             looking for upper limit
          t = q/d
          if t > t_U then            check for trivially invisible
```

```
            visible = false
        else if t > t_L then        find the minimum of the maximum
            t_L = t
        end if
    else if d > 0 then        looking for the lower limit
        t = q/d
        if t < t_L then        check for trivially invisible
            visible = false
        else if t < t_U then        find the maximum of the minimum
            t_U = t
        end if
    end if
    return(visible)
    end function
    start the main algorithm
    t_L = 0
    t_U = 1
    deltax = x_2 − x_1
    if clipt(−deltax, x_1 − x_L, t_L, t_U) = true then        left edge
        if clipt(deltax, x_R − x_1, t_L, t_U) = true then        right edge
            deltay = y_2 − y_1
            if clipt(−deltay, y_1 − y_B, t_L, t_U) = true then        bottom edge
                if clipt(deltay, y_T − y_1, t_L, t_U) = true then        top edge
                    if t_U < 1 then
                        x_2 = x_1 + t_U * deltax
                        y_2 = y_1 + t_U * deltay
                    end if
                    if t_L > 0 then
                        x_1 = x_1 + t_L * deltax
                        y_1 = y_1 + t_L * deltay
                    end if
                    Draw P_1P_2
                end if
            end if
        end if
    end if
finish
```

与Cyrus-Beck算法的比较

从3.5节Cyrus-Beck算法的$D=P_2-P_1$、$w_i=p_1-f_i$、表3-8以及两个流程图3-12和图3-15，可以看到二维的Liang-Barsky算法是一种特殊情形下的Cyrus-Beck算法。下面用一些例子进一步讲解Liang-Barsky算法及其与Cyrus-Beck算法的比较。

例3.14　Liang-Barsky裁剪——部分可见线段。

用Liang-Barsky裁剪算法在如图3-10所示的矩形窗口中裁剪线段$P_1(-1,1)P_2(9,3)$，并与例

3.9中Cyrus-Beck算法的结果作一比较。

直线的参数表示为

$$P(t) = P_1 + (P_2 - P_1)t = [-1 \quad 1] + [10 \quad 2]t$$

$$x(t) = x_1 + (x_2 - x_1)t = -1 + 10t \qquad 0 < t < 1$$

$$y(t) = y_1 + (y_2 - y_1)t = -1 + 2t$$

$$d_1 = -\Delta x = -(x_2 - x_1) = -10$$

$$d_2 = \Delta x = x_2 - x_1 = 10$$

$$d_3 = -\Delta y = -(y_2 - y_1) = -2$$

$$d_4 = \Delta y = y_2 - y_1 = 2$$

在图3-10中，$x_L = 0$，$x_R = 8$，$y_B = 0$，$y_T = 4$。所以

$$q_1 = x_1 - x_L = -1 - 0 = -1$$

$$q_2 = x_R - x_1 = 8 - (-1) = 9$$

$$q_3 = y_1 - y_B = 1 - 0 = 1$$

$$q_4 = y_T - y_1 = 4 - 1 = 3$$

应用Liang-Barsky算法，得

$t_L = 0$
$t_U = 1$
deltax = 10
clipt(d = −10, q = −1, $t_L = 0$, $t_U = 1$)
q < 0
    t = 1/10
    t = 1/10 > $t_L = 0$
        t_L = t = 1/10
return (true)
clipt(d = 10, q = 9, t_L = 1/10, $t_U = 1$)
    d > 0
        t = 9/10
        t = 9/10 < $t_U = 1$
            t_L = t = 9/10
return (true)
deltay = 2
clipt(d = −2, q = 1, t_L = 1/10, t_U = 9/10)
    d < 0
        t = −1/2
        t = −1/2 < t_L = 1/10
    return (true)
    clipt(d = 2, q = 3, t_L = 1/10, t_U = 9/10)
        d > 0
            t = 3/2
            t = 3/2 > t_U = 9/10
        return (true)
        t_U = 9/10 < 1
            $x_2 = -1 + 9/10\ (10) = 8$
            $y_2 = 1 + 9/10\ (2) = 14/5$

$t_L = 1/10 > 0$
$x_1 = -1 + 1/10\ (10) = 0$
$y_1 = 1 + 1/10\ (2) = 6/5$
Draw P_1P_2

P_1P_2的可见部分为$1/10 < t < 9/10$，即 $P_1\left(0, \frac{6}{5}\right)P_2\left(8, \frac{14}{5}\right)$，与例3.9中Cyrus-Beck裁剪算法的结果相同。 □

表3-8　Liang-Barsky和Cyrus-Beck算法之间的对应关系

边	$\mathbf{n}_i$	$\mathbf{f}_i$	$\mathbf{w}_i$	$-d_i$ $\mathbf{D}_i\cdot\mathbf{n}_i$	q_i $\mathbf{w}_i\cdot\mathbf{n}_i$	$t=-\frac{q_i}{d_i}$ $-\frac{\mathbf{w}_i\cdot\mathbf{n}_i}{\mathbf{D}_i\cdot\mathbf{n}_i}$
x_L	[1　0]	$[x_L\ \ y_T]$	$[x_1-x_L\ \ y_1-y_T]$	x_2-x_1	x_1-x_L	$-\frac{x_1-x_L}{x_2-x_1}$
x_R	[−1　0]	$[x_R\ \ y_B]$	$[x_1-x_R\ \ y_1-y_B]$	$-(x_2-x_1)$	$-(x_1-x_R)$	$-\frac{x_1-x_R}{x_2-x_1}$
y_B	[0　1]	$[x_R\ \ y_B]$	$[x_1-x_R\ \ y_1-y_B]$	y_2-y_1	y_1-y_B	$-\frac{y_1-y_B}{y_2-y_1}$
y_T	[0　−1]	$[x_L\ \ y_T]$	$[x_1-x_L\ \ y_1-y_T]$	$-(y_2-y_1)$	$-(y_1-y_T)$	$-\frac{y_1-y_T}{y_2-y_1}$

在下面例子中，用一条完全不可见的线段比较Cyrus-Beck算法和Liang-Barskey算法

例3.15　Liang-Barsky二维裁剪 ——不可见线段。
用Liang-Barsky算法在矩形窗口内裁剪直线段$P_3(-6, -1)P_4(-1, 4)$，如图3-11所示。

$$
\begin{aligned}
P(t) &= P_3 + (P_4 - P_3)t &&= [-6 \quad -1] + [5 \quad 5]\,t \\
x(t) &= x_3 + (x_4 - x_3)t &&= -6 + 5t \qquad\qquad 0 < t < 1 \\
y(t) &= y_3 + (y_4 - y_3)t &&= -1 + 5t
\end{aligned}
$$

Liang-Barsky算法

Liang-Barsky 算法中的一些值:

$d_1 = -\Delta x = -(x_4 - x_3) = -5$
$d_2 = \Delta x = x_4 - x_3 = 5$
$d_3 = -\Delta y = -(y_4 - y_3) = -5$
$d_4 = \Delta y = y_4 - y_3 = 5$

$q_1 = x_3 - x_L = -6 - 0 = -6$
$q_2 = x_R - x_3 = 8 - (-6) = 14$
$q_3 = y_3 - y_B = -1 - 0 = -1$
$q_4 = y_T - y_3 = 4 - (-1) = 3$

应用Liang-Barsky算法，得

$t_L = 0$
$t_U = 1$
deltax = 5

Cyrus-Beck 算法

Cyrus-beck 算法中的内法矢量$\mathbf{n}_i$、边界点$\mathbf{f}_i$以及矢量$\mathbf{P}(t)-\mathbf{f}$均见表3-5。
参照3.5节中的Cyrus-Beck 算法，得到

$t_L = 0$
$t_U = 1$
$\mathbf{D} = [5 \quad 5]$
$i = 1$
　$\mathbf{n}_1 = [1 \quad 0]$
　$\mathbf{w}_1 = P_3 - f_1 = [-6 \quad -1]$
　Ddotn = 5
　Wdotn = −6
　Ddotn <> 0
　　$t = -(-6/5) = 6/5$
　　Ddotn > 0 (下限)

clipt($d=-5, q=-6, t_L=0, t_U=1$)
visible = true
$d<0$
$t = {}^{-6}/_{-5} = {}^{6}/_{5}$
$t > t_U$
visible = false
return (false)
线段显然不可见

$t>1$
退出子程序
线段显然不可见

因此，Liang-Barsky算法与Cyrus-Beck算法在抛弃完全不可见线段时效率相同。 □

在下一个例子中，用一条跨越裁剪窗口的角域的不可见线段来比较Liang-Barsky算法和Cyrus-Beck算法。

例3.16 Liang-Barskey二维裁剪——跨角不可见线段。

延长例3.15中的线段P_3P_4为$P_5(-6,-1)P_6(2,7)$。如图3-11所示，这条不可见线段跨越裁剪区域左上角。比较Liang-Barskey算法和Cyrus-Beck算法对该线段的裁剪。

$$P(t) = P_5 + (P_6 - P_5)t = [-6 \quad -1] + [8 \quad 8]t \qquad 0 < t < 1$$
$$x(t) = x_5 + (x_6 - x_5)t = -6 + 8t \qquad 0 < t < 1$$
$$y(t) = y_5 + (y_6 - y_5)t = -1 + 8t \qquad 0 < t < 1$$

Liang-Barsky算法

Liang-Barsky算法中的一些值:

$d_1 = -\Delta x = -(x_6 - x_5) = -8$
$d_2 = \Delta x = x_6 - x_5 = 8$
$d_3 = -\Delta y = -(y_6 - y_5) = -8$
$d_4 = \Delta y = y_6 - y_5 = 8$

q_1、q_2、q_3、q_4和例3.15中的值相同。

应用Liang-Barsky算法，得

$t_L = 0$
$t_U = 1$
deltax = 8
clipt($d=-8, q=-6, t_L=0, t_U=1$)
visible = true
$d<0$
$t = {}^{-6}/_{-8} = {}^{3}/_{4}$
$t > t_L$
$t_L = {}^{3}/_{4}$
return (true)
clipt($d=8, q=14, t_L={}^{3}/_{4}, t_U=1$)
visible = true
$d>0$
$t = {}^{14}/_{8} = {}^{7}/_{4}$
$t > t_L$
$t > t_U$
return (true)

Cyrus-Beck 算法

Cyrus-Beck 算法中的内法向量n_i，边界点f_i以及矢量$P(t)-f$均见表3-5。

参照3.5节中的Cyrus-beck 算法，得到

$t_L = 0$
$t_U = 1$
$\mathbf{D} = [8 \quad 8]$
$i = 1$
$\mathbf{n}_1 = [1 \quad 0]$
$\mathbf{w}_1 = P_5 - f_1 = [-6 \quad -1]$
Ddotn = 8
Wdotn = −6
Ddotn <> 0
$t = -({}^{-6}/_{8}) = {}^{3}/_{4}$
Ddotn > 0 下限
$t < 1$
$t_L = \text{Max}({}^{3}/_{4}, 0) = {}^{3}/_{4}$
$i = 2$
$\mathbf{n}_2 = [-1 \quad 0]$
$\mathbf{w}_2 = P_5 - f_2 = [-14 \quad -5]$
Ddotn = −8
Wdotn = 14
Ddotn <> 0
$t = -({}^{-14}/_{-8}) = {}^{7}/_{4}$
Ddotn < 0 上限
$t > 0$
$t_U = \text{Min}({}^{7}/_{4}, 1) = {}^{7}/_{4}$
$i = 3$
$\mathbf{n}_3 = [0 \quad 1]$

clipt($d=-8, q=-1, t_L=3/4, t_U=1$)
　visible = true
　$d<0$
　$t=-1/-8=1/8$
　$t<t_U$
　$t<t_L$
　return (true)
clipt($d=8, q=5, t_L=3/4, t_U=1$)
　visible = true
　$d>0$
　$t=5/8$
　$t<t_L$
　visible = false
　return (false)
　线段不可见

　$\mathbf{w}_3=P_5-f_3=[-6 \quad -1]$
　Ddotn = 8
　Wdotn = −1
　Ddotn <> 0
　$t=-(-1/8)=1/8$
　Ddotn > 0　　下限
　$t<1$
　$t_L=\text{Max}(1/8, 3/4)=3/4$
$i=4$
　$\mathbf{n}_4=[0 \quad -1]$
　$\mathbf{w}_4=P_5-f_4=[-14 \quad -5]$
　Ddotn = −8
　Wdotn = 5
　Ddotn <> 0
　$t=-(5/-8)=5/8$
　Ddotn < 0　　上限
　$t>0$
　$t_U=\text{Min}(5/8, 1)=5/8$
$t_L>t_U$
线段不可见

两个算法都是在计算了4个t值之后抛弃这条完全不可见线段，因此它们效率的相同。 □

这些例子与在图3-12和图3-15中的流程图说明，Liang-Barsky算法与Cyrus-Beck算法在本质上是一样的。

3.7 Nicholl-Lee-Nicholl二维裁剪

Nicholl-Lee-Nicholl[Nich87]利用迴避原则得到了一个高效率的二维裁剪算法。不过该算法没有拓展到三维和多边形裁剪区域，而是仔细地检查直线和裁剪区域的相对位置来简单地抛弃显然不可见线段，从而避免了不必要的复杂求交运算。运用迴避原则设计算法或改进算法的效率无疑会增加算法的复杂度。Nicholl-Lee-Nicholl算法也不例外。

如图3-16所示，Nicholl-Lee-Nicholl算法将二维平面划分为三种类型的区域：内部、边域和角域。对一条线段来说，它的起点P_1所在区域必属上述三种之一，而它的另一端点P_2可以位于图3-16所示的九个区域中的任意一个区域内。

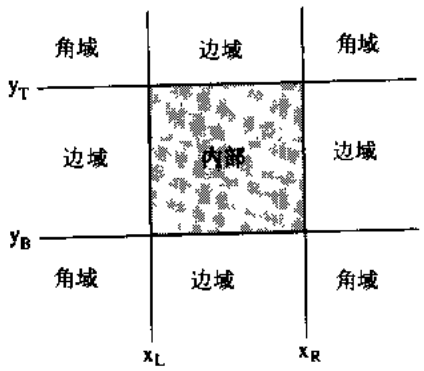


图3-16　Nicholl-Lee-Nicholl算法的区域划分

无论P_1位于裁剪区域的内部、边域或角域，如图3-17所示，连接P_1和裁剪区域的四个角点的直线都将裁剪区域划分为四个子区域。按直线通过的角点在裁剪区域中的方位将这四条

直线分别记为LT、RT、LB和RB，表示它们所经过的角点分别处于裁剪区域的左上、右上、左下和右下方位。通过比较直线P_1P_2和这四条直线的斜率，可以确定直线的可见性、直线和裁剪区域边界的交点的个数以及交点所在的边界。

例如，在图3-17c中，P_1位于左下角域内。假设P_2位于裁剪区域外的任一区域中，记m为直线P_1P_2的斜率，m_{LT}、m_{RT}、m_{RB}和m_{LB}分别为经过P_1及裁剪区域四个角点的直线的斜率，那么

if $m > m_{LT}$ **then** visible = false　　显然不可见
if $m < m_{RB}$ **then** visible = false　　显然不可见
if $m_{RB} \leqslant m \leqslant m_{LT}$ **then** visible = true　　两个交点
　　if $m = m_{LT}$ **then**　　特殊情况：一个交点
　　if $m_{LB} < m < m_{LT}$ **then**　　与左边、顶边相交
　　if $m_{RT} < m \leqslant m_{LB}$ **then**　　与底边、顶边相交
　　if $m_{RB} < m < m_{RT}$ **then**　　与底边、右边相交
　　if $m = m_{RB}$ **then**　　特殊情况：一个交点

计算斜率需要两次减法和一次除法。但是通过首先检查P_2是否在左边的左侧或底边的下面，可以避免这些复杂计算。通过检查可消去其P_2端点位于底边和RB线之间以及P_2位于左侧和LT线之间的那些线段。

if $x_2 < x_L$ **then** visible = false　　显然不可见
if $y_2 < y_B$ **then** visible = false　　显然不可见

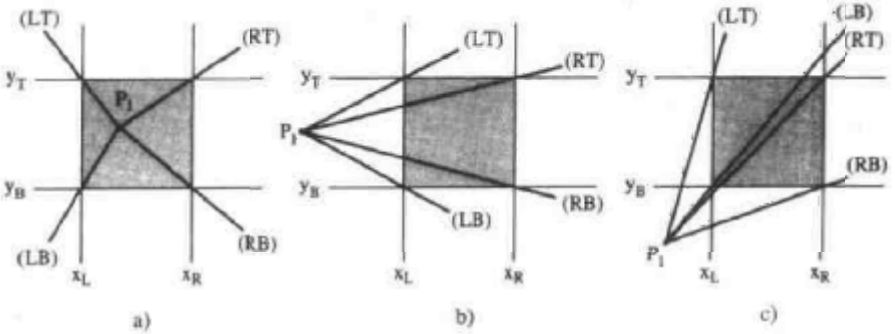


图3-17　平面的细分

a) P1在裁剪区域内部　b) P1在一边域　c) P1在一角区域

假定P_1位于裁剪区域的左下角域，当P_2位于裁剪区域之外时，整个算法可描述如下。

Nicholl–Lee–Nicholl two-dimensional clipping algorithm

subroutine nln($P_1, P_2, x_L, x_R, y_B, y_T$)

P_1 *and* P_2 *are the line end points with components* x_1, y_1, x_2, y_2
P_1 *is assumed to be in the lower left corner, and not coincident with the lower left corner, of the clipping region*
x_L, x_R, y_B, y_T *are the left, right, bottom and top window edges*
m, m_{LT}, m_{LB}, m_{RT}, m_{RB} *are the slopes of the line, the line through the left-top, left-bottom, right-top and right-bottom clipping region corners.*

　　if $x_1 < x_L$ **then**　　P_1 *in lower left corner*

```
if y1 < yB then
    if x2 < xL then
        visible = false        trivially invisible
        exit subroutine
    else if y2 < yB then
        visible = false        trivially invisible
        exit subroutine
    else        may be partially visible, 1 or 2 intersections
        deltax = x2 − x1
        deltay = y2 − y1
        deltaxLT = xL − x1
        deltayLT = yT − y1
        m = deltay/deltax
        mLT = deltayLT/deltaxLT
        if m > mLT then        passes left of LT corner
            visible = false
            exit subroutine
        else
            if m = mLT then        LT corner intersection
                x1 = x2 = xL
                y1 = y2 = yT
            else
                deltaxLB = deltaxLT        not really needed
                deltayLB = yB − y1
                mLB = deltayLB/deltaxLB
                if mLB < m and m < mLT then        left & top
                    x2 = x1 + deltayLT/m
                    y2 = yT
                    x1 = xL
                    y1 = y1 + m * deltaxLT
                else
                    deltaxRT = xR − x1
                    deltayRT = deltayLT        not really needed
                    mRT = deltayRT/deltaxRT
                    if mRT < m and m ≤ mLB then  bottom & top
                        x2 = x1 + deltayRT/m
                        y2 = yT
                        x1 = x1 + deltayLB/m
                        y1 = yB
                    else
                        deltaxRB = deltaxRT        not really needed
                        deltayRB = deltayLB        not really needed
                        mRB = deltayRB/deltaxRB
                        if mRB < m and m < mRT then    bottom
                            y2 = y1 + m * deltaxRB          & right
                            x2 = xR
                            x1 = x1 + deltayRB/m
                            y1 = yB
```

