

电子科技大学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

硕士学位论文

MASTER DISSERTATION

论文题目 TrueType 字体引擎的研究与实现

学科专业 软件工程

指导教师 张建中 副教授

作者姓名 刘翔

班学号 200520606009

分类号_____

UDC^{注1}_____

学 位 论 文

TrueType 字体引擎的研究与实现

(题名和副题名)

刘 翔

(作者姓名)

指导教师姓名 **张建中** **副教授**

电子科技大学 成 都

(职务、职称、学位、单位名称及地址)

申请专业学位级别 **硕士** 专业名称 **软件工程**

论文提交日期 **2008.4** 论文答辩日期 **2008.5**

学位授予单位和日期 **电子科技大学**

答辩委员会主席_____

评阅人_____

2008 年 月 日

注 1：注明《国际十进分类法 UDC》的类号。

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

签名： 刘翔 日期： 2008 年 5 月 9 日

关于论文使用授权的说明

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

签名： 刘翔 导师签名： 张健
日期： 2008 年 5 月 9 日

摘 要

计算机字形信息处理技术是研究在计算机中描述、储存和以不同形式输出(即生成还原)字形符号形状的一个专门的研究领域。从上世纪七十年代开始,计算机突破过去的单纯以科学计算和数据处理的“计算”应用,进展到包括事务处理、文字处理等多方面的应用,各种文字在计算机中的存储问题即被提了出来。因此数字化的字形表示在计算机应用发展中发挥了重要作用。伴随光栅设备的出现和发展,作为计算机字形信息处理技术重要内容之一的计算机字形输出还原技术一直受到人们的重视。

由于 TrueType 字体的广泛应用,将由 TrueType 字体所描述的字符转化为相应 Bitmap 位图的 TrueType 字体引擎就成为了字体引擎家族中重要的成员,同时也是打印机驱动程序中的重要部分。所以对 TrueType 字体引擎的研究和实现有着比较重要的现实意义。同时由于本文所介绍的 TrueType 字体引擎是独立于微软字体引擎的,所以就使它有着更为灵活和广泛的应用空间。

为达到以上目的,在详细研究了 TrueType 字体技术, TrueType 字体文件的基本构成和一些背景知识的基础上,分析和研究了 TrueType 字体引擎的基本结构;并以此入手,提出了一种独立于微软字体引擎的,可用于打印机驱动程序的 TrueType 字体引擎设计模型;并最终实现了以上设计。

作为项目组成员,作者参与了整个项目的前期设计和全部资料分析;然后具体负责设计和实现了光栅化模块。

在光栅化模块的设计实现中,作者在研究了轮廓内外点判断方法的基础上,将传统的多边形扫描线填充算法进行了合理地改进,最后提出了一种适用于 TrueType 字体字符轮廓的可填充交叉轮廓的扫描线填充算法。它的特点是根据 TrueType 字体轮廓的特殊描述方式——用控制点来描述字体轮廓——提出了纵横双向扫描的扫描方式。纵横双向扫描不仅避免了对水平线段和垂直线段的特殊处理;而且还可以提高字体轮廓的填充精度。该填充算法还有一个特点就是通过两次“虚拟描画”分别记录轮廓顶点亦即轮廓与扫描线的交点坐标,更为重要的是它还同时记录了字体轮廓与扫描线相交的方向信息,这使它在填充轮廓时能很有效的通过线圈数法实现轮廓内外点判断,从而具有填充交叉轮廓的功能。

关键词: TrueType, 字体引擎, 轮廓, 光栅化

Abstract

The technique dealing with information of computer font is the special domain which researches how to describe, store and output font symbol in different form on the computer. From the seventies of the last century, the computer broke through the simple calculating application in science calculating and disposing data. It can be used to transaction processing, word processing and so on. And there is a problem that how to store the kinds of words. So the digital font plays an important role in the development of the computer applications. Along with the appearance and development of the raster equipment, as the important part of the technique dealing with information of computer font, the technique is laid store by people, which is used to revert and output the computer font.

Result of the wild use of the TrueType font, the TrueType font engine is main member in the font engine family and important part in the printer driver, which transforms the characters described by TrueType font into bitmap. There is the significance to investigate and implement the TrueType font engine. Meanwhile it is more agility and useful because of it is independent of Microsoft font engine, which is advanced by this disquisition.

For this purpose, this disquisition introduced the TrueType technique and the basic structure of the TrueType file. Proceed with this we research the key concepts about the TrueType font engine. At last we found out a design of the TrueType font engine, which is dependent of Microsoft font engine and can also be used in printer driver.