```
                    else if m = mRB then  RB corner inter.
                        x1 = x2 = xR
                        y1 = y2 = yB
                    else if m < mRB then   passes below RB
                        visible = false
                        exit  subroutine
                    end if       end bottom & right
                end if       end bottom & top
            end if       end left & top
        end if       end LT corner intersection
    end if       end passes left of LT corner
end if       end x2 < xL
end if       end y1 < yB
end if       P1 in lower left corner.
return
```

对以下的每一种情形，均必须分别设计类似的子程序P_1位于其他每一个角域或边域，P_2位于裁剪区域的外部或内部；或P_1位于裁剪区域的内部，P_2位于裁剪区域的外部或内部。对于最后一种情形P_1P_2完全可见。这些子程序还要有层次地组织起来才能构成一个有效的算法。显然，这里面有很多的情况需要考虑，最终算法相当复杂，在设计、测试和检验该算法时需小心从事。

Nicholl、Lee和Nicholl指出，利用裁剪问题的对称性，可以减少设计、测试和验证该算法的工作量。根据对称性，处理一个角域或边域的情形后，其他情形的处理可通过围绕原点旋转90°、180°或270°，或者对一个坐标轴或直线$y=-x$作反射变换而得到。并且，还可以将处理基本情形的子程序代码由机器自动转换为其他情形的程序代码（例如[Arsa79]）。

Duvanenko等人（见[Duva93]）假定在一个数值计算环境中加法比减法快、乘法除法比加法和减法慢而乘法比除法快，通过对各类算法所涉及的整型数和浮点数的机器操作的分析，发现Nicholl-Lee-Nicholl(NLN)算法比Newman和Sproul书中介绍的Cohen-Sutherland算法（见[Newm79]）快。但判定显然不可见线段的速度并不高于改进的Cohen-Sutherland算法（见[Dnva93]）。此外，他们还发现NLN算法和改进的Cohen-Sutherland算法都比Liang-Barsky算法要快。Duvanenko等人还发现，Sutherland-Hodgman多边形裁剪算法(参见3-19节)用于二维线段裁剪时，其裁剪速度和上述算法相当，甚至比它们还要快，特别是在并行执行的时候。

3.8 内裁剪和外裁剪

在3.7节中着重讨论了直线段在一个区域或多边形内部的裁剪。但是直线段也可以在一区域或多边形外部进行裁剪，即决定线段的哪些部分位于区域之外，并画出这些位于区域外面的部分。例如，在图3-13中，线段P_1P_2位于窗口之外的可见部分为：$0 \leqslant t < \frac{1}{4}$ 和 $\frac{5}{6} < t \leqslant 1$，即从(-1,1)到$\left(0,\frac{3}{2}\right)$以及从$\left(\frac{7}{3},\frac{8}{8}\right)$到(3,3)。直线段对窗口的内裁剪和外裁剪的结果均示于图3-13中。

如图3-18所示，外裁剪在处理多窗口显示时非常重要。在图3-18中，窗口1－3的优先级高于显示器窗口，而窗口1和窗口3的优先级高于窗口2。因此，显示器窗口中的数据需先对显

示器窗口作内裁剪，然后再对窗口1～3作外裁剪；窗口2内的数据除需对窗口本身作内裁剪外，还需对窗口1和窗口3作外裁剪；至于窗口1和窗口3内的数据，只需各自作内裁剪就可以了。

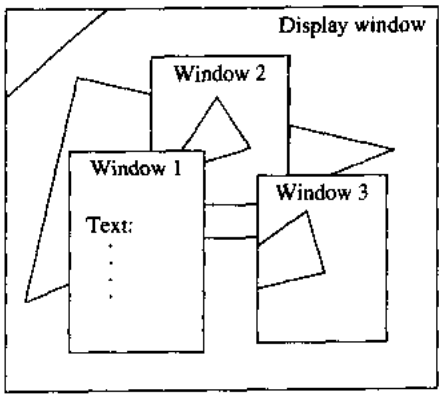


图3-18　多窗口裁剪

外裁剪还可用于凹多边形窗口裁剪线段。图3-19给出一个凹多边形，其顶点为$V_1V_2V_3V_4V_5V_6V_1$。连接V_3和V_5(图3-19中的虚线)，可由凹多边形构造出一个凸多边形。应用Cyrus-Beck算法，先将P_1P_2对此凸多边形作内裁剪得$P_1'P_2'$，再将$P_1'P_2'$对多边形$V_3V_5V_4V_3$作外裁剪，得到$P_1'P_2''$，即为所求。

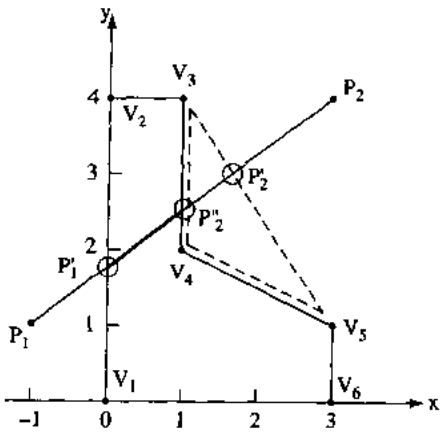
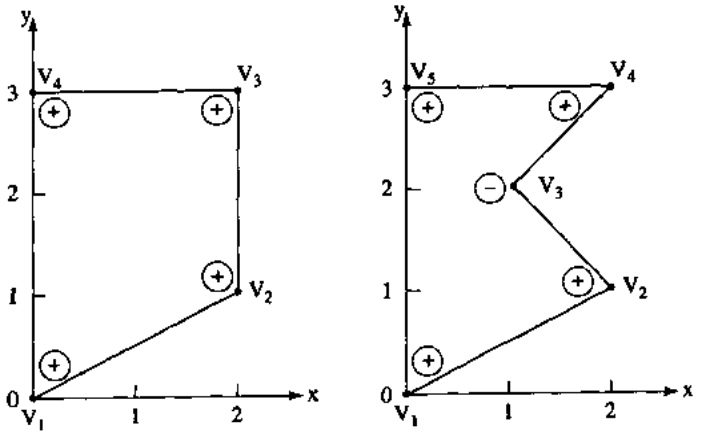


图3-19　线段在凹多边形内的裁剪

3.9　凸多边形的判定和内法线确定

Cyrus-Beck裁剪算法要求裁剪区域是凸的，并需求出区域每一边界上的内法线。对于二维的多边形窗口，如果边界上任意两点的连线均位于多边形的内部，那么它是凸的。实际判定的时候，只需依序计算多边形相邻两边矢量的叉积，根据矢量叉积的符号即可判定多边形是否为凸的。具体如下（参考图3-20）：若各叉积的符号

全部为零　　——多边形各边共线
一部分为正，一部分为负　　——凹多边形
全部大于等于零　　——凸多边形，沿着边的正向，内法线指向其左侧
全部小于等于零　　——凸多边形，沿着边的正向，内法线指向其右侧



a) 凸多边形　　b) 凹多边形

图3-20 矢量叉积的符号

也可以取多边形的一顶点为基点，计算由基点至多边形相邻两顶点矢量的叉积，上述判断规则不变。

矢量的叉积垂直于多边形所在的平面。设两平面矢量为$\mathbf{V}_1$和$\mathbf{V}_2$，则其叉积为$(V_{x_1}V_{y_2}-V_{x_2}V_{y_1})\mathbf{k}$，其中$\mathbf{k}$为垂直于两矢量所在的平面的单位矢量。

利用两个互相垂直矢量的点积为零这一事实可计算出多边形边的法矢量。若法矢量的分量为n_x和n_y，而已知矢量的两分量分别为V_{e_x}和V_{e_y}，则

$$\mathbf{n}\cdot\mathbf{V}_e=(n_x\mathbf{i}+n_y\mathbf{j})\cdot(V_{e_x}\mathbf{i}+V_{e_x}\mathbf{j})=n_xV_{e_x}+n_yV_{e_y}=0$$

即

$$n_xV_{e_x}=-n_yV_{e_y}$$

由于所需求的是法矢量的方向，不失一般性，可假定n_y为1。故边的法矢量为

$$\mathbf{n}=-\frac{V_{e_y}}{V_{e_x}}\mathbf{i}+\mathbf{j}$$

设边矢量由顶点V_{i-1}和V_i构成，若矢量$V_{i-1}V_{i+1}$与法矢量$\mathbf{n}$的点积为正，则$\mathbf{n}$为内法矢量，否则$\mathbf{n}$为外法矢量。这时，改变其x、y分量的符号可求得内法矢量。下面是一个简单的例子。

例3.17 矢量的叉积。

图3-20a为一简单的凸多边形，图3-20b为一凹多边形，表3-9和表3-10中分别给出了完整的结果。作为一个具体例子，确定在图3-20a中V_2处的叉积和V_1V_2边的内法矢量。

在V_2处的相邻边为

$$\mathbf{V_1V_2}=2\mathbf{i}+\mathbf{j}\qquad\mathbf{V_2V_3}=2\mathbf{j}$$

其叉积为

$$\mathbf{V_1V_2}\otimes\mathbf{V_2V_3}=4\mathbf{k}$$

式中$\mathbf{k}$为垂直于两矢量所在平面的单位矢量，此叉积为正。从表3-9中可看出，多边形所有顶点处的叉积均为正，因此该多边形为凸。

表 3-9

V_1	$\mathbf{V_4V_1}\otimes\mathbf{V_1V_2}$	$[\ 0\ \ -3]\otimes[\ 2\ \ 1]=+6$
V_2	$\mathbf{V_1V_2}\otimes\mathbf{V_2V_3}$	$[\ 2\ \ 1]\otimes[\ 0\ \ 2]=+4$
V_3	$\mathbf{V_2V_3}\otimes\mathbf{V_3V_4}$	$[\ 0\ \ 2]\otimes[-2\ \ 0]=+4$
V_4	$\mathbf{V_3V_4}\otimes\mathbf{V_4V_1}$	$[-2\ \ 0]\otimes[\ 0\ \ -3]=+6$

表3-10对应于图3-20b所示的多边形，其中V_3处叉积为负，而其余顶点处的叉积均为正，故此多边形为凹。

$\mathbf{V_1V_2}$边矢量的法矢量为

$$\mathbf{n}=-\frac{1}{2}\mathbf{i}+\mathbf{j}$$

表 3-10

顶　点	矢　量	叉　积
V_1	$\mathbf{V_5V_1}\otimes\mathbf{V_1V_2}$	$[\ 0\ \ -3]\otimes[\ 2\ \ 1]=+6$
V_2	$\mathbf{V_1V_2}\otimes\mathbf{V_2V_3}$	$[\ 2\ \ 1]\otimes[-1\ \ 1]=+3$
V_3	$\mathbf{V_2V_3}\otimes\mathbf{V_3V_4}$	$[-1\ \ 1]\otimes[\ 1\ \ 1]=-2$
V_4	$\mathbf{V_3V_4}\otimes\mathbf{V_4V_5}$	$[\ 1\ \ 1]\otimes[-2\ \ 0]=+2$
V_5	$\mathbf{V_4V_5}\otimes\mathbf{V_5V_1}$	$[-2\ \ 0]\otimes[\ 0\ \ -3]=+6$

或

$$\mathbf{n}=-\mathbf{i}+2\mathbf{j}$$

矢量$\mathbf{V_1V_3}$为

$$\mathbf{V_1V_3}=2\mathbf{i}+3\mathbf{j}$$

所以，

$$\mathbf{n}\cdot\mathbf{V_1V_3}=(-\mathbf{i}+2\mathbf{j})\cdot(2\mathbf{i}+3\mathbf{j})=4>0$$

故n是内法矢量。 □

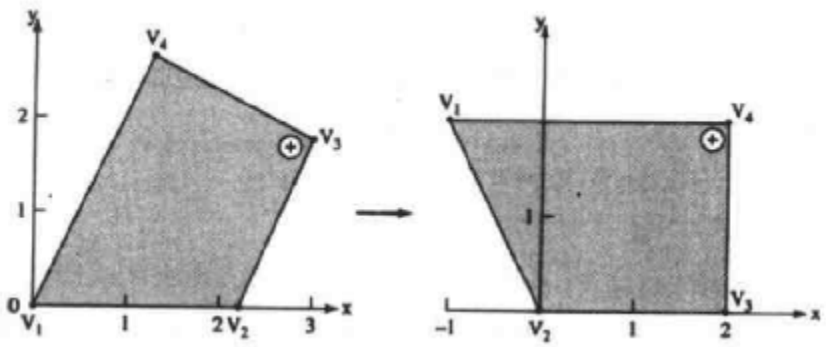
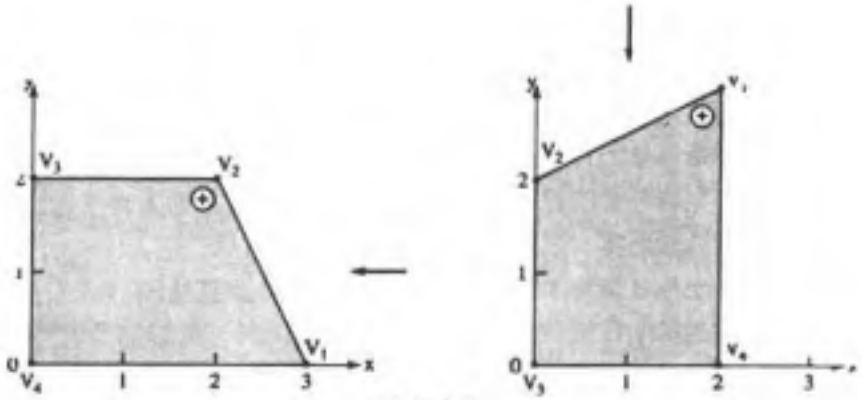
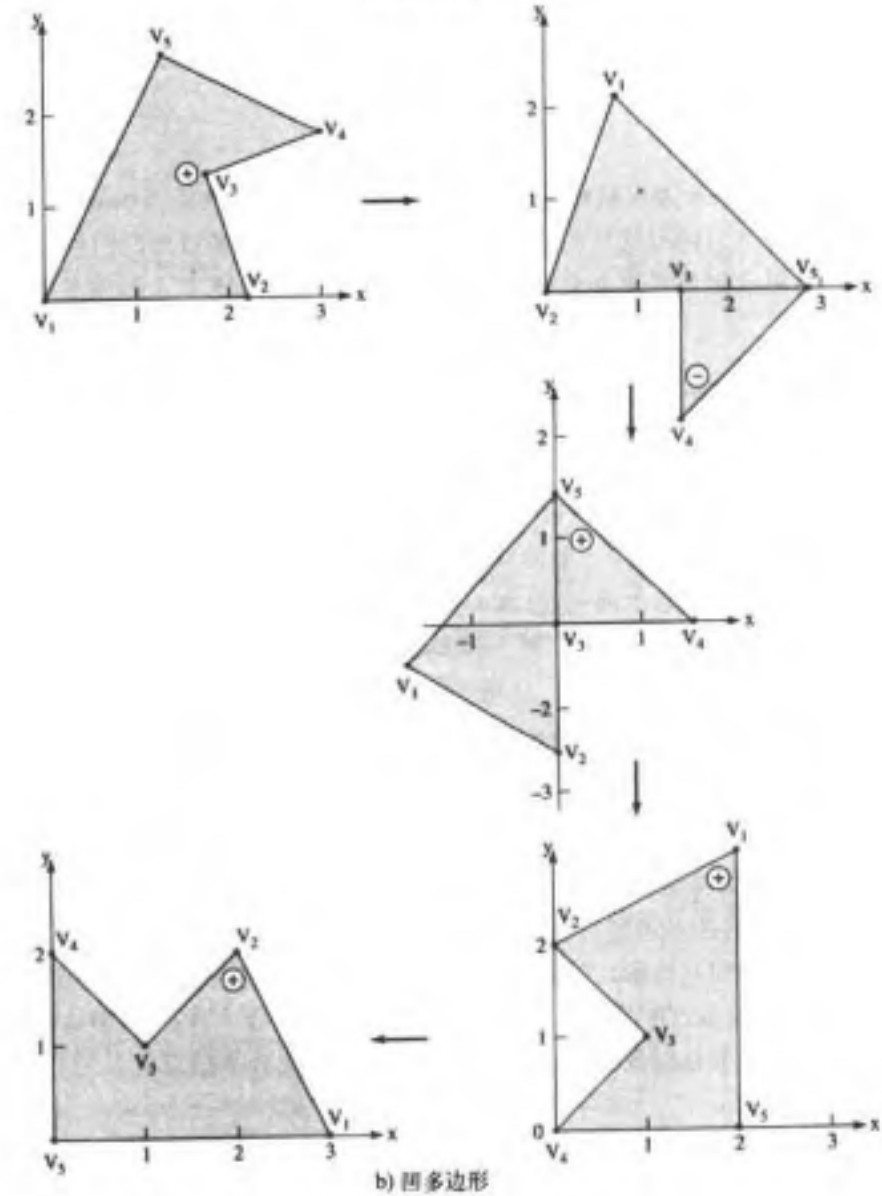


图3-21　利用旋转和平移来判定凸多边形和凹多边形



a) 凸多边形



b) 凹多边形

图3-21 （续）

也可通过平移和旋转多边形窗口来判定窗口的凹凸性并确定各边上的内法向。具体如下:

对多边形的每一个顶点V_i，平移多边形使V_i位于坐标原点。
绕原点旋转多边形，使V_{i+1}落在x轴的正半轴上。
检查第i+2个顶点y坐标分量的符号。
若所有第i+2个顶点y坐标分量的符号相同，则为凸多边形，否则为凹多边形。
若第i+2个顶点y坐标分量为零，则第i、i+1、i+2三顶点共线。
若所有第i+2个顶点y坐标分量均为零，则多边形退化为一条直线段。
凸多边形每一条边的内法向在旋转后的坐标系中与y轴同向矢量的符号取为第i+2个顶点y分量的符号。
通过沿反方向旋转，可求出内法向的原始方向。

对图3-20所示的凸、凹多边形应用上述算法，不同阶段的结果均示于图3-21中。旋转和平移算法由Rogers给出（见[Roge90a]）。

3.10 凹多边形分割

许多裁剪算法都要求多边形裁剪区域为凸的，例如上节讨论的Cyrus-Beck算法，而且在后面各节中还有很多地方都有这样的要求。将上节中用来判定多边形凸凹性的旋转、平移方法作简单推广，就可以把凹多边形分割为若干个凸多边形。假定多边形顶点按逆时针方向排列，算法可表述为:

对多边形的每一个顶点V_i，平移多边形使Vi位于坐标原点。
绕原点旋转多边形，使V_{i+1}落在x轴的正半轴上。
检查第i+2个顶点y坐标分量的符号。如果符号为正或零，那么多边形在该边处是凸的；如果符号为负，那么多边形在该边处是凹的。分割多边形。
分割多边形的时候，检查第i+2个顶点后面的各顶点的y坐标分量，直至发现一个顶点的y坐标分量大于等于零。这个顶点在x轴的上方，或恰好落在x轴上，记为V_{i+j}。那么，分割下来的多边形为$V_{i+1}V_{i+2}\cdots V_{i+j}V_i$；剩下的多边形为$V_iV_{i+1}V_{i+j}\cdots V_i$。
对分割的多边形再次使用该算法，直到它们都为凸多边形。

上述算法的分割结果并不是最优的分割，即生成的凸多边形数目不是最少。此外，这一算法也不适用于自交多边形的分割。例如学生画的星形和领结等非简单多边形。

通过下例可进一步说明上述算法。

例3.18　凹多边形的分割。

考虑图3-21b所示的凹多边形。当顶点V_2位于坐标原点，V_3在x轴正半轴上时，V_4的y坐标分量为负，故多边形为凹多边形。检查后面各顶点的y坐标分量，发现V_5的y坐标分量等于零，即$i+j=5$。分割下来的多边形为$V_3V_4V_5$；剩下的多边形为$V_1V_2V_3V_5$。对多边形$V_3V_4V_5$和多边形$V_1V_2V_3V_5$再次使用该算法时发现它们都为凸多边形。因此分割结束。　□

3.11 三维裁剪

在将前面所述的裁剪方法推广到三维之前，有必要先讨论一下三维裁剪体的形状。常用的三维裁剪体有长方体和平截头棱锥体，如图3-22所示,其中长方体或裁剪盒适用于平行投影

或轴测投影。平截头棱锥体又称视域四棱锥 (frustum of vision)，它适用于透视投影。这两种裁剪体均为六面体:即包括左侧面、右侧面、顶面、底面、前面（近平面）和后面（远平面）。当然还有其他的裁剪体，如太空飞船的货舱，但不常见。

如同二维裁剪一样，完全可见线段或显然不可见线段可采用推广的Cohen-Sutherland端点编码方法加以判别。在三维裁剪中，需采用六位端点编码，第一位还是对应最右端位。各位置1的规则与二维裁剪中的编码方法类似，具体为:

第一位置1——若端点位于裁剪体的左边
第二位置1——若端点位于裁剪体的右边
第三位置1——若端点位于裁剪体的下边
第四位置1——若端点位于裁剪体的上边
第五位置1——若端点位于裁剪体的前边
第六位置1——若端点位于裁剪体的后边

否则，相应位置零。同样，若一条线段的两端点编码都为零，则此线段的两端点均可见，且此线段也可见；若两端点编码逐位逻辑与不为零，则线段为完全不可见线段；若逐位逻辑与为零，则线段可能部分可见或完全不可见。这时需将线段对裁剪体求交,才能最后确定。

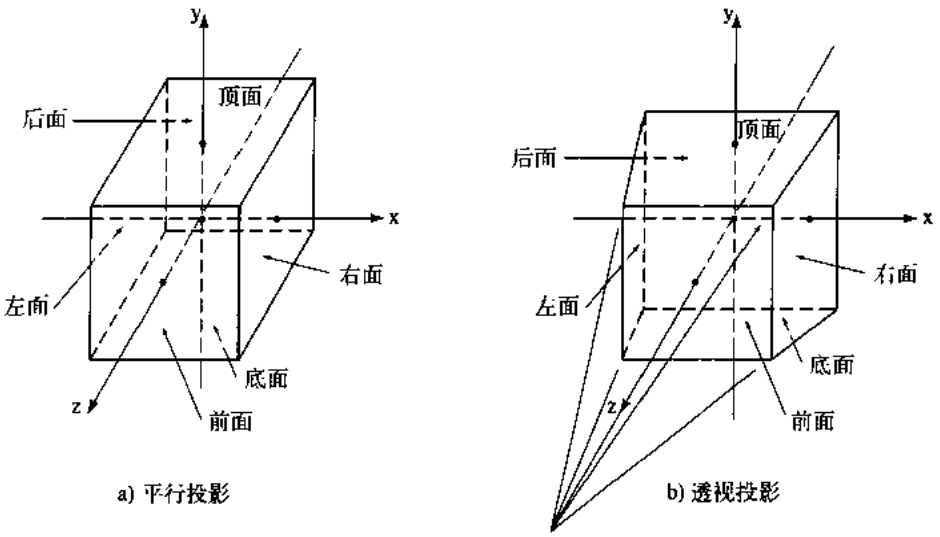


图3-22 三维裁剪

对于长方裁剪体，决定线段的端点编码不过是二维算法的直接推广，然而对于图3-22b所示的透视裁剪体来说，却要另作考虑。一个方法（见[Newm73]和[Newm79]）是将裁剪体变换成一规范体，即当$z_{far}=1$时，$x_{right}=1$，$x_{left}=-1$，$y_{top}=1$，$y_{bottom}=-1$。取左手坐标系，若$z_{near}=\alpha$，$0<\alpha\leqslant 1$，且投影中心位于坐标原点上，这时端点编码的条件可大为简化。

一个更为直接的方法是对裁剪体稍做变形，使得在一个右手坐标系中，连接投影中心和透视裁剪体的中心的直线与z轴重合，如图3-22b所示。

图3-23是一个透视裁剪体的俯视图。在此图中，对应于裁剪体右侧面的方程为

$$x=\frac{z-z_{CP}}{z_Y-z_{CP}}x_R=z\alpha_1+\alpha_2$$

其中 $$\alpha_1 = \frac{x_R}{z_Y - z_{CP}},\ a_2 = -a_1 z_{CP}$$

这一方程可用来决定一个空间点是否位于该平面右边、上面或左边，即是否位于裁剪的外侧、上面或内部。将点P的x、z坐标代入$x - z\alpha_1 - \alpha_2$，得

$$\begin{aligned} f_R = x - z\alpha_1 - \alpha_2 &> 0, \text{若}P\text{位于平面右方} \\ &= 0, \text{若}P\text{位于平面上} \\ &< 0, \text{若}P\text{位于平面左方} \end{aligned}$$

对于左侧面、顶面和底面、其判别函数分别为

$$\begin{aligned} f_L = x - z\beta_1 - \beta_2 &< 0, \text{若}P\text{位于平面左方} \\ &= 0, \text{若}P\text{位于平面上} \\ &> 0, \text{若}P\text{位于平面右方} \end{aligned}$$

其中 $$\beta_1 = \frac{x_L}{Z_Y - z_{CP}},\ \beta_2 = -\beta_1 z_{CP}$$

和

$$\begin{aligned} f_T = y - z\gamma_1 - \gamma_2 &> 0, \text{若}P\text{位于平面上方} \\ &= 0, \text{若}P\text{位于平面上} \\ &< 0, \text{若}P\text{位于平面下方} \end{aligned}$$

其中

$$\gamma_1 = \frac{y_T}{z_Y - z_{CP}},\quad \gamma_2 = -\gamma_1 Z_{CP}$$

以及

$$\begin{aligned} f_B = y - z\delta_1 - \delta_2 &< 0, \text{若}P\text{位于平面下方} \\ &= 0, \text{若}P\text{位于平面上} \\ &> 0, \text{若}P\text{位于平面上方} \end{aligned}$$

其中 $$\delta_1 = z_{CP} \qquad \delta_2 = -\delta_1 z_{CP}$$

最后，前面和后面的判别函数为

$$\begin{aligned} f_H = z - z_H &> 0, \text{若}P\text{位于平面前方} \\ &= 0, \text{若}P\text{位于平面上} \\ &< 0, \text{若}P\text{位于平面后方} \end{aligned}$$

和

$$\begin{aligned} f_Y = z - z_Y &< 0, \text{若}P\text{位于平面后方} \\ &= 0, \text{若}P\text{位于平面上} \\ &> 0, \text{若}P\text{位于平面前方} \end{aligned}$$

当z_{CP}趋于无穷时，裁剪体趋于长方体，相应的判别函数变成长方体的判别函数。

如同Liang和Barsky（见[Lian84]）所指出的，当端点位于投影中心之后时，上述方法可能产生错误的编码。这是由于透视裁剪体的左侧面、右侧面、顶面、底面交于投影中心。这时

点可能同时位于左侧面的左边和右侧面的右边。Liang和Barsky提出了一个纠正方法，简而言之,当$z < z_{CP}$时，只需将对应于左面-右面、顶面-底面的码位互相取补就可以了(见3.15节)。

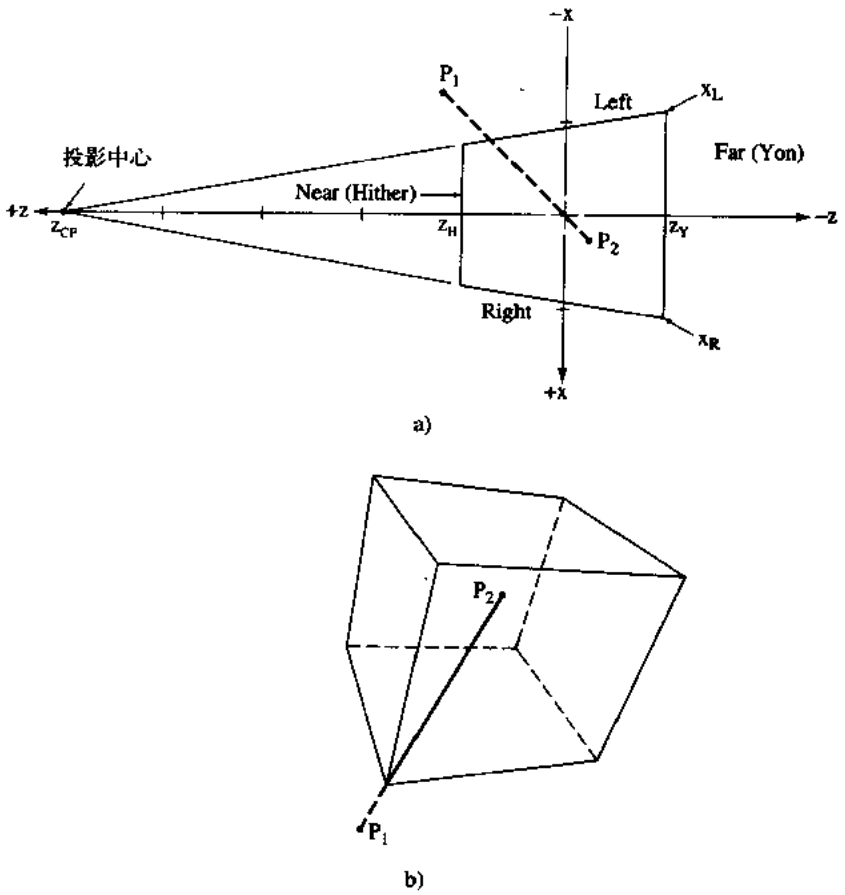


图3-23 透视裁剪体视图

3.12 三维中点分割算法

3.3节给出的中点分割算法可直接推广到三维情形，它的伪代码程序也只需改变数组Pcodes和Window数组的维数，并按三维情形改写Endpoint 和Sum两子程序。三维情形的端点编码子程序的伪代码程序如下:

subroutine module for calculating three-dimensional perspective volume end point codes

subroutine Endpoint(P, Window; Pcode)

P_x, P_y, P_z *are the* x, y *and* z *components of the point* P

Window *is a* 1×7 *array containing the left, right, bottom, top, near, far edges and the center of projection,* x_L, x_R, y_B, y_T, z_N, z_F, z_{CP}, *respectively*

Pcode *is a* 1×6 *array containing the end point code*

calculate α_1, α_2, β_2, γ_1, γ_2, δ_1, δ_2

$\alpha_1 = x_R/(z_Y - z_{CP})$

$\alpha_2 = -\alpha_1 z_{CP}$

```
    β1 = xL/(zY − zCP)
    β2 = −β1zCP
    γ1 = yT/(zY − zCP)
    γ2 = −γ1zCP
    δ1 = yB/(zY − zCP)
    δ2 = −δ1zCP
    determine the end point codes
    if Px − Pzβ1 − β2 < 0 then Pcode(6) = 1 else Pcode(6) = 0
    if Px − Pzα1 − α2 > 0 then Pcode(5) = 1 else Pcode(5) = 0
    if Py − Pzδ1 − δ2 < 0 then Pcode(4) = 1 else Pcode(4) = 0
    if Py − Pzγ1 − γ2 > 0 then Pcode(3) = 1 else Pcode(3) = 0
    if Pz − zN > 0 then Pcode(2) = 1 else Pcode(2) = 0
    if Pz − zF < 0 then Pcode(1) = 1 else Pcode(1) = 0
return
subroutine Sum(Pcode; Sum)
Pcode is a 1 × 6 array containing the end point code
Sum is the element-by-element sum of Pcode
    calculate the sum
    Sum = 0
    for i = 1 to 6
        Sum = Sum + Pcode(i)
    next i
return
```

下面给出一个三维中点裁剪算法的例子。

例3.19　三维中点分割。

考虑从P_1(−600, −600, 600)到P_2(100, 100, −100)的直线段在透视裁剪体内的裁剪，其中坐标按屏幕坐标单位取值。在远裁剪平面上有裁剪体的坐标为$x_R = y_T$=500，$x_L = Y_B$ = −500。近平面为z_H = 357.14，远平面为z_Y = −500，透视中心为z_{cp} = 2500。图3-23a为裁剪体的俯视图，图3-23b为透视图。裁剪体各面的判别函数为

右面(Right):　$f_R = 6x + z - 2500$

左面(Left):　$f_L = 6x - z + 2500$

顶面(Top):　$f_T = 6y + z - 2500$

底面(Bottom):　$f_B = 6y - z + 2500$

前面(Near):　$f_H = z - 357.14$

后面(Far):　$f_Y = z + 2500$

端点P_1的编码为 (010101)，端点P_2的编码为 (000000)。因为两个端点的编码不全为零，所以线段不是完全可见段。又因为两端点编码的逻辑与为(000000)，所以该线段也不是显然不可见段。由P_2的端点编码为 (000000) 可知P_2在裁剪体内，是离P_1最远的可见点。线段P_1P_2与裁剪体只有一个交点。采用整数运算计算P_1P_2的中点为

$$x_m = \frac{x_2 + x_1}{2} = \frac{100 + (-600)}{2} = -250$$

$$y_m = \frac{y_2 + y_1}{2} = \frac{100 + (-600)}{2} = -250$$

$$z = \frac{z_2 + z_1}{2} = \frac{-100 + 600}{2} = 250$$

中点的端点编码为 (000000)，故P_mP_2为完全可见段，而P_1P_m为部分可见段。继续细分P_1P_m。整个分割过程如表3-11所示。

表 3-11

P_1			P_2			P_m			注 释
-600	-600	600	-100	-100	-100	-250	-250	250	继续P_1P_m
-600	-600	600	-250	-250	250	-425	-425	425	继续P_mP_2
-425	-425	425	-250	-250	250	-338	-338	337	继续P_1P_m
-425	-425	425	-338	-338	337	-382	-382	381	继续P_mP_2
-382	-382	381	-338	-338	337	-360	-360	359	继续P_mP_2
-360	-360	359	-338	-338	337	-349	-349	348	继续P_1P_m
-360	-360	359	-349	-349	348	-355	-355	353	继续P_1P_m
-360	-360	359	-355	-355	353	-358	-358	356	继续P_mP_2
-358	-358	359	-355	-355	353	-357	-357	354	继续P_1P_m
-358	-358	356	-357	-357	354	-358	-358	355	继续P_mP_2
-358	-358	355	-357	-357	354	-358	-358	354	成功

实际交点应为(-357.14, -357.14, 357.14)，误差是因为整数运算引起的。 □

3.13 三维Cyrus-Beck算法

在前面推导Cyrus-Beck二维裁剪算法[Cyrus78]的过程中，除了要求裁剪区域为凸区域外，对裁剪区域的形状并无特殊限制，因此可以是三维凸体。前面所推导出的算法可直接应用于三维情形。原来表示裁剪窗口的边数k现替换为裁剪体的面数(见图3-12)。所有矢量均具有x、y、z三分量，子程序Dotproduct直接推广到三维矢量。为了完整地说明该算法，下面举一些例子。首先考虑裁剪体为长方体的情形。

例3.20 三维Cyrus-Beck算法。

如图3-24所示,从 $P_1\left(-2, -1, \frac{1}{2}\right)$到 $P_2\left(\frac{3}{2}, \frac{3}{2}, -\frac{1}{2}\right)$的线段在裁剪体$(x_L, x_R, y_B, y_T, z_H, z_Y)$ = (-1, 1, -1, 1, 1, -1)内被裁剪。由观察可得六个内法矢量为

顶面: $\mathbf{n}_T = -\mathbf{j} = [\ 0\ \ -1\ \ 0\ ]$

底面: $\mathbf{n}_B = \ \mathbf{j} = [\ 0\ \ 1\ \ 0\ ]$

右面: $\mathbf{n}_R = -\mathbf{i} = [-1\ \ 0\ \ 0]$

左面: $\mathbf{n}_L = \ \mathbf{i} = [\ 1\ \ 0\ \ 0\ ]$

前面: $\mathbf{n}_H = -\mathbf{k} = [\ 0\ \ 0\ -1\ ]$

后面: $\mathbf{n}_Y = \ \mathbf{k} = [\ 0\ \ 0\ \ 1\ ]$

各裁剪面上的点可由观察决定。如果选取裁剪体对角线的两端点，则两点就足够了，即

$$f_T = f_R = f_H(1,1,1)$$

和

$$f_B = f_L = f_Y(-1,-1,-1)$$

另外也可取每一个裁剪平面的中心。

P_1P_2的方向矢量为

$$\mathbf{D}=\mathbf{P}_2-\mathbf{P}_1=\left[\frac{3}{2}\quad\frac{3}{2}\quad-\frac{1}{2}\right]-\left[-2\quad-1\quad\frac{1}{2}\right]$$
$$=\left[\frac{7}{2}\quad\frac{5}{2}\quad-1\right]$$

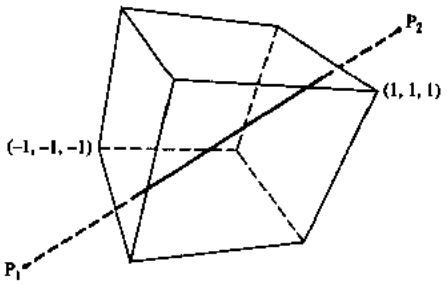


图3-24　Cyrus-Beck 裁剪——三维矩形裁剪体

对于边界点$f_L(-1,-1,-1)$，有

$$\mathbf{w}=\mathbf{P}_1-\mathbf{f}=\left[-2\quad-1\quad\frac{1}{2}\right]-[-1\quad-1\quad-1]$$
$$=\left[-1\quad0\quad\frac{3}{2}\right]$$

左侧裁剪平面的内法矢量为

$$\mathbf{n}_L=[1\quad0\quad0]$$

故

$$\mathbf{D}\cdot\mathbf{n}_L=\left[\frac{7}{2}\quad\frac{5}{2}\quad-1\right]\cdot[1\quad0\quad0]=\frac{7}{2}>0$$

求下限值，

$$\mathbf{w}\cdot\mathbf{n}_L=\left[-1\quad0\quad\frac{3}{2}\right]\cdot[1\quad0\quad0]=-1$$

$$t_L=-\frac{1}{7/2}=\frac{2}{7}$$

完整的结果见表3-12。

表 3-12

平面	n	f	w	w·n	D·n	t_L①	t_U
顶面	[0 −1 0]	(1, 1, 1)	[−3 −2 −1/2]	2	−5/2		4/5
底面	[0 1 0]	(−1,−1,−1)	[−1 0 3/2]	0	5/2	0	
右面	[−1 0 0]	(1, 1, 1)	[−3 −2 −1/2]	3	−7/2		6/7
左面	[1 0 0]	(−1,−1,−1)	[−1 0 3/2]	−1	7/2	2/7	
前面	[0 0 −1]	(1, 1, 1)	[−3 −2 −1/2]	1/2	1	−1/2	
后面	[0 0 1]	(−1,−1,−1)	[−1 0 3/2]	3/2	−1		3/2

① $\mathbf{D}\cdot\mathbf{n}<0$，对应上限$t_U$；$\mathbf{D}\cdot\mathbf{n}>0$，对应下限$t_L$。

从表3-12可以看出，最大下限为$t_L = \frac{2}{7}$，最小上限为$t_U = \frac{4}{5}$。P_1P_2的参数方程为：

$$P(t) = \begin{bmatrix} -2 & -1 & \frac{1}{2} \end{bmatrix} + \begin{bmatrix} \frac{7}{2} & \frac{5}{2} & -1 \end{bmatrix} t$$

分别代入t_L和t_U，得到线段与左侧裁剪面的交点：

$$P\left(\frac{2}{7}\right) = \begin{bmatrix} -2 & -1 & \frac{1}{2} \end{bmatrix} + \begin{bmatrix} \frac{7}{2} & \frac{5}{2} & -1 \end{bmatrix} \left(\frac{2}{7}\right)$$
$$= \begin{bmatrix} -1 & -\frac{2}{7} & \frac{3}{14} \end{bmatrix}$$

和线段与裁剪体顶面的交点：

$$P\left(\frac{4}{5}\right) = \begin{bmatrix} -2 & -1 & \frac{1}{2} \end{bmatrix} + \begin{bmatrix} \frac{7}{2} & \frac{5}{2} & -1 \end{bmatrix} \left(\frac{4}{5}\right)$$
$$= \begin{bmatrix} \frac{4}{5} & 1 & -\frac{3}{10} \end{bmatrix}$$ □

对标准的透视体的裁剪稍复杂一点。除了远近平面外，其余各边界面的内法矢量需由计算决定，而不能简单地由观察得出。

例3.21 对透视体的裁剪。

仍考虑例3.20所给的、由$P_1\left(-2, -1, \frac{1}{2}\right)$到$P_2\left(\frac{3}{2}, \frac{3}{2}, -\frac{1}{2}\right)$的线段在透视体$(x_L, x_R, y_B, y_T, z_H, z_Y) = (-1, 1, -1, 1, 1, -1)$内的裁剪，投影中心$z_{CP} = 5$。见图3-22b。

裁剪体近平面和远平面的内法矢量可由观察得出。其余四个裁剪面的内法矢量可由计算连接投影中心到投影平面$z = 0$的四个角的矢量的叉积得到，分别为

$$\mathbf{V1} = [\ 1\ \ 1\ -5]$$
$$\mathbf{V2} = [-1\ \ 1\ -5]$$
$$\mathbf{V3} = [-1\ -1\ -5]$$
$$\mathbf{V4} = [\ 1\ -1\ -5]$$

故裁剪面的内法矢量为

$$T = \mathbf{V1} \otimes \mathbf{V2} = [\ \ 0\ -10\ -2]$$
$$L = \mathbf{V2} \otimes \mathbf{V3} = [\ 10\ \ \ 0\ -2]$$
$$B = \mathbf{V3} \otimes \mathbf{V4} = [\ \ 0\ \ 10\ -2]$$
$$R = \mathbf{V4} \otimes \mathbf{V1} = [-10\ \ \ 0\ -2]$$
$$H = [0\ 0\ -1]$$
$$Y = [0\ 0\ \ 1]$$

由于投影中心同时位于裁剪体的四个平面上，故四个面上的边界点可取为

$$f_T = f_L = f_B = f_R(0, 0, 5)$$

前面和后面的边界点可取它们的中心：

$$f_H(0, 0, 1) \quad 和 \quad f_Y(0, 0, -1)$$

P_1P_2的方向矢量为

$$\mathbf{D} = \mathbf{P}_2 - \mathbf{P}_1 = \begin{bmatrix} \frac{7}{2} & \frac{5}{2} & -1 \end{bmatrix}$$

对位于左侧裁剪面上的边界点有

$$\mathbf{w} = \mathbf{P}_1 - \mathbf{f}_L = \left[-2 \quad -1 \quad \frac{1}{2}\right] - [0 \ 0 \ 5]$$
$$= \left[-2 \quad -1 \quad -\frac{9}{2}\right]$$

注意到

$$\mathbf{D} \cdot \mathbf{n}_L = \left[\frac{7}{2} \quad \frac{5}{2} \quad -1\right] \cdot [10 \ 0 \ -2] = 37 > 0$$

求得下限t值

$$\mathbf{w} \cdot \mathbf{n}_L = \left[-2 \quad -1 \quad -\frac{9}{2}\right] \cdot [10 \ 0 \ -2] = -11$$

$$t_L = -\frac{-11}{37} = \frac{11}{37} = 0.297$$

完整的结果见表3-13 。

从表3-13可以看出，最大的下限值t_L = 0.297，而最小的上限值t_U = 0.826。由线段的参数方程可求出两交点为

$$P(0.297) = [-0.961 \quad -0.258 \quad 0.203]$$

及

$$P(0.826) = [0.891 \quad 1.065 \quad -0.323]$$

它们分别位于左边和顶部的裁剪面上。 □

表 3-13

平面	n	f	w	w·n	D·n①	t_L	t_U
顶面	[0 −10 −2]	(0, 0, 5)	[−2 −1 −9/2]	19	−23		0.826
底面	[0 10 −2]	(0, 0, 5)	[−2 −1 −9/2]	−1	27	0.037	
右面	[−10 0 −2]	(0, 0, 5)	[−2 −1 −9/2]	29	−33		0.879
左面	[10 0 −2]	(0, 0, 5)	[−2 −1 −9/2]	−11	37	0.297	
前面	[0 0 −1]	(0, 0, 1)	[−2 −1 −1/2]	1/2	1	−0.5	
后面	[0 0 1]	(0, 0, −1)	[−2 −1 3/2]	3/2	−1		1.5

① 如果$\mathbf{D} \cdot \mathbf{n} < 0$，对应上限值$t_U$；如果$\mathbf{D} \cdot \mathbf{n} > 0$，对应下限值$t_L$。

最后一例将考虑具有七个裁剪面的非标准裁剪体。

例3.22 对任意体的裁剪。

裁剪体如图3-25所示，是割去一角的立方体。各个表面多边形的顶点如下所示。

右面: (1, −1, 1), (1, −1, −1), (1, 1, −1), (1, 1, 1)

左面: (−1, −1, 1), (−1, −1, −1), (−1, 1, −1), (−1,1, 0), (−1, 0, 1)

底面: (1, −1, 1), (1, −1, −1), (−1, −1, −1), (1, −1, −1)

顶面: (1, 1, 1), (1, 1, −1), (−1, 1, −1), (−1, 1, 0), (0, 1, 1)
前面: (1, −1, 1), (1, 1, 1), (0, 1, 1), (−1, 0, 1), (−1, −1, 1)
后面: (−1, −1, −1), (1, −1, −1), (1, 1, −1), (−1, 1, −1)
斜截面: (−1, 0, 1), (0, 1, 1), (−1, 1, 0)

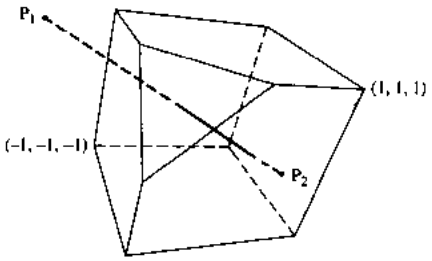


图3-25 Cyrus-Beck 裁剪——非标准三维裁剪体

表3-14给出了线段 $P_1\left(-2, \frac{3}{2}, 1\right)P_2\left(\frac{3}{2}, -1, -\frac{1}{2}\right)$ 在此裁剪体内被裁剪的整个过程。从表3-14可以看出，最大的下限值 $t_L = \frac{1}{3}$，而最小的上限值 $t_U = \frac{6}{7}$，所以两交点为

$$P\left(\frac{1}{3}\right) = \left[-\frac{5}{6} \quad \frac{2}{3} \quad \frac{1}{2}\right] \text{ 和 } P\left(\frac{6}{7}\right) = \left[1 \quad -\frac{9}{14} \quad -\frac{2}{7}\right]$$

它们分别位于斜截面和右侧的裁剪面上。 □

表 3-14

平面	n	f	w	w·n	D·n	t_L	t_U
顶面	[0 −1 0]	(1, 1, 1)	[−3 ½ 0]	−½	5/2	⅕	
底面	[0 1 0]	(−1, −1, −1)	[−1 5/2 2]	5/2	−5/2		1
右面	[−1 0 0]	(1, 1, 1)	[−3 ½ 0]	3	−7/2		6/7
左面	[1 0 0]	(−1, −1, −1)	[−1 5/2 2]	−1	7/2	2/7	
前面	[0 0 −1]	(1, 1, 1)	[−3 ½ 0]	0	3/2	0	
后面	[0 0 1]	(−1, −1, −1)	[−1 5/2 2]	2	−3/2		4/3
斜截面	[1 −1 −1]	(−1, 0, 1)	[−1 3/2 0]	−5/2	15/2	⅓	

① 如果D · n < 0, 对应上限值t_U; 如果D · n > 0, 对应下限值t_L。

可以看到，Cyrus-Beck裁剪算法的计算量随裁剪区域边界的边或面的数目线性增长。

3.14 Liang-Barsky三维裁剪

Liang-Barsky二维裁剪算法可以很容易地扩展到三维空间。考虑一个视域棱锥体，两个视角分别为θ_x、θ_y，投影中心即视点位于坐标原点，见图3-26a[⊖]。视域体内部可定义为

⊖ 注意，此处为左手坐标系。

$$-z_e < \cot\left(\frac{\theta_x}{2}\right)x_e < z_e$$

$$-z_e < \cot\left(\frac{\theta_y}{2}\right)y_e < z_e$$

其中下标e表示“眼睛”坐标系。上面的不等式隐含$-z_e < z_e$和$z_e > 0$，即场景在正半平面中。

若做一简单的变换:

$$x = \cot\left(\frac{\theta_x}{2}\right)x_e$$

$$y = \cot\left(\frac{\theta_y}{2}\right)y_e$$

$$z = z_e$$

则视域棱锥体的不等式化为

$$-z < x < z$$

$$-z < y < z$$

上式隐含了$z > 0$。变换后得到一个视角为90°的视域棱锥体，见图3-26b。

再次考虑参数直线段$P(t) = P_1 + (P_2 - P_1)t$，$0 < t < 1$。其分量形式为

$$\begin{aligned} x(t) &= x_1 + (x_2 - x_1)t &&= x_1 + \Delta x t \\ y(t) &= y_1 + (y_2 - y_1)t &&= y_1 + \Delta y t \qquad 0 < t < 1 \\ z(t) &= z_1 + (z_2 - z_1)t &&= z_1 + \Delta z t \end{aligned}$$

将它们代入前面的不等式组，得到

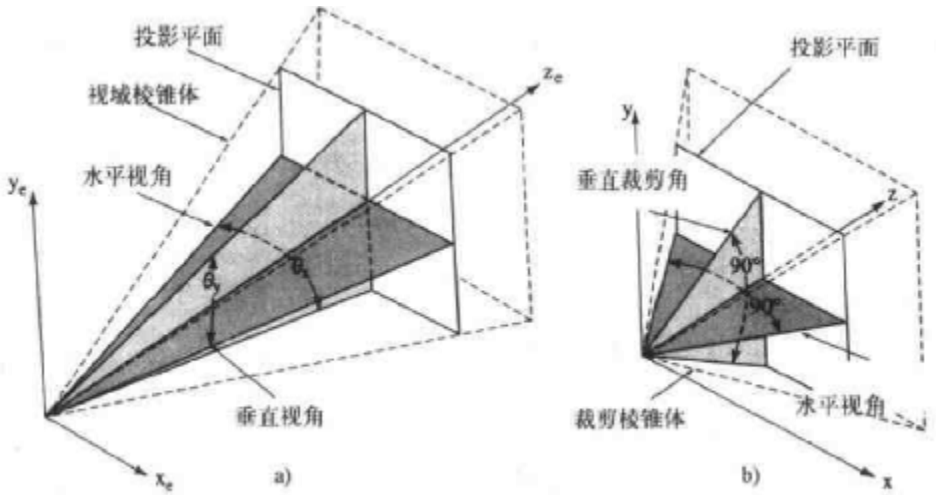


图3-26　三维裁剪体

a) 视域棱锥体　b)变换后张角为90°的视域棱锥体

$$-(\Delta x + \Delta z)t < x_1 + z_1 \qquad \text{和} \qquad (\Delta x - \Delta z)t < z_1 - x_1$$

$$-(\Delta y + \Delta z)t < y_1 + z_1 \qquad \text{和} \qquad (\Delta y - \Delta z)t < z_1 - y_1$$

如前面在二维Liang-Barsky裁剪算法中所做的讨论(3.6节)，把那些不等式写成如下形式:

$$d_i t \leqslant q_i \qquad i = 1, 2, 3, 4 \tag{3-9}$$

其中

$$\begin{aligned} d_1 &= -(\Delta x + \Delta z) & q_1 &= x_1 + z_1 \\ d_2 &= \Delta x - \Delta z & q_2 &= z_1 - x_1 \\ d_3 &= -(\Delta y + \Delta z) & q_3 &= y_1 + z_1 \\ d_4 &= \Delta y - \Delta z & q_4 &= z_1 - y_1 \end{aligned}$$

并且上述四个不等式分别了定义了左侧、右侧、底部和顶部的四条边。

一个视域棱锥体加上近裁剪平面$z=n$和远裁剪平面$z=f$构成一个视域四棱锥，见图3-22。视域四棱锥沿z轴方向其内部为

$$n \leqslant z \leqslant f$$

代入参数直线段的方程，得

$$\begin{aligned} -\Delta z t &\leqslant z_1 - n \\ \Delta z t &\leqslant f - z_1 \end{aligned}$$

表达成不等式(3-9)的形式，得

$$\begin{aligned} d_5 &= -\Delta z & q_5 &= z_1 - n \\ d_6 &= \Delta z & q_6 &= f - z_1 \end{aligned}$$

参数直线和视域四棱锥任一裁剪平面的交点为$t = q_i/d_i$。q_i的符号决定了端点P_1位于相应裁剪平面的哪一侧[⊖]。如果$q_i > 0$，则P_1在平面的可见侧；如果$q_i < 0$，则P_1位于不可见侧；最后，如果$d_i = 0$且$q_i < 0$，则线段在裁剪平面不可见侧且平行于该平面，因而显然不可见。这些条件和约束与二维Liang-Barsky裁剪算法完全相同，因此裁剪程序也一样。三维Liang-Barsky算法可表述如下。

Liang–Barsky three-dimensional clipping algorithm

P_1 *and* P_2 *are the line end points with components* $x_1, y_1, z_1, x_2, y_2, z_2$
t_L *and* t_U *are the lower and upper parametric limits*
x_L, x_R, y_B, y_T *are the left, right, bottom and top window edges*

the **clipt** *function is the same as in the two-dimensional Liang–Barsky algorithm (see Sec. 3–6)*