As a member of the project group, I participated in the whole design of the project and all analysis of the datum. Then I am with responsibility for the design and implement the raster module.

In this process, I improved on the traditional polygon scan line filling algorithm in reasons, base on the reaching of the algorithm which be used to judge the points which are inside the outline or not. In the end I found out a design of the polygon scan line filling algorithm, which applies to TrueType font character outlines that may be cross between outlines. It has several characteristics. One of them is scanning vertically and horizontally. It can improve filling precision and predigest disposal. Another one is

“virtually portrayal”. It can record the vertexes of the outline and the cross points between outlines and scan lines. And more important point is that it can record the direction when the scan line crosses the outline. With this information we can use winding number easily.

Key words: TrueType, font engine, outline, raster

目 录

第一章 引言	1
1.1 课题的来源及目的	1
1.2 课题的意义	1
1.3 文献综述	1
1.4 本文内容安排	2
第二章 TrueType 字体技术.....	3
2.1 Font 是什么?	3
2.1.1 Symbol Set 和 Typeface.....	3
2.1.2 Spacing.....	4
2.1.3 Pitch.....	4
2.1.4 Height.....	4
2.1.5 Style.....	4
2.1.6 Stroke Weight.....	5
2.2 什么是 TrueType 字体?	5
2.2.1 概念.....	5
2.2.2 Bitmap Font 和 Scalable Font.....	5
2.2.3 Glyph.....	5
2.2.4 TrueType 字体的历史.....	6
2.2.5 TrueType 字体的优点	6
2.3 True Type 字体文件解析.....	7
2.3.1 基本结构.....	7
2.3.2 数据类型.....	8
2.3.3 文件内容简介.....	8
2.3.3.1 TrueType 字体文件的开头	8
2.3.3.2 第一级目录.....	9
2.3.4 glyf 表详解	10
2.3.4.1 Glyph 描述部分	10
2.3.4.2 简单 glyph	11

2.3.4.3 复合 glyph	15
2.3.5 Cmap 表详解	19
第三章 坐标空间	23
3.1 各种坐标空间	23
3.1.1 设备空间 Device Space	23
3.1.2 用户空间 User Space	24
3.1.3 文本空间 Text space	25
3.1.4 轮廓空间 Glyph space	25
3.1.5 其他空间.....	25
3.1.6 各个坐标空间之间的转化.....	25
3.2 坐标变换的数学原理	26
3.3 几种基本的坐标变换矩阵	28
3.3.1 几种基本的坐标变换矩阵.....	28
3.3.2 复合变换.....	29
第四章 TrueType 字体引擎的基本原理	31
4.1 Font 文件的生成.....	31
4.2 从 Font 文件到纸面	31
4.3 TrueTypeFont 技术相关的基本概念.....	32
4.3.1 轮廓 (Outline)	32
4.3.2 Funits 和 EM.....	33
4.3.3 Funits 和格栅 (grid)	34
4.4 放缩 (Scale) glyph	36
4.4.1 从 FUnits 到 Pixels	36
4.4.2 光栅设备的特性.....	36
4.5 指令化 (Gridfitting)	38
4.5.1 指令.....	39
4.5.2 TrueType 指令解释器	40
4.5.3 指令的运用.....	41
4.5.3.1 Font 程序	42
4.5.3.2 CVT 程序	42
4.5.4 存储区域.....	43
4.5.5 图形状态 (The Graphics State)	43

4.6 光栅化	43
第五章 True Type 字体引擎的实现.....	46
5.1 外部设计	47
5.2 内部设计	47
5.3 GRIDFITTING 模块.....	48
5.3.1 指令化 TRUETYPE 字体的有关概念.....	49
5.3.1.1 扫描转换模式与控制舍入.....	49
5.3.1.2 点的概念及其管理.....	49
5.3.2 距离的确定与管理.....	50
5.3.2.1 距离的确定.....	50
5.3.2.2 距离的管理.....	51
5.3.3 使用 CUTIN 控制.....	51
5.3.4 设计 TrueType 指令解释器.....	52
5.4 字符光栅化	56
5.4.1 多边形扫描线填充算法基础.....	56
5.4.1.1 顶点处理.....	56
5.4.1.2 连贯性的运用.....	57
5.4.2 贝塞尔曲线.....	58
5.4.2.1 中间点坐标的计算.....	58
5.4.2.2 贝塞尔曲线细分算法.....	59
5.4.3 轮廓点内外点的判断.....	61
5.4.3.1 奇偶判断法.....	61
5.4.3.2 线圈数 (winding number) 法.....	62
5.4.3.3 两种判断方法的比较.....	62
5.4.4 Raster 化模块的实现	63
5.4.4.1 第一次“虚拟描画”	67
5.4.4.2 第二次“虚拟描画”	69
5.4.4.3 填充与像素丢失控制模式.....	71
第六章 结论	75
致 谢	76
参考文献	77
攻硕期间取得的研究成果	79