```
t_L = 0
t_U = 1
deltax = x_2 − x_1
deltaz = z_2 − z_1
if clipt(−deltax − deltaz, x_1 + z_1, t_L, t_U) = true then
    if clipt(deltax − deltaz, z_1 − x_1, t_L, t_U) = true then
        deltay = y_2 − y_1
```

⊖ 由于视域四棱锥的每一侧面均垂直于一坐标面，故这些侧面的方程均不涉及垂直方向的坐标，因而q_i可构成对点与相应侧面位置关系的判别函数。例如视域四棱锥的左侧面垂直于 xz 坐标平面，其平面方程为 $x+z=0$。判别函数为 x_1+z_1。

```
            if clipt(−deltay − deltaz, y1 + z1, tL, tU) = true then
                if clipt(deltay − deltaz, z1 − y1, tL, tU) = true then
                    if tU < 1 then
                        x2 = x1 + tU * deltax
                        y2 = y1 + tU * deltay
                        z2 = z1 + tU * deltaz
                    end if
                    if tL > 0 then
                        x1 = x1 + tL * deltax
                        y1 = y1 + tL * deltay
                        z1 = z1 + tL * deltaz
                    end if
                    Draw P1P2
                end if
            end if
        end if
    end if
finish
```

下面用一个例子进一步讲解该算法。

例3.23　Liang-Barsky三维裁剪。

应用Liang-Barsky三维裁剪算法裁剪从$P_1(0,\ 0,\ -10)$到$P_2(0,0,10)$的直线段，近平面和远平面分别为 $n = 1$ 和 $f = 2$。

Liang-Barsky算法中所涉及的各参数的值为:

$$
\begin{aligned}
d_1 &= -(\Delta x + \Delta z) = -20 & q_1 &= x_1 + z_1 = -10\\
d_2 &= \ \ \Delta x - \Delta z = -20 & q_2 &= z_1 - x_1 = -10\\
d_3 &= -(\Delta y + \Delta z) = -20 & q_3 &= y_1 + z_1 = -10\\
d_4 &= \ \ \Delta y - \Delta z = -20 & q_4 &= z_1 - y_1 = -10\\
d_5 &= -\Delta z = -20 & q_5 &= z_1 - n = -11\\
d_6 &= \ \ \Delta z = 20 & q_6 &= f - z_1 = 12
\end{aligned}
$$

依据Liang–Barsky三维裁剪算法，有

$$
\begin{aligned}
&\text{clipt}(d_1 = -20,\ q_1 = -10,\ t_L = 0,\ \ \ t_U = 1) \Rightarrow t_L = 1/2 \quad \text{可见}\\
&\text{clipt}(d_2 = -20,\ q_2 = -10,\ t_L = 1/2,\ t_U = 1) \Rightarrow t_L = 1/2 \quad \text{可见}\\
&\text{clipt}(d_3 = -20,\ q_3 = -10,\ t_L = 1/2,\ t_U = 1) \Rightarrow t_L = 1/2 \quad \text{可见}\\
&\text{clipt}(d_4 = -20,\ q_4 = -10,\ t_L = 1/2,\ t_U = 1) \Rightarrow t_L = 1/2 \quad \text{可见}\\
&\text{clipt}(d_5 = -20,\ q_5 = -\ 9,\ t_L = 1/2,\ t_U = 1) \Rightarrow t_L = 11/20 \quad \text{可见}\\
&\text{clipt}(d_6 = \ \ 20,\ q_6 = -12,\ t_L = 11/20,\ t_U = 1) \Rightarrow t_L = 12/20 \quad \text{可见}
\end{aligned}
$$

直线段可见部分的参数范围为: $t_L = 11/20 \leqslant t \leqslant 12/20 = t_U$。$P_1$和$P_2$分别为

$$
\begin{aligned}
x_1 &= x_1 + t_L \Delta x = x_1 = 0\\
y_1 &= y_1 + t_L \Delta y = y_1 = 0\\
z_1 &= z_1 + t_L \Delta z = -10 + 11/20\,(20) = 1
\end{aligned}
$$

$$x_2 = x_1 + t_U \Delta x = x_1 = 0$$
$$y_2 = y_1 + t_U \Delta y = y_1 = 0$$
$$z_2 = z_1 + t_U \Delta z = -10 + {}^{12}/_{20}(20) = 2$$

P_1P_2的可见部分为$1 \leqslant z \leqslant 2$，如所预计的那样，位于近平面和远平面之间。

3.15 齐次坐标裁剪

如果在齐次坐标系中进行裁剪(见Rogers[Roge90a])，而且涉及透视变换，则需要谨慎处理。根本原因是因为此时直线不一定被单个裁剪平面分割成区域内和区域外两部分。它有可能在无穷远处“卷回”，使得两段可能同时在区域内可见。Blinn方法[Blin78a]先对所有的线段进行裁剪，然后再除以齐次坐标进行透视变换，以防线段从无穷远处卷回。

3.15.1 Cyrus-Beck算法

当裁剪线段完全位于视点或投影中心(见例3.21)的前面时，Liang-Barsky和Cyrus-Beck算法可在透视变换空间中正确地实现视域四棱锥对线段的裁剪。但是当线段延伸到投影中心之后时，算法可能将部分可见线段误判为不可见线段而抛弃。要获得正确的结果，需先将线段对定义在正常坐标空间中的实际裁剪体进行裁剪，然后再对这些结果作透视变换。注意在作透视变换前，可对线段和裁剪体施加任意的仿射变换(例如旋转、平移等)。下面举例说明。

例3.24 当线段延伸到投影中心之后时的Cyrus-Beck裁剪。

考虑直线段$P_1(0, 1, 6)P_2(0, -1, -6)$，在裁剪体$(x_L, x_R, y_B, y_T, z_H, z_Y) = (-1, 1, -1, 1-1, 1)$内的裁剪。投影中心位于$z_{CP} = 5$处。直线段$P_1P_2$穿过裁剪体，其始点$P_1$位于投影中心的后面。

经过透视变换 (见 Rogers[Roge90a])，线段端点的齐次坐标为

$$\begin{matrix} P_1 \\ P_2 \end{matrix}\begin{bmatrix} 0 & 1 & 6 & 1 \\ 0 & -1 & -6 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\frac{1}{5} \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 0 & 1 & 6 & -\frac{1}{5} \\ 0 & -1 & -6 & \frac{11}{5} \end{bmatrix}$$

注意到端点P_1P_2变换后的齐次坐标符号相反，这说明直线经由无穷处卷回。即当投影到平面$h=1$时，直线从P_1离开P_2向无穷远处伸展然后再卷回P_2。除以齐次坐标分量得到通常的坐标:

$$P_1(0,-5,-30) \quad 和 \quad P_2\left(0, -\frac{5}{11}, -\frac{30}{11}\right)$$

并发现位于裁剪体之前、投影中心之后的点P_1经历无穷远点卷回到裁剪体的后面。由于线段的两个端点均位于裁剪体之后，故Cyrus-Beck算法把它当作不可见线段予以抛弃。

现在我们先取实际的立方裁剪体对线段作裁剪，然后再作透视变换。仍取例3.20中每个裁剪平面上的内法矢量和点，直线首先被立方体(−1, 1, −1, 1, −1, 1)裁剪。此时P_1P_2的方向为

$$\mathbf{D} = \mathbf{P}_2 - \mathbf{P}_1 = [0\ \ -1\ \ -6] - [0\ \ 1\ \ 6] = [0\ \ -2\ \ -12]$$

结果见表3-15。

由表3-15可见，可见部分的线段为${}^{5}/_{12} < t < {}^{7}/_{12}$，裁剪所得的线段的端点为

$$P\left(\frac{5}{12}\right)=[0\quad 1\quad 6]+[0\quad -2\quad -12]\left(\frac{5}{12}\right)=\left[0\quad \frac{1}{6}\quad 1\right]$$

$$P\left(\frac{7}{12}\right)=[0\quad 1\quad 6]+[0\quad -2\quad -12]\left(\frac{7}{12}\right)=\left[0\quad -\frac{1}{6}\quad -1\right]$$

表 3-15

平面	n	f	w	w·n	D·n⊖	t_L	t_U
顶面	[0 −1 0]	(1, 1, 1)	[−1 0 5]	0	2	0	
底面	[0 1 0]	(−1,−1,−1)	[1 2 7]	2	−2		1
右面	[−1 0 0]	(1, 1, 1)	[−1 0 5]	1	0		
左面	[1 0 0]	(−1,−1,−1)	[1 2 7]	1	0		
前面	[0 0 −1]	(1, 1, 1)	[−1 0 5]	−5	12	5/12	
后面	[0 0 1]	(−1,−1,−1)	[1 2 7]	7	−12		7/12

⊖ 如果D · n < 0, 对应上限值t_U; 如果D · n > 0, 对应下限值t_L。

将它们变换到透视空间，其齐次坐标表示为（见[Roge 900]）

$$\begin{bmatrix}0 & \frac{1}{6} & 1 & 1\\ 0 & -\frac{1}{6} & -1 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & -\frac{1}{5}\\ 0 & 0 & 0 & 1\end{bmatrix}=\begin{bmatrix}0 & \frac{5}{24} & \frac{5}{4} & 1\\ 0 & -\frac{5}{36} & -\frac{5}{6} & 1\end{bmatrix}$$

此即线段P_1P_2上的可见部分，这才是正确结果。　□

3.15.2 Liang-Barsky算法

Liang-Barsky[Lian84]通过对裁剪体进行变形来将线裁剪算法扩展到齐次坐标空间。裁剪在透视变换之后的透视空间进行，但此时尚未投影到物理空间，即尚未除以齐次坐标。在3.14节讨论的Liang-Barsky三维裁剪算法中，透视投影的中心即视点位置取为原点，因而在坐标系中，场景中所有点的$z_e>0$，即场景位于正半侧。故当直线段的起点位于透视投影中心即视点的后面时，Liang-Barsky齐次裁剪算法会出错。

对原视域棱锥体进行变换，使之成为一个张角为90°的直棱锥体，加上远、近两个裁剪平面后，组成视域四棱锥。其裁剪区域为

$$-z<x<z\qquad -z<y<z\qquad n<z<f$$

其中n和f表示近平面和远平面的位置。

透视变换后（见[Roge90a]），原平面上的z坐标变成：

$$z^*=\frac{z}{h}$$

其中h是齐次权因子坐标。再经过下面的变换使 z^* 归一化，即$0<z^*<1$

$$z^*=\frac{f}{f-n}\left(1-\frac{n}{z}\right)$$

代入裁剪区域不等式，得

$$-h < x < h \qquad -h < y < h \qquad 0 < z < h$$

又因为

$$x = x_1 + \Delta x t \qquad y = y_1 + \Delta y t \qquad z = z_1 + \Delta z t$$

以及

$$h = h_1 + \Delta h t$$

其中，$\Delta h = h_2 - h_1$，代入后得

$$\begin{aligned} -(\Delta x + \Delta h)t &< x_1 + h_1 \quad \text{和} \quad (\Delta x - \Delta h)t < h_1 - x_1 \\ -(\Delta y + \Delta h)t &< y_1 + h_1 \quad \text{和} \quad (\Delta y - \Delta h)t < h_1 - y_1 \\ -\Delta z\, t &< z_1 \quad \text{和} \quad (\Delta z - \Delta h)t < h_1 - z_1 \end{aligned}$$

因此

$d_1 = -(\Delta x + \Delta h)$	$q_1 = x_1 + h_1$	左面
$d_2 = \Delta x - \Delta h$	$q_2 = h_1 - x_1$	右面
$d_3 = -(\Delta y + \Delta h)$	$q_3 = y_1 + h_1$	底面
$d_4 = \Delta y - \Delta h$	$q_4 = h_1 - y_1$	顶面
$d_5 = -\Delta z$	$q_5 = z_1$	近平面
$d_6 = \Delta z - \Delta h$	$q_6 = h_1 - z_1$	远平面

下面是Liang-Barsky齐次坐标裁剪算法的伪代码程序，其中Clipt函数与3.6节中二维Liang-Barskey裁剪算法一样。

Liang–Barsky three-dimensional homogeneous clipping algorithm

P_1 *and* P_2 *are the end points with components*
$x_1, y_1, z_1, h_1, x_2, y_2, z_2, h_2$
t_L *and* t_U *are the lower and upper parametric limits*
x_L, x_R, y_B, y_T *are the left, right, bottom and top window edges*

the **clipt** *function is the same as in the two-dimensional Liang–Barsky algorithm (see Sec. 3–6)*

```
t_L = 0
t_U = 1
deltax = x_2 − x_1
deltah = h_2 − h_1
if clipt(−deltax − deltah, x_1 + h_1, t_L, t_U) = true then
    if clipt(deltax − deltah, h_1 − x_1, t_L, t_U) = true then
        deltay = y_2 − y_1
        if clipt(−deltay − deltah, y_1 + h_1, t_L, t_U) = true then
            if clipt(deltay − deltah, h_1 − y_1, t_L, t_U) = true then
                deltaz = z_2 − z_1
                if clipt(−deltaz, z_1, t_L, t_U) = true then
                    if clipt(deltaz−deltah, h_1−z_1, t_L, t_U) = true then
                        if t_U < 1 then
                            x_2 = x_1 + t_U * deltax
                            y_2 = y_1 + t_U * deltay
```