第一章 引言

1.1 课题的来源及目的

本课题来源于日本某大型电子企业研究所与作者实习所在的日本 EIT 株式会社合作的一个研究项目。该项目的目的是：在不调用微软字体引擎的情况下，以某型号打印机驱动程序作为平台，设计实现一个从解析 TrueType Font 文件开始到生成字符 Bitmap 位图的 TrueType Font 引擎。

1.2 课题的意义

计算机字形信息处理技术是研究在计算机中描述、储存和以不同形式输出(即生成还原)字形符号形状的一个专门的研究领域。从上世纪七十年代开始，计算机突破过去的单纯以科学计算和数据处理的“计算”应用，进展到包括事务处理、文字处理等多方面的应用，各种文字在计算机中的存储问题即被提了出来，数字化的字形表示在计算机应用发展中发挥了重要作用^[1]。伴随光栅设备的出现和发展，作为计算机字形信息处理技术重要内容之一的计算机字形输出还原技术一直受到人们的重视^{[2][3]}。

由于 TrueType 字体的广泛应用，将由 TrueType 字体所描述的字符转化为相应 Bitmap 位图的 TrueType 字体引擎就成为了字体引擎家族中重要的成员，同时也是打印机驱动程序中的重要部分。所以对 TrueType 字体引擎的研究和实现有着比较重要的现实意义。同时由于本文所介绍的 TrueType 字体引擎是独立于微软的字体引擎的，所以就使它有着更为灵活和广泛的应用空间^[4]。

1.3 文献综述

TrueType 是由 Apple 公司和 Microsoft 公司联合提出的一种数学字形描述技术。它用数学函数描述字体轮廓外形，含有字形构造、颜色填充、数字描述函数、流程条件控制、栅格处理控制、附加提示控制等指令。TrueType 采用几何学中二次 B 样条曲线及直线来描述字体的外形轮廓，其特点是：TrueType 既可以作打印字体，又可以用作屏幕显示；由于它是由指令对字形进行描述，因此它与分辨率无关，输出时总是按照打印机的分辨率输出。无论放大或缩小，字符总是光滑的，不会

有锯齿出现。但相对 PostScript 字体来说，其质量要差一些。特别是在文字太小时，就表现得不是很清楚^{[5][6]}。

TrueType 字体，中文名称全真字体。它具有如下优势：

1. 真正的所见即所得字体。由于 TrueType 字体支持几乎所有输出设备，因而无论在屏幕、激光打印机、激光照排机上，还是在彩色喷墨打印机上，均能以设备的分辨率输出，因而输出很光滑。
2. 支持字体嵌入技术。存盘时可将文件中使用的所有 TrueType 字体采用嵌入方式一并存入文件之中，使整个文件中所有字体可方便地传递到其它计算机中使用。嵌入技术可保证未安装相应字体的计算机能以原格式使用原字体打印。
3. 操作系统的兼容性。MAC 和 PC 机均支持 TrueType 字体，都可以在同名软件中直接打开应用文件而不需要替换字体^{[7][8]}。

目前国外的几家大型的打印机厂商如 HP 和 Canon 都有自己的字体引擎技术。而且这些技术都是属于保密范围，很多技术细节对外界并不公开。由于国内还没有能够自行研制生产大型打印机的厂家，所以这方面的技术还不是很成熟。

1.4 本文内容安排

本文围绕 TrueType 字体引擎的有关问题，在以某型号打印机驱动程序的基础上设计和实现了一种独立于微软字体引擎，可用于打印机驱动程序的 TrueType 字体引擎。本文对内容做如下安排：

第二章，介绍 TrueType 字体技术的基本内容，分析研究 TrueType 字体文件结构，并以 Arial 字体作为例子，详细论述对于字体轮廓描述较为重要的 glyf 表和 camp 表；第三章，作为研究准备，研究分析了各种坐标空间，及其之间的关系；第四章，研究分析了 TrueType 字体引擎的基本原理和基本组成；第五章，设计并实现了一种独立于微软的字体引擎的 TrueType 字体引擎，并重点论述了 Raster 模块的实现；第六章，对课题的研究工作进行总结。