```
                    z2 = z1 + tU * deltaz
                    h2 = h1 + tU * deltah
                end if
                if tL > 0 then
                    x1 = x1 + tL * deltax
                    y1 = y1 + tL * deltay
                    z1 = z1 + tL * deltaz
                    h1 = h1 + tL * deltah
                end if
                Draw P1P2
            end if
        end if
    end if
  end if
end if
finish
```

在下一个例子中，我们考虑一条起点位于投影中心后面的线段来观察Liang-Barsky齐次裁剪算法会出现的问题。如果线段的起点不位于投影中心的后面，那么Liang-Barsky算法能够正确地进行裁剪。

例3.25 Liang-Barsky齐次坐标裁剪。

Liang-Barsky齐次坐标裁剪体是一个视点位于原点的90°视域四棱锥（见图3-26）。假设近平面和远平面分别位于 $z = n = 1$ 和 $z = f = 2$。考虑齐次坐标系中的一条从P_1(0 0 −10 −9)到P_2(0 0 10 11)的直线段。其起点位于投影中心即视点的后面，直线段沿z轴方向穿过裁剪体到达裁剪体后的另一端点(注意 $h_1 = -9 < 0$ 和 $h_2 = 11 > 0$)。

参照Liang-Barsky齐次裁剪算法，我们有

$$
\begin{aligned}
d_1 &= -(\Delta x + \Delta h) = -20 & q_1 &= x_1 + h_1 = -9\\
d_2 &= \Delta x - \Delta h = -20 & q_2 &= h_1 - x_1 = -9\\
d_3 &= -(\Delta y + \Delta h) = -20 & q_3 &= y_1 + h_1 = -9\\
d_4 &= \Delta y - \Delta h = -20 & q_4 &= h_1 - y_1 = -9\\
d_5 &= -\Delta z = -20 & q_5 &= z_1 = -10\\
d_6 &= \Delta z - \Delta h = 0 & q_6 &= h_1 - z_1 = 1
\end{aligned}
$$

$\text{clipt}(d_1 = -20, q_1 = -9, t_L = 0, t_U = 1) \Rightarrow t_L = 9/20$　可见

$\text{clipt}(d_2 = -20, q_2 = -9, t_L = 9/20, t_U = 1) \Rightarrow t = 9/20$　可见

$\text{clipt}(d_3 = -20, q_3 = -9, t_L = 9/20, t_U = 1) \Rightarrow t = 9/20$　可见

$\text{clipt}(d_4 = -20, q_4 = -9, t_L = 9/20, t_U = 1) \Rightarrow t = 9/20$　可见

$\text{clipt}(d_5 = -20, q_5 = -10, t_L = 9/20, t_U = 1) \Rightarrow t_L = 1/2$　可见

$\text{clipt}(d_6 = 0, q_6 = 1, t_L = 1/2, t_U = 1) \Rightarrow$　可见

可见部分的参数范围为：$t_L = 1/2 \leqslant t \leqslant 1 = t_U$。因为 $t_U = 1$，所以P_2不变。计算P_1：

$$
\begin{aligned}
x_1 &= x_1 + t_L \Delta x = x_1 = 0\\
y_1 &= y_1 + t_L \Delta y = y_1 = 0
\end{aligned}
$$

$$z_1 = z_1 + t_L \Delta z = -10 + \frac{1}{2}(20) = 0$$

$$h_1 = h_1 + t_L \Delta z = -9 + \frac{1}{2}(20) = 1$$

在齐次坐标中$P_1 = [0\ 0\ 0\ 1]$。这一结果表明，如同Cyrus-Beck 算法一样，Liang-Barsky算法对于起点位于视点之后、终点位于视点之前的线段的裁剪结果也是错的。要得到正确的结果，可以像例3.24那样首先在透视变换前的裁剪体内做裁剪。具体细节留作练习(见习题3.15)。 □

3.16 内法矢量和三维凸集合的确定

如前所述，在二维情形下,可以采用平移和旋转方法判别凸多边形并计算各边的内法矢量。这一方法也可推广到三维情形。在三维情形下,相应算法为:

对于裁剪体的每一个多边形表面:

平移裁剪体，使得多边形表面的一个顶点重合于坐标原点。

绕原点旋转多边形，使得与该顶点连接的一条边重合于一个坐标轴，如x轴。

绕此坐标轴旋转，使得多边形表面与一坐标平面（如 $z = 0$ 平面)重合。

检查裁剪体其余顶点垂直于该平面的坐标分量(如z分量)的符号。

若所有顶点的该分量都同号或为零，则相对此平面裁剪体是凸的。若相对所有表面，裁剪体都是凸的，则裁剪体为凸多面体。否则为凹多面体。

若所有顶点的该分量都为零，则裁剪体退化为一个平面。

凸平面的内法矢量在旋转后的坐标系中与垂直于该表面的坐标轴同向，矢量符号取其余顶点的该坐标分量的符号。

作相反旋转后，即可得原内法矢量。

例3.26 裁剪体凸性的判断。

作为具体例子，仍考虑前面在例3.22中削去一角的立方体，如图3-27a所示。现用上述算法，判断裁剪体对于表面abc的凸性。先平移裁剪体，使得a点重合于坐标原点。相应的4×4齐次坐标变换矩阵(见[Roge90a])为

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & -1 & 1 \end{bmatrix}$$

平移后投影到 $z = 0$ 平面上的视图，如图3-27b所示。

再绕z轴旋转，旋转角度 $\theta = -45°$，使得ab重合于x轴。相应的齐次坐标变换矩阵为(见[Roge90a])

$$[R_z] = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

旋转后投影到$z = 0$平面上的视图如图3-27c所示。剩下的工作是绕轴旋转一个适当角度，使得平面abc重合于坐标平面$y = 0$。在图3-27c中，点c坐标为(0.565685, 0.565685, −0.8)，故绕x轴的旋转角度为

$$\alpha = \tan^{-1}\left(\frac{y}{z}\right) = \tan^{-1}\left(\frac{0.565685}{-0.8}\right) = -35.2644°$$

绕x轴旋转α角度后，将裁剪体置于坐标平面$y=0$的下方。而绕x轴旋转$(180-\alpha)^\circ$角度后，则将裁剪体置于坐标平面$y=0$的上方。这里取后者，最后投影到$z=0$平面上的视图见图3-27d，相应的旋转矩阵为

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

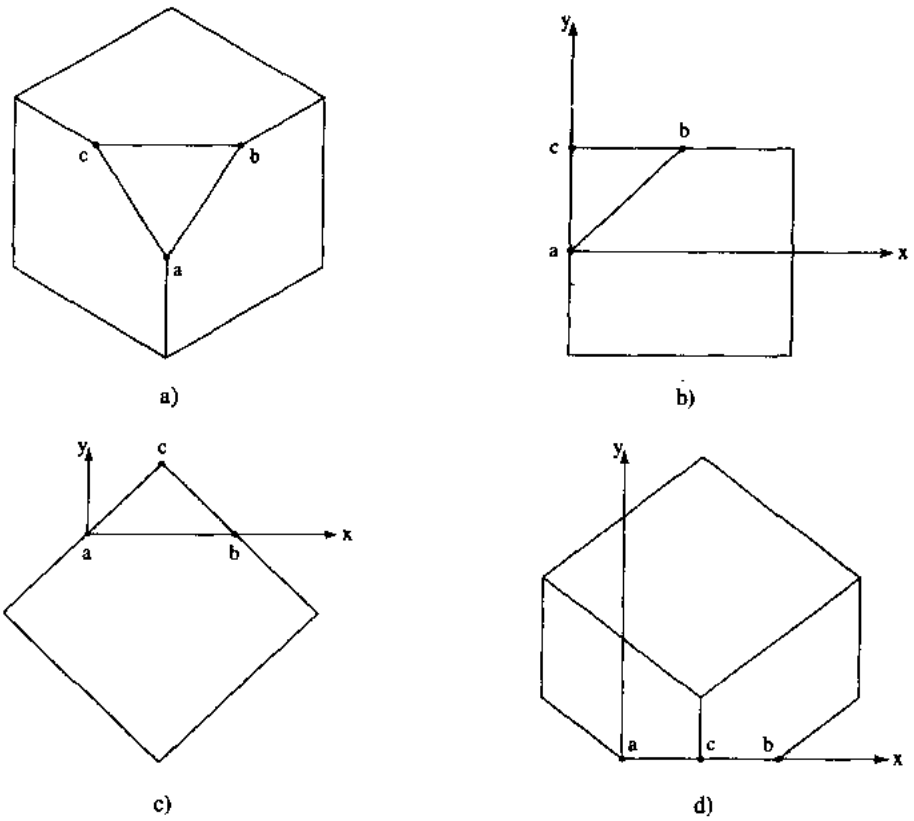


图3-27　确定体的凸性和内法矢量

在新坐标系中，裁剪体所有其他顶点的y坐标均为正。故对于平面abc，裁剪体为凸。平面abc的内法矢量为

$$\mathbf{n}' = [0 \quad \text{sign}(y) \quad 0] = [0 \quad 1 \quad 0]$$

做相反旋转得

$$\mathbf{n} = [0.5774 \quad -0.5774 \quad -0.5774]$$

或

$$\mathbf{n} = [1 \quad -1 \quad -1]$$

即为所求。注意，要判定裁剪体为凸多面体，需逐个表面检查裁剪体的凸性。　□

3.17　凹体分割

类似于3.8节对凹多边形的裁剪，对凹体的裁剪可以通过对组成凹体的凸体作适当的内裁

剪和外裁剪子以实现。对上节所述的旋转、平移算法做适当扩展，可将简单凹体分割成凸体的组合。对多面体，分割过程可以表达如下。

对每一个多面体表面:

平移，使得多边形表面的一个顶点重合于坐标原点。

绕原点旋转，使得该顶点的一个邻边重合于坐标轴，如x轴。

绕此坐标轴旋转，使得多边形表面与一坐标平面（如z=0平面)重合。

检查裁剪体其余顶点垂直于该平面的坐标分量(如z分量)的符号。

若所有顶点的该分量都同号或为零，则对于此平面来说，裁剪体是凸的。否则，裁剪体为为凹多面体，沿着该多边形表面分割裁剪体。

对每一个分割出来的体重复调用该过程，直到每一个体都为凸体。

现举例说明该算法。

例3.27 凹多面体的分割。

考虑如图3-28a所示的凹多面体，每一个表面多边形为:

背面: $P_1(3,0,0), P_2(0,0,0), P_3(0,2,0), P_4(1,2,0)$
$P_5(1,3/2,0), P_6(3/2,3/2,0), P_7(3/2,2,0), P_8(3,2,0)$

前面: $P_9(3,0,2), P_{10}(0,0,2), P_{11}(0,2,2), P_{12}(1,2,2)$
$P_{13}(1,3/2,2), P_{14}(3/2,3/2,2), P_{15}(3/2,2,2), P_{16}(3,2,2)$

左面: $P_2(0,0,0), P_{10}(0,0,2), P_{11}(0,2,2), P_3(0,2,0)$

右面: $P_1(3,0,0), P_8(3,2,0), P_{16}(3,2,2), P_9(3,0,2)$

底面: $P_1(3,0,0), P_2(0,0,0), P_{10}(0,0,2), P_9(3,0,2)$

顶左面: $P_{10}(0,0,2), P_4(1,2,0), P_3(0,2,0), P_{11}(0,2,2)$

槽左面: $P_{13}(1,3/2,2), P_5(1,3/2,0), P_4(1,2,0), P_{12}(1,2,2)$

槽底面: $P_{13}(1,3/2,2), P_{14}(3/2,3/2,2), P_6(3/2,3/2,0), P_5(1,3/2,0)$

槽右面: $P_6(3/2,3/2,0), P_7(3/2,2,0), P_{15}(3/2,2,2), P_{14}(3/2,3/2,2)$

顶右面: $P_{16}(3,2,2), P_8(3,2,0), P_7(3/2,2,0), P_{15}(3/2,2,2)$

采用上述算法，取图3-28a中标识为abc的槽左面检查多面体的凹凸性。首先平移多面体使得P_5(即图3-28a中点a)重合于坐标原点，这同时使得P_{13}(即图3-28a中点b)落在z轴的正半轴上。在x,y,z方向的平移量分别为$-1, 3/2, 0$。结果投影到平面 $z=0$ 上，见图3-28b。绕z轴旋转$-90°$，使得平面abc重合于坐标平面y=0。结果如图3-28c所示。投影到平面z=0上的视图见图3-28d。

检查y坐标分量，可知多面体为凹体。沿平面 $y=0$ 将凹体分割成V_1、V_2两个多面体。V_1位于$y=0$平面的上方，V_2位于下方。在原坐标系中，V_1，V_2的各表面如下。

V_1:

左面: $P_2(0,0,0), P_{10}(0,0,2), P_{11}(0,2,2), P_3(0,2,0)$

右下: $P'_{10}(1,0,2), P'_5(1,0,0), P_5(1,3/2,0), P_{13}(1,3/2,2)$

右上: $P_{13}(1,3/2,2), P_5(1,3/2,0), P_4(1,2,0), P_{12}(1,2,2)$

顶面: $P_{10}(0,0,2), P_4(1,2,0), P_3(0,2,0), P_{11}(0,2,2)$

底面: $P_2(0,0,0), P'_5(1,0,0), P'_{10}(1,0,2), P_{10}(0,0,2)$

前面: $P_{10}(0,0,2), P'_{10}(1,0,2), P_{13}(1,3/2,2), P_{12}(1,2,2), P_{11}(0,2,2)$

后面: $P'_5(1,0,0), P_2(0,0,0), P_3(0,2,0), P_4(1,2,0), P_5(1,3/2,0)$

V_2:

左面:	$P_5'(1,0,0), P_{10}'(1,0,2), P_{13}(1, 3/2, 2), P_5(1, 3/2, 0)$
右面:	$P_1(3,0,0), P_8(3,2,0), P_{16}(3,2,2), P_9(3,0,2)$
槽右面:	$P_6(3/2, 3/2, 0), P_7(3/2, 2, 0), P_{15}(3/2, 2, 2), P_{14}(3/2, 3/2, 2)$
槽底面:	$P_{13}(1, 3/2, 2), P_{14}(3/2, 3/2, 2), P_6(3/2, 3/2, 0), P_5(1, 3/2, 0)$
顶右面:	$P_{16}(3,2,2), P_8(3,2,0), P_7(3/2, 2, 0), P_{15}(3/2, 2, 2)$
底面:	$P_5'(1,0,0), P_1(3,0,0), P_9(3,0,2), P_{10}'(1,0,2)$

对V_1、V_2再次调用上述算法，得知V_1为凸体，而V_2被分成两个多面体，其后判定这两个多面体均为凸体。原多面体的分解图见图3-28e。 □

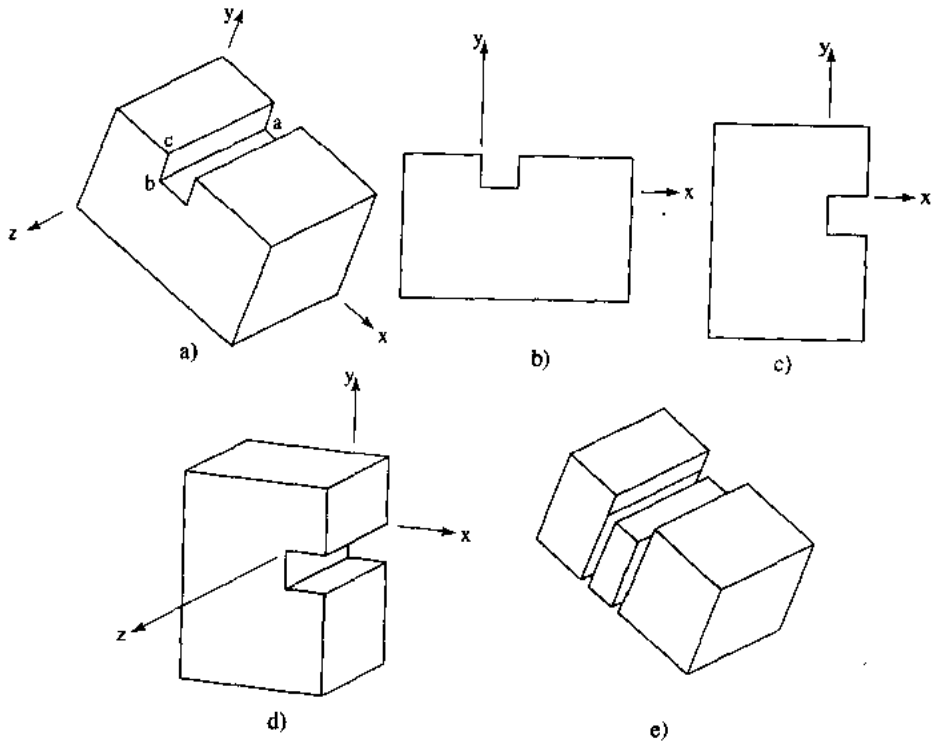


图3-28　凹体分割

3.18 多边形裁剪

前面着重讨论了线段的裁剪。多边形当然可以看作是线段的集合，在画线图形应用中，可将多边形分解为一条一条的线段进行裁剪，这时多边形裁剪显得并不重要。当一封闭的多边形作为线段集合进行裁剪时，原封闭多边形变成一个或多个开的多边形或离散的线段，如图3-29所示。

然而，当多边形作为填充的区域看待时，裁剪后必须保持封闭多边形的封闭性。这要求在图3-29中将bc、ef、fg和ha等线段加入裁剪后的多边形中。加入线段ef和fg是比较困难的，但是更为困难的是如图3-30所示，多边形裁剪结果包含多个互不连接的小多边形。图中线段ab和cd常被看作裁剪后生成的多边形的一部分。例如，若原多边形取为红色，而背景取为蓝色，则ab和cd也取为红色而显示在蓝色的背景上。显然这与实际情况不符。

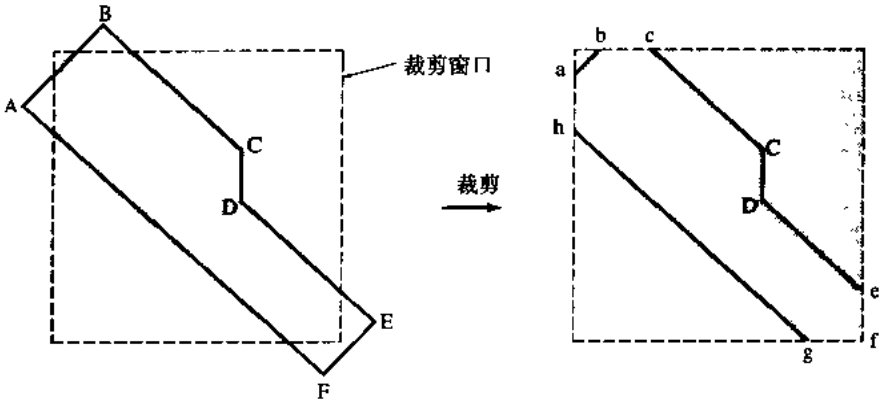


图3-29 多边形裁剪，产生开多边形

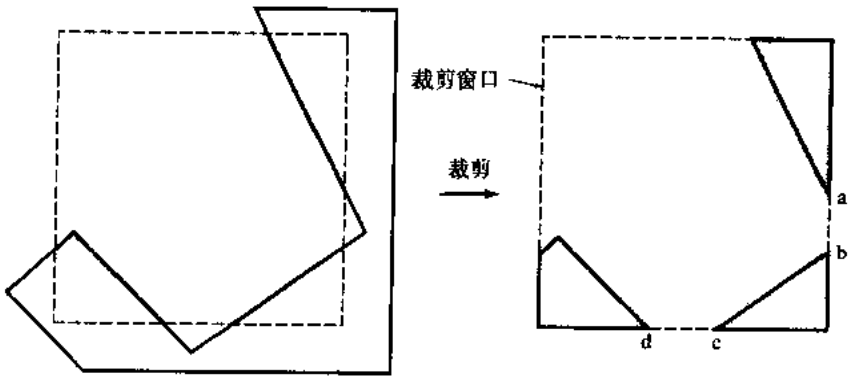


图3-30 多边形裁剪——分离多边形

3.19 逐次多边形裁剪 ——Sutherland-Hodgman算法

Sutherland-Hodgman[suth74b]算法的基本思想是，多边形对一条边或一个面的裁剪容易实现，故用窗口的边，一条一条地对原多边形和中间结果多边形进行裁剪。图3-31显示了一个矩形窗口裁剪多边形的过程。原多边形由一系列顶点$P_1,\cdots,P_n$所定义，它的边为$P_1P_2,P_2P_3,\cdots,P_{n-1}P_n,P_nP_1$。

这些边首先被窗口左边裁剪，生成一个中间多边形，如图3-31所示。然后重新调用裁剪程序，将中间多边形对窗口顶边进行裁剪，生成第二个中间多边形。继续这一过程，直至多边形被窗口的所有边都裁剪过为止。裁剪的每一步均显示在图3-31中。注意在最终结果多边形中，角点Q_8的加入并非难事。这一算法可以在任何凸多边形窗口内对任何凸或凹的、平面或非平面的多边形进行裁剪。窗口各边以什么顺序裁剪多边形是无关紧要的。

算法的输出结果是一个多边形的顶点列表，均位于裁剪面的可见一侧。由于多边形的每一条边都是单独地与裁剪面进行比较，因此只需考虑单条边和单个裁剪面之间的位置关系。取多边形顶点列表中的一点P(第一点除外)作为一边的终点，列表中位于P前面的一点S作为该边的起点，则边SP同裁剪面之间只有四种可能的关系，如图3-32所示。

每次将多边形的边与裁剪面比较之后，向裁剪后的多边形顶点列表输出一个到两个顶点，或者不输出顶点。若边完全可见，则输出P点。注意不必再输出边的起点S。因为顶点表中的点是依序处理的，S作为前一条边的终点已经被输出。若边完全不可见，则没有输出。

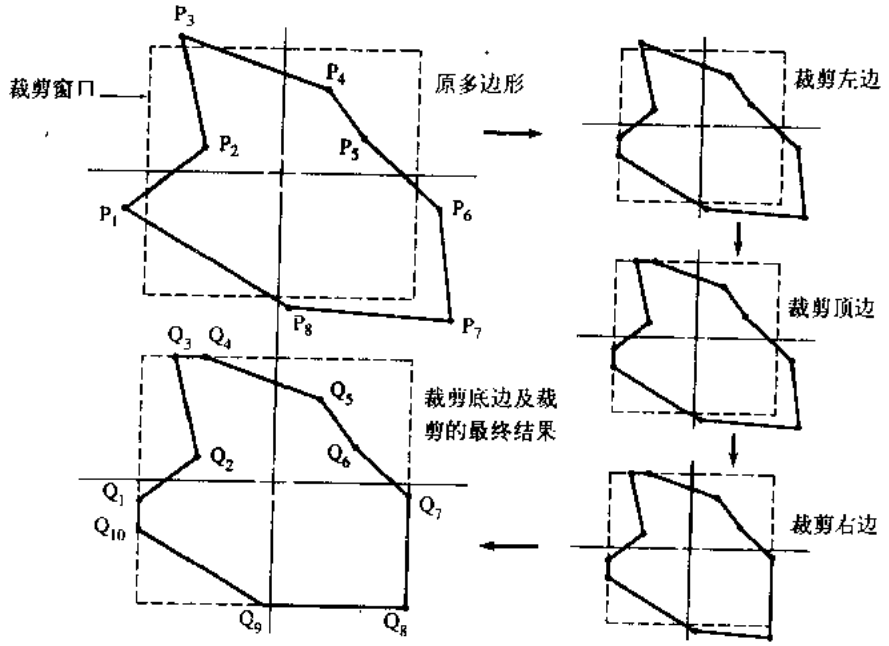


图3-31　逐次多边形裁剪

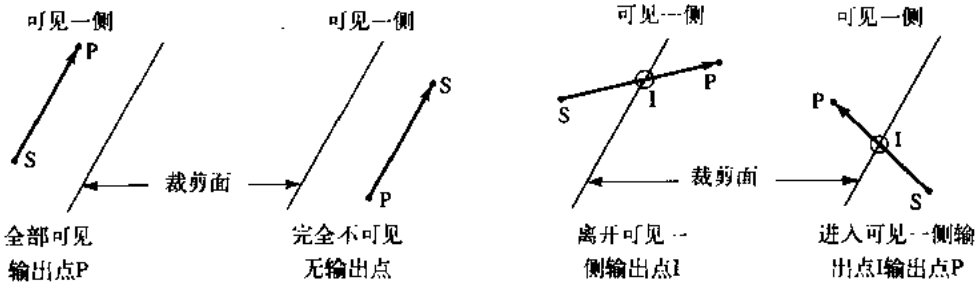


图3-32　边与裁剪面之间的关系

若边部分可见，则边或者进入、或者离开裁剪面的可见侧。如果边离开可见侧，则必须计算并输出多边形边与裁剪面的交点。如果边进入可见侧，则同样需计算并输出多边形与裁剪面的交点。由于此时边的终点P可见，故也应输出。

对于多边形的第一个顶点，只需判断它是否可见。若可见，则输出并作为起点S。若不可见，则不输出，但仍需作为起点S保存。

最后一边P_nP_1需分开处理。做法是将顶点表中的第一个顶点保存为F。这样最后一边变为P_nF，然后可与其他边一样统一处理。

3.19.1　确定一个点的可见性

在给出完整的算法之前，还有两点需加以考虑：即点的可见性判别及多边形的边同裁剪面交点的计算。点的可见性判别等同于判别点位于裁剪面的哪一侧。假定裁剪多边形的边按顺时针方向排列，多边形内部将总是位于边的右侧；如果按逆时针方向，那么多边形的内部将总是位于边的左侧。前面分析过两种方法，用来判定一个点相对于一条直线或一个平面的位置关系（可见性）：其一是检查从直线或面上的一点到被判别点的矢量与直线或平面的法矢

量的点积的符号，见3.5节；其二是将点的坐标代入线或面的方程，见3.9节。后一个是Sutherland-Hodgman 提出的方法的一种变体（见[Suth74b]）。

另外一种方法是检查位于同一个平面上两矢量叉积的z分量的符号。在裁剪平面上取两点P_1和P_2，设所考虑的点为P_3，那么这三个点确定一个平面，矢量$\mathbf{P_1P_2}$和$\mathbf{P_1P_3}$均位于该平面上。假定这一平面为xy坐标平面，则矢量叉积$\mathbf{P_1P_3}\otimes\mathbf{P_1P_2}$只包含$z$分量：

$$(x_3-x_1)(y_2-y_1) - (y_3-y_1)(x_2-x_1)$$

当z分量的符号为正、零或负时，P_3分别位于直线$\mathbf{P_1P_2}$的右侧、上面或左侧。

当裁剪窗口为一矩形且平行于坐标轴时，上述方法均十分简单。

例3.28 点对面的关系。

考虑一垂直于x轴的裁剪面 $x = w = -1$，如图3-33所示。确定点$P_3(-2, 1)$和$P_3'(2, 1)$相对于裁剪面的位置关系。

采用上述矢量叉积方法，取$P_1(-1, 0)$和$P_2(-1, 2)$。对于P_3：

$$\begin{aligned}&(x_3-x_1)(y_2-y_1) - (y_3-y_1)(x_2-x_1)\\ &= [(-2)-(-1)] \times (2-0)-(1-0) \times [(-1)-(-1)]\\ &= (-1) \times 2-1 \times 0 = -2 < 0\end{aligned}$$

表明P_3位于$\mathbf{P_1P_2}$的左侧。对于P_3'：

$$\begin{aligned}&(x_3'-x)(y_2-y_1) - (y_3'-y_1)(x_2-x_1)\\ &=3 \times 2-1 \times 0 = 6 > 0\end{aligned}$$

表明P_3'位于$\mathbf{P_1P_2}$的右侧。

代入法更为简单，此处判别函数为$x-w$。对于P_3、P_3'分别有

$$x_3 - w = -2 - (-1) = -1 < 0$$

$$x_3' - w = 2 - (-1) = 3 > 0$$

表明P_3和P_3'分别位于$\mathbf{P_1P_2}$的左侧和右侧。

裁剪面的内法矢量取$\mathbf{n} = [1\ 0]$，裁剪面上的一点取$f(-1, 0)$，计算矢量的点积。对于P_3有

$$\mathbf{n} \cdot [\mathbf{P_3}-\mathbf{f}] = [1\ 0] \cdot [-1\ 1] = -1<0$$

对于P_3'有

$$\mathbf{n} \cdot [\mathbf{P_3'}-\mathbf{f}] = [1\ 0] \cdot [3\ 1] = 3 > 0$$

再次表明P_3位于裁剪面的左侧，P_3'位于裁剪面的右侧。

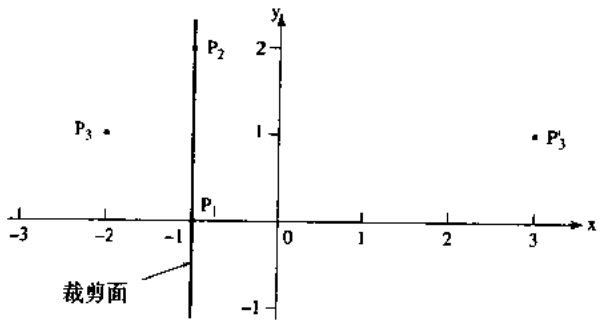


图3-33 可见性判别

通过这些可见性，若一条多边形边的两端点均可见或均不可见，则该边是完全可见或完

全不可见。若一端点可见而另一端点不可见，则多边形边与裁剪面相交，这时需求出交点。

3.19.2 线段求交

前面讨论过的任一线段求交(裁剪)方法均可采用，如Cyrus-Beck方法(见3.5节)、参数方法(见3.4节)或中点分割方法(见3.3节)等。同样如前所述，对于平行于坐标轴的矩形裁剪窗口，这些方法都十分简单。Cyrus-Beck方法和中点分割方法是完全通用的，但是二维平面中任意两条参数形式的线段求交方法尚需作进一步讨论。

设有端点分别为P_1、P_2和P_3、P_4的两条直线段。它们的参数表示分别为

$$P(s) = P_1 + (P_2 - P_1)s \qquad 0 < s < 1$$

$$P(t) = P_3 + (P_4 - P_3)t \qquad 0 < t < 1$$

在交点处$P(s)$=$P(t)$。注意$P(s)$、$P(t)$均为矢量值函数，即 $P(s) = [x(s)\ y(s)]$和$P(t) = [x(t)\ y(t)]$，从而得到交点应满足两个关于未知参数值s、t的方程，即$x(s) = x(t)$，$y(s) = y(t)$。若该方程组无解，则两直线相互平行；若 s 或 t 超出要求的范围，则两直线段不相交。用矩阵形式求解会特别方便。

例3.29 参数直线段求交。

如图3-34所示，考虑P_1[0 0]到P_2[3 2]、P_3[3 0]到P_4[0 2]的两条直线段，有

$$P(s) = [0\ \ 0] + [3\ \ 2]s$$

$$P(t) = [3\ \ 0] + [-3\ \ 2]t$$

令x分量和y分量分别相等，得

$$3s = 3 - 3t$$

$$2s = 2t$$

解得

$$s = t = \frac{1}{2}$$

$$P_i(s) = [0\ \ 0] + \frac{1}{2}[3\ \ 2]$$

所以，交点为

$$= \left[\frac{3}{2}\ \ 1\right]$$

□

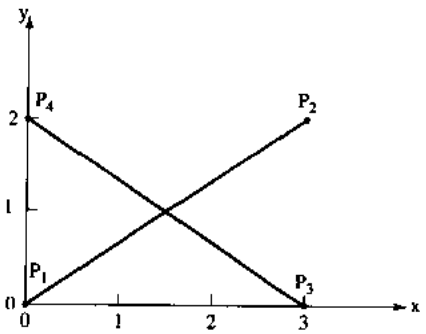


图3-34 参数直线段求交

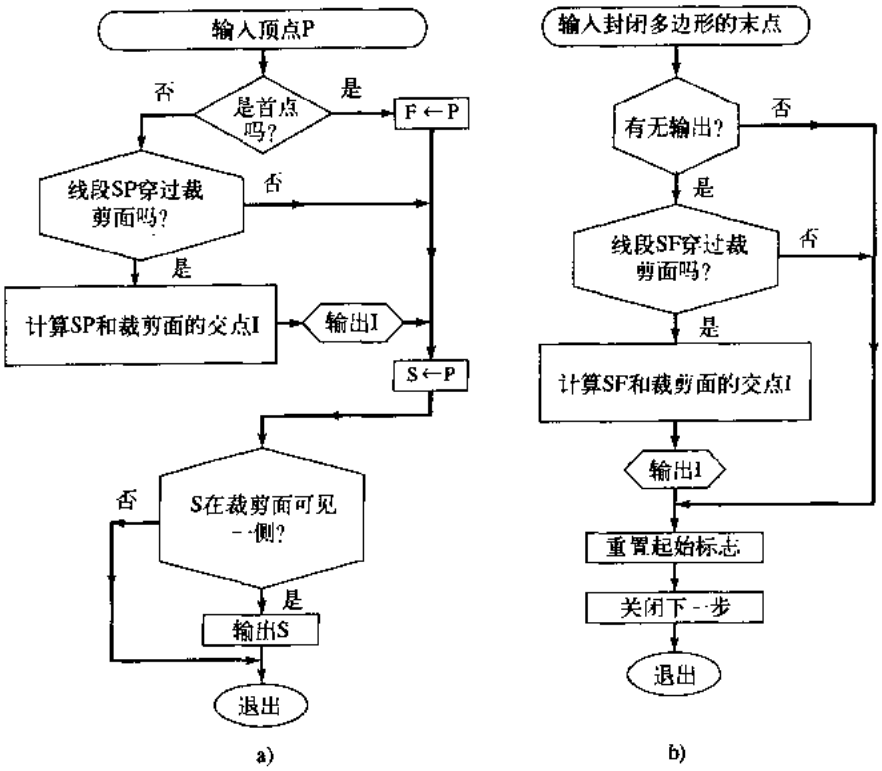


图3-35 Sutherland-Hodgman 逐次多边形裁剪框图

3.19.3 算法

在上述Sutherland-Hodgman算法中，裁剪多边形的每一边是依次处理的，故同一子程序稍加改动即可适用于多边形所有的边。注意最后一个顶点需作特殊处理。图3-35所示为该算法的流程图，摘自参考文献[Suth74b]。其中图3-35a所示过程适用于每一顶点，而图3-35b只适用于最后一个顶点。Sutherland 和Hodgman避免生成和存储中间多边形顶点的方法。具体地讲，取多边形每一条边(顶点)对窗口的所有边界逐次进行裁剪，而不是同时取多边形的所有边（顶点）对窗口的一条边界进行裁剪。当一条多边形边(顶点)被一窗口边界裁剪后，算法将递归地用下一窗口边界裁剪上次所得的结果。这使得该算法更适宜于在硬件上实现。

下面给出该算法产生并保存中间多边形的一个伪代码程序。

Sutherland–Hodgman polygon clipping algorithm

This is a sequential implementation

P *is the input polygon array*
Q *is the output polygon array*
W *is the clipping window array; the first vertex is repeated as the last vertex*
Nin *is the number of input polygon vertices*
Nout *is the number of output polygon vertices*
Nw *is the number of clipping polygon vertices plus one*
all polygon vertices are given in counterclockwise order

for each window edge
for i = 1 to Nw − 1

```
set the output counter and zero the output array
Nout = 0
Q = 0
clip each polygon edge against this window edge
for j = 1 to Nin - 1
    treat the first point specially
    if j = 1 then
        save first point
        F = Pj
        set S to the first point
        S = Pj
        check the visibility of the first point
        call Visible(S, Wi, Wi+1; Svisible)
        if Svisible ≥ 0 then
            the point is visible; output it
            call Output(S, Nout; Q)
        end if
    end if
    check if this polygon edge crosses the window edge
    call Cross(S, Pj+1, Wi, Wi+1; Spcross)
    if Spcross = no then
        replace the first point
        S = Pj+1
    else
        the polygon edge crosses the window edge
        calculate the intersection point
        call Intersect(S, Pj, Wi, Wi+1; Pintersect)
        output the intersection point
        call Output (Pintersect,Nout; Q)
        S = Pj+1
    end if
     check if the second point on the edge (now S) is visible
     call Visible(S, Wi, Wi+1; Svisible)
     if Svisible ≥ 0 then
         the point is visible; output it
         call Output(S, Nout; Q)
     end if
next j
closure—treat the edge PnP1
if there was no output, skip to the next window edge
if Nout = 0 then
    P = Q
    Nin = Nout
else
```

```
            check if the last polygon edge crosses the window edge
            call Cross(S, F, W_i, W_i+1; Spcross)
            if Spcross = no then
                P = Q
                Nin = Nout
            else
                the polygon edge crosses the window edge
                calculate the intersection
                call Intersect(S, F, W_i, W_i+1; Pintersect)
                output the intersection
                call Output(Pintersect,Nout; Q)
                Nin = Nout
            end if
        end if
    the polygon is now clipped against the edge W_i to W_i+1
    the algorithm is now reentered with the clipped polygon
    next i
finish
subroutine module to determine if the polygon edge and
  the window edge intersect
subroutine Cross(Start,Point,W1,W2;Spcross)
    determine the visibility of the starting point of the polygon edge
    call Visible(Start,W1,W2;Pvisible)
    Pvisible1 = Pvisible
    determine the visibility of the end point of the polygon edge
    call Visible(Point,W1,W2;Pvisible)
    Pvisible2 = Pvisible
      a polygon edge which begins or ends on a window edge is considered
        not to cross the edge. This point will have previously been output
      if Pvisible1 < 0 and  Pvisible2 > 0 or
          Pvisible1 > 0 and Pvisible2 < 0 then
          Spcross = yes
      else
          Spcross = no
      end if
  return
  subroutine module to determine visibility
  subroutine Visible(Point,P1,P2; Pvisible)
  determine the visibility of Point with respect to the edge P_1P_2
Pvisible < 0 Point is to the left (visible)
         = 0 Point is on the edge P_1P_2
         > 0 Point is to the right (invisible)
  the routine uses the cross-product technique
  the Sign function returns -1, 0, 1 as the argument is
    negative, zero or positive
```

```
    the direction of the edge is assumed to be from P1 to P2
    the window is assumed specified counterclockwise, so that the inside
      (visible side) is always to the left looking from P1 to P2
        Temp1 = (Pointx − P1x) * (P2y − P1y)
        Temp2 = (Pointy − P1y) * (P2x − P1x)
        Temp3 = Temp1 − Temp2
        Pvisible = −Sign(Temp3)
    return
    subroutine module to calculate intersection of two lines
    subroutine Intersect(P1,P2,W1,W2; Pintersect)
    the routine uses a parametric line formulation
    the lines P1P2 and W1W2 are assumed two-dimensional
    the matrix for the parameter values is obtained by equating the x and y
      components of the two parametric lines
    Coeff is a 2 × 2 matrix containing the parameter coefficients
    Parameter is a 2 × 1 matrix containing the parameters
    Right is a 2 × 1 matrix for the right-hand sides of the equations
    Invert is the matrix inversion function
    Parameter(1, 1) is the polygon edge intersection value
    Multiply is the matrix multiply function
        fill the coefficient matrix
        Coeff(1, 1) = P2x − P1x
        Coeff(1, 2) = W1x − W2x
        Coeff(2, 1) = P2y − P1y
        Coeff(2, 2) = W1y − W2y
        fill the right-hand side matrix
        Right(1, 1) = W1x − P1x
        Right(2, 1) = W1y − P1y
        invert the coefficient matrix
        it is not necessary to check for a singular matrix because
          intersection is ensured
        Coeff = Invert(Coeff)
        solve for the parameter matrix
        Parameter = (Coeff) Multiply (Right)
        calculate the intersection points
        Pintersect = P1 + (P2 − P1) * Parameter(1, 1)
    return
    subroutine module for polygon output
    subroutine Output(Vertex; Nout,Q)
    Vertex contains the output point
        increment the number of output vertices and add to Q
        Nout = Nout + 1
        Q(Nout) = Vertex
    return
```

例3.30将进一步说明Sutherland-Hodgman算法的实现过程。此外它还揭示出算法需处理的一个问题，即退化的边界。退化边界的一个例子是裁剪结果多边形的一部分与窗口边界重合，退化边界往复两次，将裁剪结果多边形中两个彼此分离的部分连接在一起，如图3-36所示。在许多应用中，如实区域扫描转换，这些退化边界的存在毫无影响。但对子另一些应用，例如执行某些隐藏面消除算法，则需事先删除这些退化的边界。这可采用Sutherland-Hodgman提出的顶点排序方法[Suth74b]予以处理。

例3.30 Sutherland-Hodgman多边形裁剪。

考虑图3-36所示多边形在一个正方形窗口内的裁剪，窗口边界为：$x_{left}=-1$，$x_{right}=1$，$y_{bottom}=-1$，$y_{top}=1$。多边形顶点见表3-16。具体考察边P_1P_2对窗口左侧边界的裁剪。假定窗口边界取顺时针方向，即边界的右侧为窗口内侧或可见侧。采用代入法（见例3-28），判别函数 $x-w$ 为：

$$x-w=x-(-1)=x+1$$

表 3-16

	原多边形	对左边裁剪	对顶边裁剪	对右边裁剪	裁剪后的多边形
P_1	$(1/2,-3/2)$	$(1/2,-3/2)$	$(1/2,-3/2)$	$(1/2,-3/2)$	$(-1,-1)$
P_2	$(-2,-3/2)$	$(-1,-3/2)$	$(-1,-3/2)$	$(-1,-3/2)$	$(-1,1)$
P_3	$(-2,2)$	$(-1,2)$	$(-1,1)$	$(-1,1)$	$(1,1)$
P_4	$(3/2,2)$	$(3/2,2)$	$(3/2,1)$	$(1,1)$	$(1,0)$
P_5	$(3/2,0)$	$(3/2,0)$	$(3/2,0)$	$(1,0)$	$(1/2,0)$
P_6	$(1/2,0)$	$(1/2,0)$	$(1/2,0)$	$(1/2,0)$	$(1/2,1)$
P_7	$(1/2,3/2)$	$(1/2,3/2)$	$(1/2,1)$	$(1/2,1)$	$(-1,1)$
P_8	$(-3/2,3/2)$	$(-1,3/2)$	$(-1,1)$	$(-1,1)$	$(-1,0)$
P_9	$(-3/2,1/2)$	$(-1,0)$	$(-1,0)$	$(-1,0)$	$(0,-1)$

对子 $P_1\left(\frac{1}{2},-\frac{3}{2}\right)$ 有：

$$x_1+1=\frac{1}{2}+1>0$$

故P_1位于裁剪面的右侧，可见。

对于 $P_2\left(-2,-\frac{3}{2}\right)$ 有：

$$x_2+1=-2+1<0$$

故P_2不可见。因此边P_1P_2穿过裁剪边界，需要计算交点。采用参数直线求交方法(见例3.29)解得：$x=-1$，$y=-\frac{3}{2}$。

逐次裁剪过程如图3-36所示。特别注意最后一步，即对窗口底边的裁剪。在此之前P_1尚未被裁剪掉，各中间多边形顶点和初始多边形顶点保持同一顺序。但在对窗口底边的裁剪中，P_1被裁剪掉。最终裁剪结果多边形顶点表从对应子P_2的中间顶点开始。最后一个顶点为原多

边形边P_0P_1与窗口底边的交点。

注意裁剪结果多边形在裁剪窗口左上角处有四条退化边，如图3-36所示。 □

这里叙述的Sutherland-Hodgman算法侧重于二维窗口裁剪，但实际上该算法是通用的。它可用于在一个凸的裁剪体内裁剪任一平面或非平面多边形，其中计算直线段与三维裁剪面的交点时可用Cyrus-Beck方法。Sutherland-Hodgman算法还可用于分割凹多边形(见3.10节及[Suth74b])。

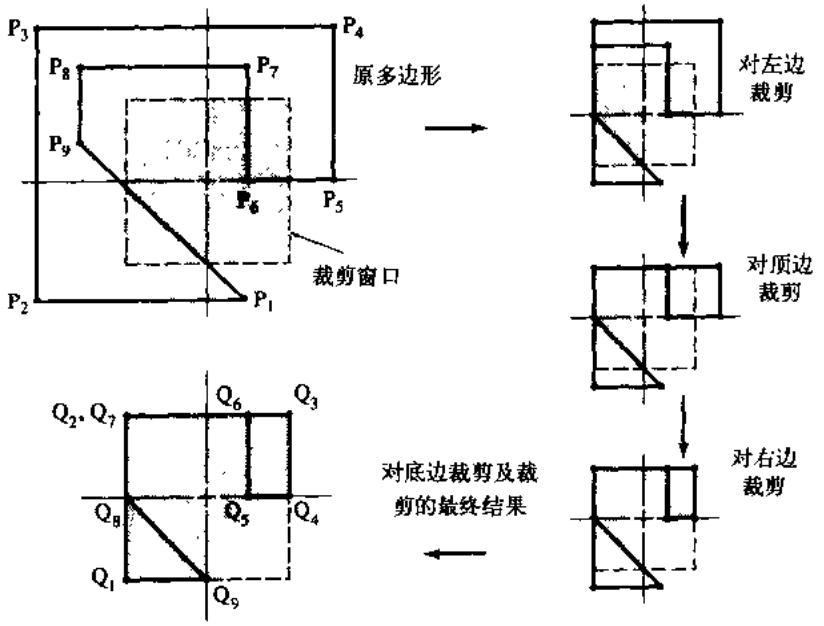


图3-36　例3.28的结果

3.20　Liang-Barsky多边形裁剪

Liang和Barsky(见[Lian83])提出了一个多边形裁剪的新算法。该算法对矩形窗口裁剪作了优化，但也可推广到任意的凸裁剪窗口。算法基础是他们的二维和三维线段裁剪概念（见[Lian84])。试验表明，对于矩形窗口，该优化算法比Sutherland-Hodgman算法快约两倍。

在Liang-Barsky算法中，假定裁剪区域和多边形共面，裁剪区域或窗口的每一条边将平

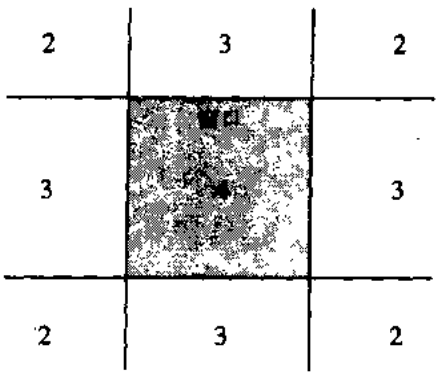


图3-37　裁剪平面的九个子区域

面分划为两个半平面。包含裁剪窗口的半平面称为可见或内半平面，另一个称为不可见或外半平面。窗口的四条边将平面划分为九个区域，如图3-37所示。每一个区域都位于一定数目的半平面的可见侧，但只有裁剪窗口位于所有四条边的内侧。

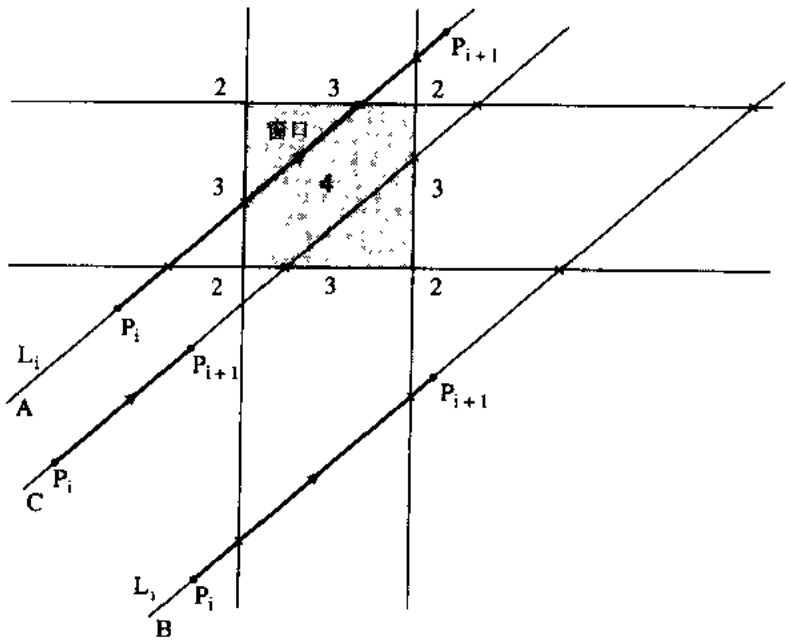


图3-38 对角直线与窗口的关系

3.20.1 进点和出点

假定多边形的边P_iP_{i+1}既非水平也非竖直，包含P_iP_{i+1}的无限长直线L_i和窗口的四条边都相交，交点用×标记，如图3-38所示。事实上，一条无限长直线无论是否和窗口相交，它总是从一个标号为2的角域出发，结束于对角线方向的另一个角域。在直线L_i和窗口边界的第一个交点处，直线从窗口边界的不可见侧进入可见侧，这种交点称为进点。在直线和窗口边界的最后交点处，直线从窗口边界的可见侧进入不可见侧，这种交点称为出点。中间两个交点出现的顺序可以是先进点后出点，如图中直线A和C；也可以是先出点后进点，如图中的直线B。

如果位于中间的两个交点的出现顺序是先进后出，那么无限长直线L_i和窗口相交，P_iP_{i+1}可能在窗口中可见。P_iP_{i+1}是全部可见还是部分可见取决于它在L_i上的位置。参见图3-38中位于直线A、C上的多边形边。

如果中间两个交点的出现顺序为先出后进（如直线B），那么无限长直线上的任何部分都不可能在窗口中可见，多边形边P_iP_{i+1}上的任何部分也不可能在窗口中可见。这与Cyrus-Beck算法判别非显然不可见线段问题等价(见3.5节和例3.12)。例如,图3-38中的直线B先与窗口的右边界相交，后与底边相交，因而不可见。

3.20.2 折点

多边形边P_iP_{i+1}的可见性状态对裁剪结果多边形的影响亦取决于其后续边。如果后续边从窗

口的另一边界进入窗口，则有必要把窗口的一个或多个角点添加到输出多边形中。这正是线裁剪和多边形裁剪之间的本质区别。Liang和Barsky称这样的窗口角点为折点，图3-39是两个折点的例子。

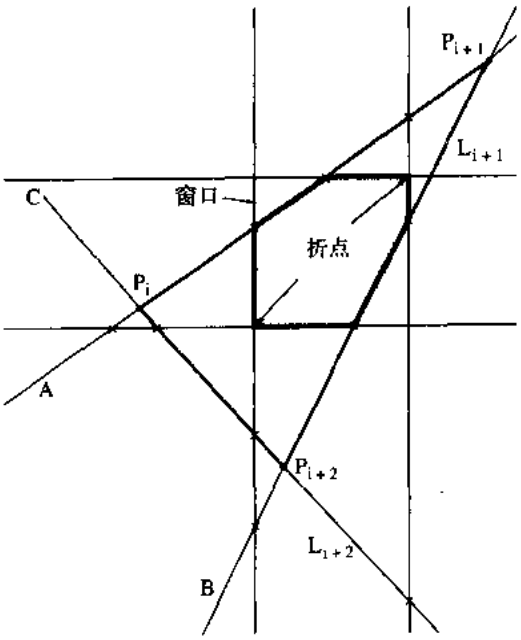


图3-39　折点

折点存在的一个必要但非充分的条件是边$P_{i+1}P_{i+2}$上有一个位于窗外的进点(见图3-39中的直线B和C)。当该条件成立时，Liang-Barsky多边形裁剪算法将离进点最近的那个窗口顶点加到输出多边形中。因为有可能整个多边形都位于窗外，所以该条件并非充分条件。最后的输出多边形中可能含有多余的折点，这是Liang-Barsky算法最主要的缺陷。例如图3-40中的多边形与窗口不相交，但由Liang-Barsky算法输出的裁剪结果多边形为窗口本身。相比而言，Sutherland-Hodgman算法输出一个退化的多边形即窗口的底边；Weiler-Atherton算法(见3.21节)则可生成正确结果。

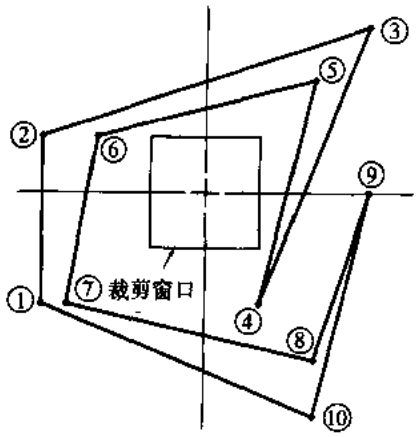


图3-40　包围裁剪窗口的非相交多边形

然而注意到，除非第一个进点是窗口的一个角点，即第一和第二个进点重合，一般而言，第一个进点不可能位于窗口之内。于是得到将一折点加到输出多边形中的充分条件：边P_iP_{i+1}包含了第一个进点。但算法中没有利用这个条件。

利用边的参数方程可以十分方便地计算直线L_i和窗口边界的交点并对这些交点排序：

$$P(t) = P_i + (P_{i+1} - P_i)t$$

这里$0 < t < 1$，表示多边形的边不包含起点P_i。若$-\infty < t < \infty$，则表示包含这条边的一条无限长直线。将起点排除在外是为了避免多边形的顶点在两条相邻的边中被重复地考虑两次。

3.20.3 算法设计

因为Liang-Barsky算法假定了一个矩形裁剪区域，所以利用参数方程的分量形式可以最有效地设计算法：

$$x = x_i + (x_{i+1} - x_i)t = x_i + \Delta x\, t$$
$$y = y_i + (y_{i+1} - y_i)t = y_i + \Delta y\, t$$

矩形裁剪区域边界可由下式给出：

$$x_{\text{left}} < x < x_{\text{right}}$$
$$y_{\text{bottom}} < y < y_{\text{top}}$$

用e和l两种下标分别表示进点和出点，窗口四条边界上交点的参数值为

$$t_{x_e} = (x_e - x_i)/\Delta x_i \qquad \Delta x_i \neq 0$$
$$t_{x_l} = (x_l - x_i)/\Delta x_i \qquad \Delta x_i \neq 0$$
$$t_{y_e} = (y_e - y_i)/\Delta y_i \qquad \Delta y_i \neq 0$$
$$t_{y_l} = (y_l - y_i)/\Delta y_i \qquad \Delta y_i \neq 0$$

其中

$$x_e = \begin{cases} x_{\text{left}} & \Delta x_i > 0 \\ x_{\text{right}} & \Delta x_i < 0 \end{cases}$$
$$x_l = \begin{cases} x_{\text{right}} & \Delta x_i > 0 \\ x_{\text{left}} & \Delta x_i < 0 \end{cases}$$
$$y_e = \begin{cases} y_{\text{bottom}} & \Delta y_i > 0 \\ y_{\text{top}} & \Delta y_i < 0 \end{cases}$$
$$y_l = \begin{cases} y_{\text{top}} & \Delta y_i > 0 \\ y_{\text{bottom}} & \Delta y_i < 0 \end{cases}$$

第一和第二个进点、出点分别为

$$t_{e_1} = \min(t_{x_e}, t_{y_e})$$
$$t_{e_2} = \max(t_{x_e}, t_{y_e})$$
$$t_{l_1} = \min(t_{x_l}, t_{y_l})$$
$$t_{l_2} = \max(t_{x_l}, t_{y_l})$$

在下面三种情况下，多边形边P_iP_{i+1}对输出多边形没有贡献：

1) 该边终止于第一个进点之前（$1 < t_{e_1}$），例如图3-41中的直线A。

2) 该边开始于第一个进点之后（$0 > t_{e_1}$），终止于第二个进点之前（$1 < t_{e_2}$），例如图3-41中的直线B。

3) 该边开始于第二个进点之后（$0 > t_{e_2}$）和第一个出点之后（$0 > t_{l_1}$），例如图3-41中的直线C。

在下面三种情况下，多边形边对输出多边形有贡献。第一种情况是考虑部分或完全可见的边：

4) 该边部分或完全包含在窗口之中，为部分可见或完全可见。条件是第二个进点位于第一个出点之前（$t_{e_2} < t_{l_1}$），且P_i位于第一个出点之前（$0<t_{l_1}$），P_{i+1}位于第二个进点之后（$1 > t_{e_2}$）。例如图3-41中的直线D、E 和F。

若该边为部分可见，则求出交点并将其输出到裁剪结果多边形。这只需将交点的参数代入直线参数方程求得一个坐标值，另一个坐标值由窗口相应的边界值给出。

在下面两种情况下需在输出结果多边形中添加折点。此时P_iP_{i+1}包含了一个位于窗口之外的进点：

5) 如果第一个进点位于P_iP_{i+1}内（$0<t_{e_1}<1$），则折点为(x_e,y_e)。例如图3-41中的直线G。

6) 如果第二个进点位于P_iP_{i+1}内（$0<t_{e_2}<1$），且位于窗口之外（$t_{l_1}<t_{e_2}$），那么，当它位于窗口的垂直边界上时（$t_{x_e}>t_{y_e}$），折点为（x_e,y_l），例如图3-41中的直线H。当它位于窗口的水平边界上时（$t_{y_e}>t_{x_e}$），折点为（x_l, y_e），例如图3-41中的直线I。

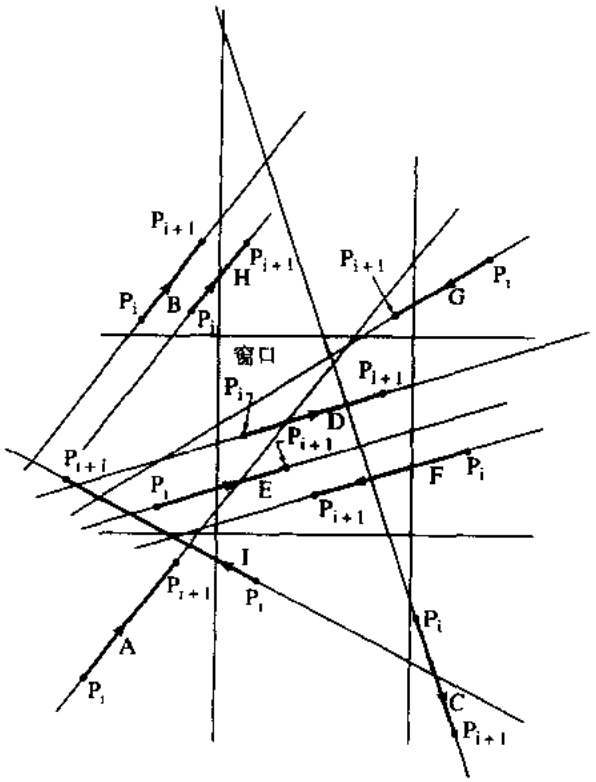


图3-41　多边形边的情况

表3-17 多边形边的输出情况

条 件	贡 献
1. $1<t_{e_1}$	无
2. $0>t_{e_1}$且$1<t_{e_2}$	无
3. $0>t_{e_2}$且$0>t_{l_1}$	无
4. $t_{e_2}<t_{l_1}$且$0<t_{l_1}$及$1>t_{e_2}$	部分可见段
5. $0<t_{e_1}<1$	折点及(x_e,y_e)
6. $0<t_{e_2}<1$且$t_{l_1}<t_{e_2}$	折点，若$t_{x_e}>t_{y_e}$，取(x_e, y_l)，否则取(x_l, y_e)

上述六种情况总结于表3-17。除了第5种情况之外，表3-17中的每一情况都是相互独立的。然而，当根据情况5输出一个折点后，如果多边形边穿过窗口，那么根据条件4,应该产生一条可见线段。如果该边整个位于窗口之外，那么根据条件6，将输出另一个折点。显然这两种情况不可能同时出现，因此构造算法时要特别仔细地考虑这些可能情况，也就是要在考虑情况4、6之前先考虑情况5。

3.20.4 水平边和垂直边

到目前为止尚没有考虑水平的和垂直的多边形边，这些边很容易通过判别Δy或$\Delta x=0$予以识别。处理水平和垂直多边形边有两种技术可以采用：第一种方法是将它们作为特殊情况来处理，例如一条水平的多边形边只能和窗口的两条垂直边界相交。一条垂直的多边形边则只能和窗口的两条水平边界相交(见习题3.14)。或者水平、垂直边可能完全位于窗外但仍然输出一个折点。第二种方法是把水平边和垂直边当作一般情况统一处理，这也是我们采用的方法。

考虑一条几乎水平的直线和窗口相交，例如图3-41中的直线E和F。对这样的一条直线有

$$t_{y_e} < t_{x_e} < t_{x_l} < t_{y_l}$$

当直线愈来愈趋向水平时，有：$\Delta y \to 0$、$t_{y_e} \to -\infty$和$t_{y_l} \to +\infty$。所以水平直线的特征是

$$-\infty < t_{x_e} < t_{x_l} < +\infty$$

类似地，考虑一条几乎垂直的直线和窗口相交，例如图3-41中的直线C，有

$$t_{x_e} < t_{y_e} < t_{y_l} < t_{x_l}$$

当直线愈来愈趋向垂直时有：$\Delta x \to 0$和$t_{x_e} \to -\infty$和$t_{x_l} \to +\infty$，所以垂直线的特征是

$$-\infty < t_{y_e} < t_{y_l} < +\infty$$

对于一条位于窗外、几乎垂直或水平的直线，例如图3-41中的直线B，依据它的方向有

$$t_{y_e} < t_{y_l} < t_{x_e} < t_{x_l} \quad 或 \quad t_{x_e} < t_{x_l} < t_{y_e} < t_{y_l}$$

若直线水平，则t_{y_e}和t_{y_l}为无穷大，上述不等式变成：

$$-\infty < -\infty < t_{x_e} < t_{x_l} \quad 或 \quad t_{x_e} < t_{x_l} < +\infty < +\infty$$

若直线垂直，则t_{x_e}和t_{x_l}无穷大，上述不等式变成：

$$t_{y_e} \le t_{y_l} < +\infty < +\infty \quad 或 \quad -\infty < -\infty < t_{y_e} < t_{y_l}$$

从算法的角度上说，使用$-\infty$更加有效。

3.20.5 算法

下面是该算法的结构:

```
Outline of Liang–Barsky polygon clipping algorithm
  for regular rectangular windows

for each polygon edge P(i)P(i+1)
    determine the direction of the edge
      and whether it is diagonal, vertical or horizontal
    determine the order of the edge–window intersections
    determine the t-values of the entering edge–window intersections
      and the t-value for the first leaving intersection
    analyze the edge
    if 1 >= t_enter_1 then        condition 2 or 3 or 4 or 6 and not 1
       if 0 < t_enter_1 then        condition 5 — turning vertex
          call output(turning vertex)
       end if
       if 1 >= t_enter_2 then        condition 3 or 4 or 6 and not 1 or 2
          determine second leaving t-value
          there is output
          if (0 < t_enter_2) or (0 < t_leave_1) then cond. 4 or 6, not 3
             if t_enter_2 <= t_leave_1 then  cond. 4 — visible segment
                call output(appropriate edge intersection)
             else  end condition 4 and begin condition 6 — turning vertex
                call output(appropriate turning vertex)
             end if       end condition 6
          end if       end condition 4 or 6
       end if       end condition 3 or 4 or 6
    end if       end condition 2 or 3 or 4 or 6
next i       end edge P(i)P(i+1)
```

下面是完整算法的伪代码程序。

```
Liang–Barsky polygon clipping algorithm
  for regular rectangular windows

subroutine lbpoly(Nin,x,y,W;Nout,Q)
x is the x component of the input polygon array
y is the y component of the input polygon array
Q is the output polygon array
W is the clipping window array.
  W(1) = xleft
  W(2) = xright
  W(3) = ybottom
  W(4) = ytop
Nin is the number of input polygon vertices
Nout is the number of output polygon vertices
all polygon vertices are given in counterclockwise order
assume that infinity is represented by some real number

set up window edges
```

```
xleft = W(1)
xright = W(2)
ybottom = W(3)
ytop = W(4)
close the polygon
x(Nin + 1) = x(1)
y(Nin + 1) = y(1)
begin the main loop
for i = 1 to Nin
    deltax = x(i + 1) − x(i)
    deltay = y(i + 1) − y(i)
    determine the direction of the edge
      and whether it is diagonal, vertical or horizontal
    determine the order of the edge–window intersections
    if deltax > 0 then      not a vertical line
       x_enter = xleft      diagonal line running left to right
    x_leave = xright
 else if deltax < 0 then      diagonal line running right to left
    x_enter = xright
    x_leave = xleft
 else    vertical line (insure correct x_enter,x_leave,y_enter,y_leave)
    if y(i + 1) > y(i) then
       y_enter = ybottom
       y_leave = ytop
    else
       y_enter = ytop
       y_leave = ybottom
    endif
endif
if deltay > 0 then      not a horizontal line
    y_enter = ybottom      diagonal line running bottom to top
    y_leave = ytop
else if deltay < 0 then      diagonal line running top to bottom
    y_enter = ytop
    y_leave = ybottom
else    horizontal line (insure correct x_enter,x_leave,y_enter,y_leave)
    if x(i + 1) > x(i) then
       x_enter = xleft
       x_leave = xright
    else
       x_enter = xright
       x_leave = xleft
    end if
end if
determine the t-values of the entering edge–window intersections
  and the t-value for the first leaving intersection
if deltax <> 0 then
    t_enter_x = (x_enter − x(i))/deltax      diagonal line
else
```

```
    t_enter_x = −∞        vertical line
  end if
  if deltay <> 0 then
    t_enter_y = (y_enter − y(i))/deltay        diagonal line
  else
    t_enter_y = −∞        horizontal line
  end if
  if t_enter_x < t_enter_y then        first entry at x then y
    t_enter_1 = t_enter_x
    t_enter_2 = t_enter_y
  else      first entry at y then x
    t_enter_1 = t_enter_y
    t_enter_2 = t_enter_x
  end if
  analyze the edge
  if 1 >= t_enter_1 then        condition 2 or 3 or 4 or 6 and not 1
    if 0 < t_enter_1 then        condition 5 — turning vertex
      call output(xin,yin,Nout,Q)
    end if
    if 1 >= t_enter_2 then        condition 3 or 4 or 6 and not 1 or 2
      determine second leaving t-value
      if deltax <> 0 then        diagonal line
        t_leave_x = (x_leave − x(i))/deltax
      else      vertical line
        if (xleft <= x(i)) and (x(i) <= xright) then      P(i) inside
          t_leave_x = ∞
        else      P(i) outside
          t_leave_x = −∞
        end if
      end if
      if deltay <> 0 then        diagonal line
        t_leave_y = (y_leave − y(i))/deltay
      else      horizontal line
        if (ybottom <= y(i)) and (y(i) <= ytop) then
          t_leave_y = ∞        P(i) inside
        else
          t_leave_y = −∞        P(i) outside
        end if
      end if
      if t_leave_x < t_leave_y then        first exit at x
        t_leave_1 = t_leave_x
      else      first exit at y
        t_leave_1 = t_leave_y
      end if
      there is output
      if (0 < t_enter_2) or (0 < t_leave_1) then  cond. 4 or 6, not 3
        if t_enter_2 <= t_leave_1 then  cond. 4 — visible segment
          if 0 < t_enter_2 then        P(i) outside window
```

```
            if t_enter_x > t_enter_y then        vertical boundary
               call output(xin,y(i) + tinx*deltay; Nout,Q)
            else       horizontal boundary
               call output(x(i) + tiny*deltax,yin; Nout,Q)
            end if
         end if
         if 1 > t_leave_1 then        P(i+1) outside window
               if t_leave_x < t_leave_y then        vertical boundary
                  call output(xout,y(i)+toutx*deltay; Nout,Q)
               else       horizontal boundary
                  call output(x(i) + touty*deltax,yout; Nout,Q)
               end if
            else       P(i+1) inside window
               call output(x(i+1),y(i+1); Nout,Q)
            end if       end condition 4
         else       condition 6 — turning vertex
            if t_enter_x > t_enter_y then        second entry at x
               call output(xin,yout; Nout,Q)
            else       second entry at y
               call output(xout,yin; Nout,Q)
            end if
         end if       end condition 6
      end if       end condition 4 or 6
   end if       end condition 3 or 4 or 6
end if       end condition 2 or 3 or 4 or 6
next i       end edge P(i)P(i+1)
return
```

3.21 凹裁剪区域——Weiler-Atherton算法

前面讨论的裁剪算法均要求裁剪区域是凸的，然而在许多应用场合如消除隐藏面等，则需要考虑对凹区域的裁剪。由Weiler和Atherton(见[Weil77])提出的裁剪算法可以满足这一要求。该算法功能强大，但略显复杂。它可以用一个含内孔的凹多边形去裁剪另一个也有内孔的凹多边形。被裁剪的多边形简称为*主多边形*，裁剪区域称为*裁剪多边形*。不难看出，主多边形被裁剪多边形裁剪后所产生的新的边界实际上是裁剪多边形边界的一部分。因此无需额外生成新的边界，这样裁剪所得的多边形的数目是最少的。

算法将主多边形和裁剪多边形定义为顶点的环形列表，多边形的外边界取顺时针方向,内边界或孔取逆时针方向。这种约定保证了在遍历顶点列表的时候，多边形的内部总是位于前进方向的右侧。主多边形和裁剪多边形的边界可能相交，也可能不相交。若它们相交，则交点必成对地出现。其中一个交点为主多边形边进入裁剪多边形内部时的交点，而另一交点为其离开时的交点。算法从进入交点开始，沿主多边形的外部边界按照顺时针方向向前跟踪，直至找到它与裁剪多边形的一个交点为止。在交点处再向右转，开始沿裁剪多边形的外部边界按照顺时针方向跟踪，直至发现它与主多边形的一个交点后再向右转，再次沿主多边形的边界跟踪。继续上述过程，直至到达算法起始点位置。主多边形的内边界则按逆时针方向跟踪，见图3-42。

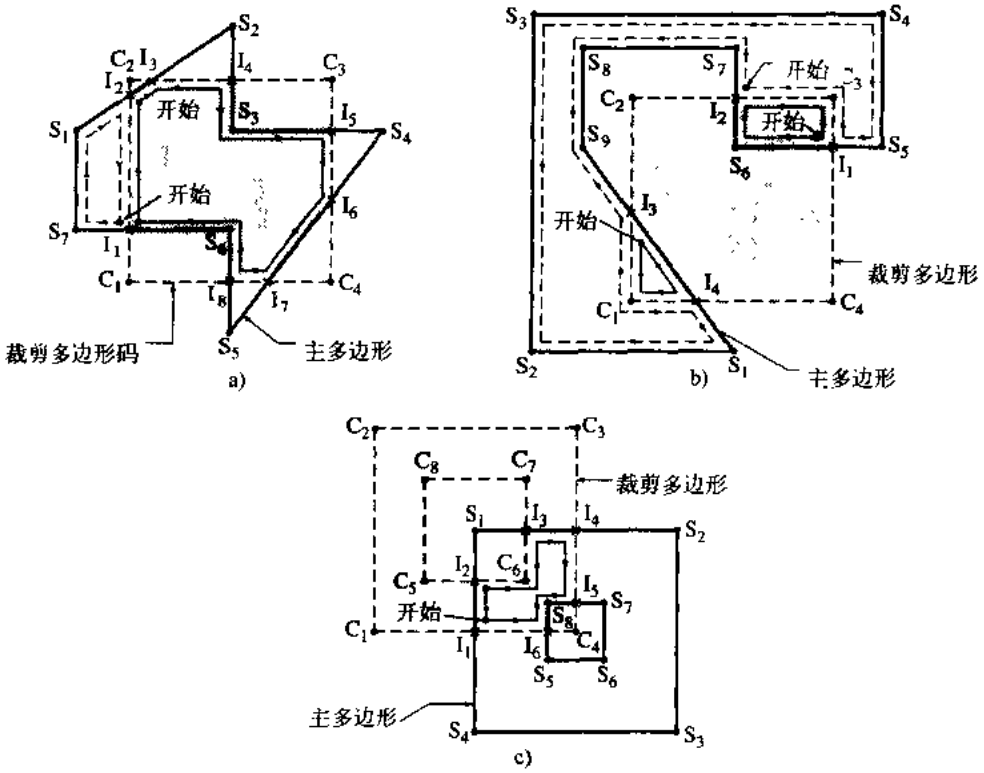


图3-42　Weiler-Atherton 裁剪

a) 简单多边形　b) 包围多边形　c) 带孔的多边形和窗口

该算法可以正式叙述如下。

求出主多边形与裁剪多边形的交点:

将交点加入到主多边形和裁剪多边形的顶点表中，并注以标记。对同一交点，在主多边形和裁剪多边形间建立双向链接。

处理不相交的多边形边界:

建立两个表，分别记录位于裁剪多边形内部的边界和外部的边界。位于主多边形外的裁剪多边形边界被忽略，而位于主多边形内的裁剪多边形边界将构成主多边形的孔。因此裁剪多边形的边界将被适当拷贝到主多边形的内表和外表之中。

建立两类交点表:

其一为进点表，它仅包含主多边形边进入裁剪多边形内部时的交点。另一个是出点表，它包含主多边形边离开裁剪多边形内部时的交点。沿着多边形边界，两类交点将交替出现。因此对于每一对交点，只需进行一次类型判别就够了。

进行裁剪:

位于裁剪多边形内的多边形可按下述方法求出:

若交点表为空，则处理结束。否则从进点表中取一个交点。

跟踪主多边形顶点表，直至遇到下一个交点。复制这一段主多边形顶点表并记入内表中。

根据交点间的链接，转到裁剪多边形顶点表中的相应位置。

跟踪裁剪多边形顶点表，直至遇到下一交点。复制这一段裁剪多边形顶点表并

记入内表中。

根据交点间的链接，再转回主多边形顶点表。

重复上述过程，直至回到起始交点处。至此所得的位于裁剪多边形内的新多边形已经封闭。

位于裁剪多边形外的多边形也可用同一方法求得，不过这时起始交点应取自出点表且应沿相反方向跟踪裁剪多边形顶点表。新生成的多边形顶点表将记入外表中。

将孔即内部边界连到相应的外部边界上。由于外部边界取顺时针方向而内部边界取逆时针方向，所以可通过检查边界的走向而实现。处理结束。

下面几个例子将进一步说明算法的实现过程。

例3.31 Weiler-Atherton多边形裁剪——简单多边形。

考虑一个主多边形在一个裁剪正方形内的裁剪，如图3-42a所示。两个多边形之间的交点在图中标识为I_i。下表给出主多边形和裁剪多边形的顶点表。交点I_2、I_4、I_6和I_8置于进点表中，而I_1、I_3、I_5和I_7则置于出点表中。

为了获得内裁剪结果多边形，先取出进点表中的第一个交点I_2。执行前面所述过程的结果见图3-42a中以及上表中主多边形和裁剪多边形表中带箭头的实线。所生成的内裁剪结果多边形为

$$I_2I_3I_4S_3I_5I_6I_7I_8S_6I_1I_2$$

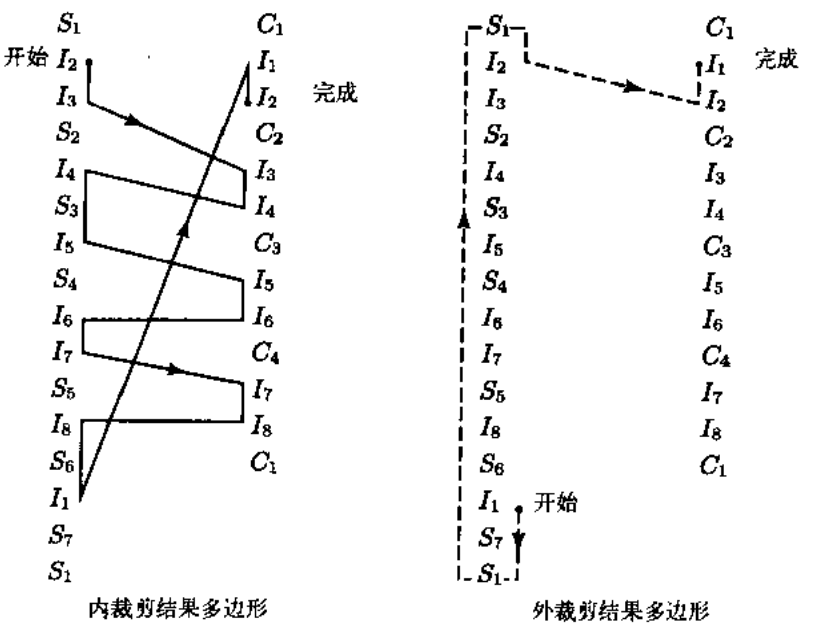
若取进点表中其他交点作为起点，如I_4、I_6或I_8等，所得裁剪结果相同。

为了获得外裁剪结果多边形，先从出点表中取出第一个交点I_1。执行前面描述过程的结果见图3-42a中以及上表中主多边形和裁剪多边形链表中带箭头的虚线。所生成的外裁剪结果多边形为

$$I_1S_7S_1I_2I_1$$

若从出点表中陆续取出交点I_3、I_5和I_7作为起点，则分别生成下列外裁剪结果多边形：

$$I_3S_2I_4I_3 \qquad I_5S_4I_6I_5 \qquad I_7S_5I_6I_7$$



例3.32中将讨论一个更为复杂的主多边形，它部分地包围裁剪多边形。 □

例3.32 Weiler-Atherton 多边形裁剪——外围多边形。

主多边形和裁剪多边形以及它们的交点如图3-42b所示。交点I_1、I_3置于进点表中。而I_2、I_4则置于出点表中。下表列出了主多边形表和裁剪多边形表。

主多边形表	裁剪多边形表	主多边形表	裁剪多边形表
S_1	C_1	S_1	C_1
S_2	I_3	S_2	I_3
S_3	完成 C_2	S_3	C_2
S_4	I_2	S_4	完成 I_2
S_5	C_3	S_5	C_3
开始 I_1	完成 I_1	I_1	I_1
S_6	C_4	S_6	C_4
I_2	I_4	I_2	开始 I_4
S_7	C_1	S_7	C_1
S_8		S_8	
S_9		S_9	
开始 I_3		I_3	
I_4		I_4	
S_1		S_1	
内裁剪结果多边形		外裁剪结果多边形	

为了得到内裁剪结果多边形，先后从进点表中取出I_1、I_3作为起点。裁剪结果见图3-42b以及上表中主多边形和裁剪多边形表中带箭头的实线。所生成的内裁剪结果多边形分别为

$$I_1S_6I_2C_3I_1 \quad 和 \quad I_3I_4C_1I_3$$

从出点表中取出I_2作为起点，求得外裁剪结果多边形为

$$I_2S_7S_8S_9I_3C_1I_4S_1S_2S_3S_4S_5I_1C_3I_2$$

若先取出I_4作为起点，将得到同一多边形。裁剪结果见图3-42b以及上表中主多边形和裁剪多边形表中带箭头的虚线。注意，为了生成外裁剪结果多边形,必须反向遍历裁剪多边形表。 □

最后一例考虑带孔的凹多边形在另一有孔的凹多边形窗口内的裁剪。

例3.33 Weiler-Atherton裁剪——带内孔的边界。

主多边形和裁剪多边形以及它们的交点如图3-42c所示。交点I_1、I_3和I_5置于进点表中，而I_2、I_4和I_6置于出点表中。下表列出了主多边形和裁剪多边形表。

注意内边界即孔上的顶点按逆时针方向排列，内边界表和外边界表均为独立环形表。

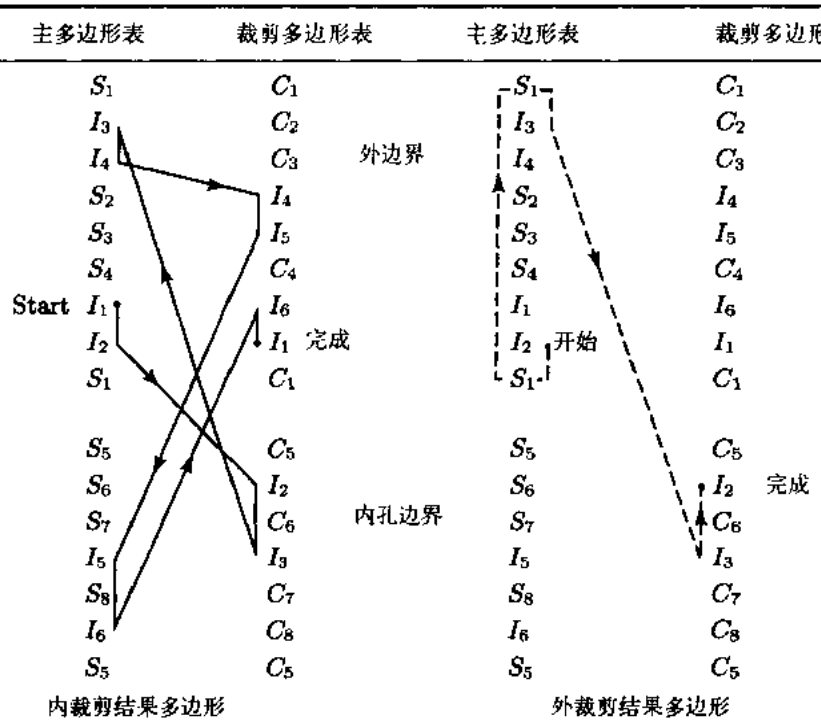
若从进点表中取出交点I_1作为起点，则生成的内裁剪结果多边形为

$$I_1I_2C_6I_3I_4I_5S_8I_6I_1$$

如图3-42c以及上表中主多边形和裁剪多边形表中带箭头的实线所示。若取出I_3、I_5，作为起点，所得结果相同。

从出点表中取出I_2作为起点，生成外裁剪结果多边形为

$$I_2S_1I_3C_6I_2$$



注意，主多边形表包含有两个分离的边界，即内边界和外边界，它们均各自封闭。因此，将从外边界表尾的S_1转回到表的首端S_1而不是转到内孔边界表上的S_5。从外边界转到内边界总是发生在由主多边形表跳转到裁剪多边形表的时候，或反之。如上面主多边形表和裁剪多边形表中的虚线所示。类似地，从出点表中取I_4或I_6作为起点，将生成外裁剪结果多边形：

$$I_4S_2S_3S_4I_1I_6S_5S_6S_7I_5I_4$$

□

特殊情形

要保证Weiler-Atherton算法工作可靠,必须仔细地判定交点的有无及配置情况。例如在恰好接触情况下，即当主多边形的顶点或一条边位于裁剪多边形的边上或与裁剪多边形的边重合时，它们之间应看作不相交。

多边形边的交点可出现在一个多边形的顶点上，如图3-43所示；也可能出现在两个多边形的顶点上，如图3-44所示。在这些图中，虚线表示裁剪多边形的边，实线表示主多边形的边，箭头表示边的方向。若同一条边的两端均附有箭头，则表示可以取其中任一方向作为该边的方向。

图3-43a和图3-44a中的交点，为应加入主多边形和裁剪多边形链表中的真正的交点，而图3-43b和3-43c及对称情况为接触型交点，不应加入到主多边形和裁剪多边形链表中去。图3-44b和图3-44c也是接触型交点，不应加入到主多边形和裁剪多边形链表中。

若多边形的某边与同一多边形的另外一些边在一个或几个点上形成接触型交点，则该多边形称为退化多边形。如图3-43c和3-44c所示，不包括它们的反射情况，可能会产生一些退化的边，如图3-45所示。算法中必须识别和防止出现这些退化的边。

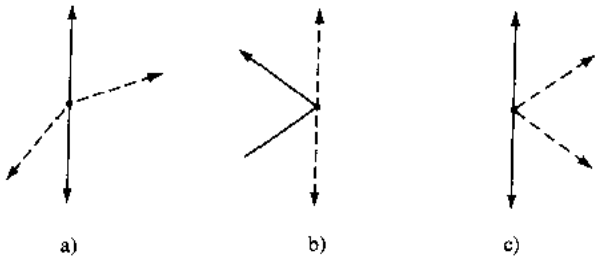


图3-43　单个顶点与多边形边的交——虚线为裁剪多边形，实线为主多边形

a)真正相交——一个交点　b) 恰好接触——无交点
c) 恰好接触——无交点，可能为退化的多边形

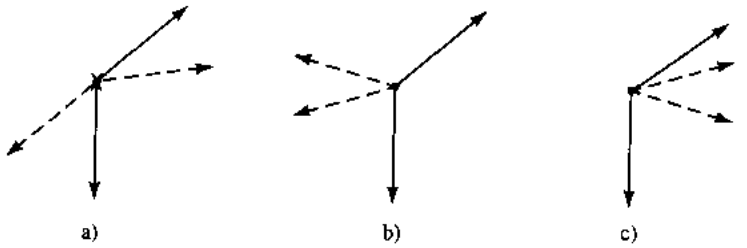


图3-44　两个顶点与多边形边的交——虚线为裁剪多边形，实线为主多边形

a) 真正相交——一个交点　b) 恰好接触——无交点
c) 恰好接触——无交点，可能为退化的多边形

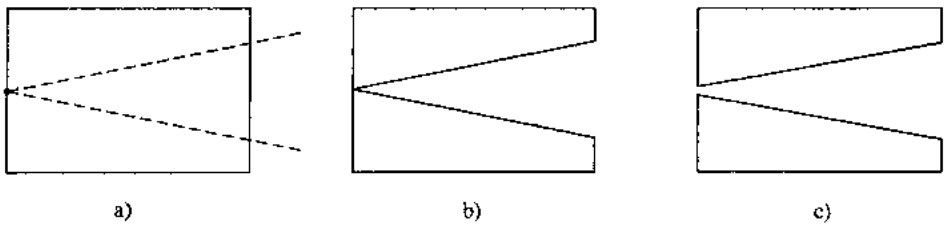


图3-45　退化边

a) 恰好接触点的原始多边形　b) 退化多边形　c) 转化为非退化的多边形

此外，多边形的边可能相互毗邻或重合，如图3-46所示:

如果边相互毗邻，如图3-46a所示，没有交点加入主多边形和裁剪多边形链表中。

如果裁剪多边形始于主多边形内部，和主多边形的一条边重合，然后又折回主多边形内部，如图3-46b所示，那么以×标识的两个交点被加入主多边形和裁剪多边形的列表中。

如果裁剪多边形始于主多边形内部，和主多边形的一条边重合，然后离开主多边形朝外延伸，如图3-46c所示，那么第一个交点（图中以×标识）加入主多边形和裁剪多边形的链表中。

如果裁剪多边形始于主多边形外部，和主多边形的一条边重合，然后进入主多边形的内部，如图3-46d所示，那么的第二个交点（图中以×标识）加入主多边形和裁剪多边形链表中。

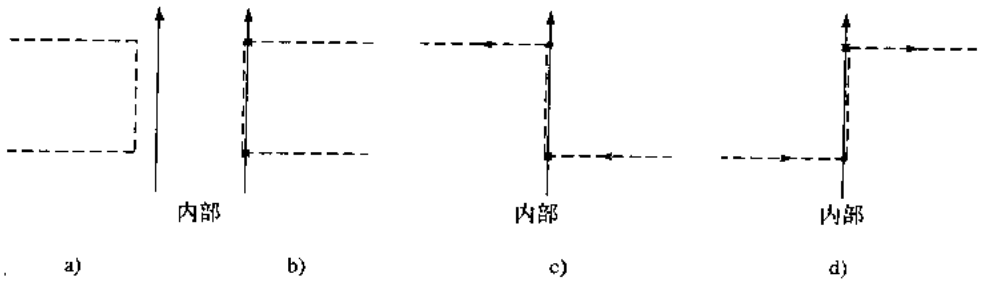


图3-46 重合的多边形边——虚线为裁剪多边形。实线为主多边形

a) 毗邻的边——无交点 b) 来自主多边形内部又折回内部的重合边——两个交点 c) 来自主多边形内部然后穿越到外部的重合边——一个交点 d) 来自主多边形外部然后进入其内部的重合边——一个交点

3.19节中确定点和平面的关系的一些测试方法(见例3.28)，可以用来确定多边形的一条边和其他边的关系。例如在图3-43a中，裁剪多边形的一个顶点位于主多边形一条边上。通过判定裁剪多边形上该顶点的前后相邻顶点分别位于主多边形的两侧可确认该顶点为一有效交点。当裁剪多边形的一条边与主多边形边重合时，则需要运用类似于4.13节中所介绍的Schumacker簇优先级测试的技术。

主多边形和裁剪多边形可能没有交点，这时称它们为松散的多边形。位于主多边形外部的裁剪多边形可以忽略。而位于主多边形内部的裁剪多边形在主多边形的内部裁去一块，形成一个孔。松散的主多边形或裁剪多边形被添加到进点或出点链表之中。具体如下:

松散主多边形:

如果位于裁剪多边形内部，则加入内裁剪结果多边形链表中（见图3-47a）；

如果位于裁剪多边形外部，则加入外裁剪结果多边形链表中（见图3-47b）。

松散主多边形内孔:

如果位于裁剪多边形内部，则加入内裁剪结果多边形内孔链表中（见图3-47c）

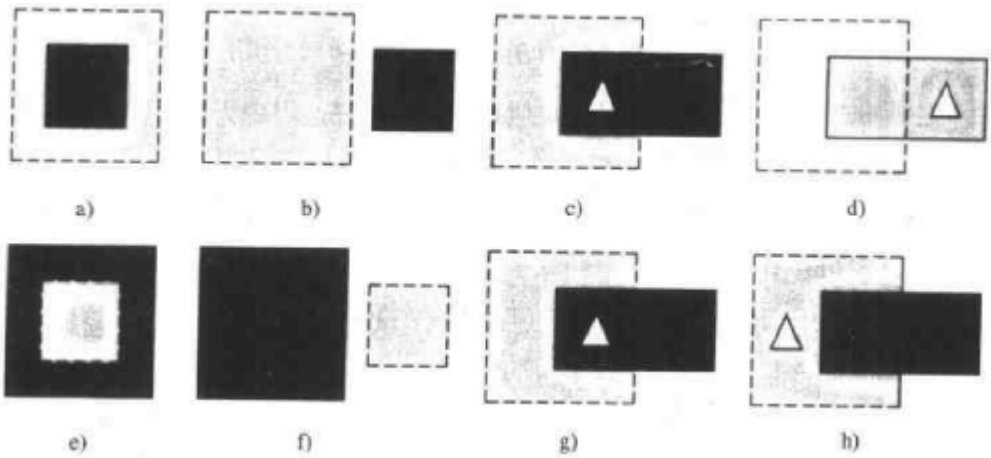


图3-47 松散多边形

a) 主多边形位于裁剪多边形内部 b) 主多边形位于裁剪多边形外部

c) 主多边形孔位于裁剪多边形内部 d) 主多边形孔位于裁剪多边形外部

e) 裁剪多边形位于主多边形内部 f) 裁剪多边形位于主多边形外部

g) 裁剪多边形孔位于主多边形内部 h) 裁剪多边形孔位于主多边形外部

如果位于裁剪多边形外部，则加入外裁剪结果多边形内孔链表中（见图3-47d）

松散裁剪多边形:

加入内裁剪结果多边形链表，见图3-47e。

改变多边形各边的方向，加入到外裁剪结果多边形链表中，见图3-47f。

松散内孔裁剪多边形:

加入外裁剪结果多边形链表,见图3-47g。

改变多边形各边的方向，也加入到内裁剪结果多边形链表中（见图3-47h）。

松散多边形可以用4.4节Warnock隐藏面算法中描述的多边形分离和包含测试方法确定。Weiler(见[Weil78])给出了具体实现的有关细节。

Weiler(见[Weil80])采用图论的概念重新讨论了裁剪算法，尤其是简化了对接触型交点和退化多边形情形的处理。但对算法的描述在概念上要复杂得多，因此这里仍采用了最初的算法描述方式。

3.22 字符裁剪

字符或文稿可由软件、固件或硬件产生，它既可由一些独立的线段、画线生成，也可用点阵表示来生成。用软件生成的画线式字符可以像其他线段一样，进行旋转、平移、缩放以及采用前面所述算法在任意取向为任意窗口所裁剪。图3-48显示了一个典型的例子。

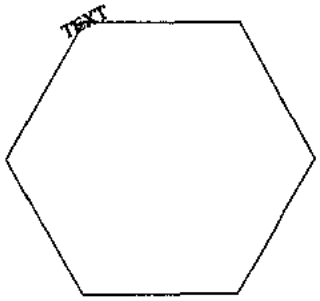


图3-48　软件画线式字符的裁剪

由软件生成的点阵式字符原则上可作类似处理，但处理过程较为繁琐。具体而言,当包含字符的字符方框为任一窗口裁剪时，必须将字符方框每一像素同裁剪窗口进行比较，以确定它位于窗内还是窗外。若位于窗内，则该像素被激活，否则不予考虑。点阵式字符的裁剪原理见图3-49。

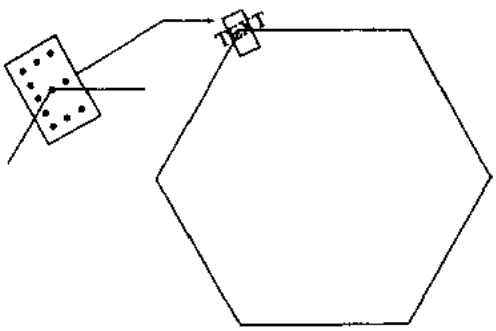
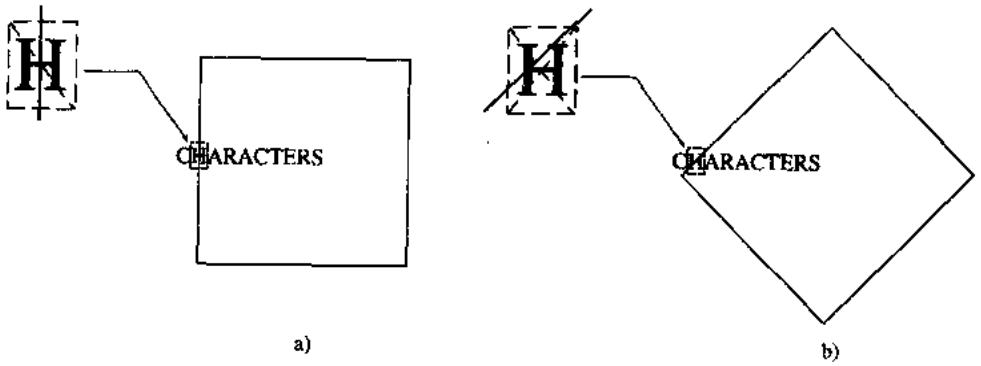


图3-49　软件点阵式字符的裁剪

由硬件生成的字符的裁剪受到较大的限制。一般而言，不是完全可见的字符均被消去。这可将字符方框同裁剪窗口进行比较，若整个方框位于窗口内，则显示相应字符，否则不予显示。当矩形字符方框与矩形窗口平行时，仅需取字符方框的任一对角线同裁剪窗口进行比较，见图3-50a。对于特殊形状窗口或窗口与字符方框不平行，方框的两对角线都需同裁剪窗口比较，见图3-50b。

如果字符由固件产生，那么字符裁剪工具可能非常有限也可能非常广泛，取决于固件中实现的裁剪算法。



a) b)

图3-50 硬件生成字符的裁剪