第二章 TrueType 字体技术

2.1 Font 是什么

所谓的 Font, 可以认为就是一种文字 Code 列到一幅显示出来的图像的对应关系。例如, 打开记事本, 写入几个字符, 然后用二进制编辑软件打开后, 里面只有一些字符的 Code。这些 Code, 通过选择不同的 Font, 显示不同的图像, 这种对应就是 Font。实际上 Font 这个词经常被混用于关联的很多地方。因此很多时候也没必要抠的太仔细。和 Font 相关的还有很多基础概念, 下面将一一讲述。

2.1.1 Symbol Set 和 Typeface

一般来说, 我们用“字”这个概念本身就存在很多不确定性和经验性。比如说, “A”这个字, 不管是写成 **A** 还是 *A* 我们都会认为这是同一个字, 不是“B”。虽然从理论上这种“认为”没有任何严格的定义, 但是作为一种常识和默认的公理, 我们规定了一组 Code 到一种形状的对应, 这可以看作是 Symbol Set。

可以大致认为, Symbol Set 决定了文字的图样的基础, 而 Typeface 决定了细节。例如, Arial Typeface + Roman 8 Symbol Set 下, Code 0x41 打出来是“A”这个样子。其中, Roman 8 Symbol Set 决定了内容是 **A** 而不是其他什么图样, 比如 WingDing 之类的 。而 Arial Typeface 决定了这个“A”是写成了 **A** 这个样子, 而不是 *A* (这是 Courier New)。

Symbol Set 和 Typeface 合起来就是 Font。

实际上, 不同的 Symbol Set 中, 如果一样的 Code 对应的是同样的“字”的话, 那么在同一种 Typeface 下图样是一致的。例如, 对于大多数 Symbol Set 来说, 0x41 到 0x5A 的 Code 对应的都是大写字母 A 到 Z, 那么这些 Symbol Set 加同一个 Typeface 得到的 Font 的 0x41 到 0x5A 的图样都会是一样的。也因此, Typeface 常常和 Font 的概念混用。

Font 中, 将 Typeface 和某一种 Symbol Set 绑定的称为 Bound Font; 不将 Typeface 和 Symbol Set 绑定的称为 Unbound Font。TrueType 字体都是 Unbound Font, Bitmap Font 都是 Bound Font。仔细想想, 会发现这个是很自然的。

相近的 Typeface 常常组成了一个 Typeface Family。比如说

Arial Arial Bold
Arial Italic Arial Bold Italic

都是 Arial 这个 Typeface Family 的。诸如

Italic Bold

这样的修饰，当系统中没有对应的 Font 时，常常采用“软”修饰的方法来代替。

2.1.2 Spacing

决定每一个字符所占的 Box 的宽度是固定的还是变动的。比如：

COURIER ARIAL

可以从“I”所占的宽度看出，Courier Font 的“C”也好“I”也好，其图样所占的 Box 宽度是固定的，称为 Fixed Spacing Font；而 Arial Font 的“A”明显和“I”的图样所占的 Box 宽度不一样，这称为 Proportional Spacing Font。

2.1.3 Pitch

只对“Fixed Spacing Font”有效，决定 Box 的宽度。此时也相应地决定了字符图样的大小。

2.1.4 Height

决定字符的高度，从而也决定了大小。对于 Fixed Spacing Font，因为只要 Spacing 决定了大小也就相应决定，所以当同时设定 Fixed Spacing Font 的 Pitch 和 Height 属性时，Height 属性被忽略掉只有 Pitch 属性来决定字符的大小。

2.1.5 Style

包括下面的几种修饰

Upright/Italic,
Condensed/Normal/Expanded,
Solid/Outline/Shadow

上述的效果都是在 WORD 中选择 Font 得到的效果^[12]。

2.1.6 Stroke Weight

决定字符的线的粗细，比如

Medium/Bold

2.2 什么是 TrueType 字体

2.2.1 概念

TrueType 是一种 Font 的技术。由 Apple 开发完成，并借助 Windows 和 MacOS 等 PC 操作系统的推广，最终成为了目前应用的最广泛的 Font 技术。TrueType 这个词也经常被混用，也没必要抠的太细。下面提到 TrueType 的时候，一般就是指 TrueType Font 技术。

2.2.2 Bitmap Font 和 Scalable Font

最早的 Font 都是 Bitmap 的点阵式，也就是一个 Code 对应了一个图样点阵，称为 Pattern。这种 Font 的优点是简单。不过随着实际应用的需求，要求能够实现文字进行放缩，旋转，扭曲等处理。显然 Bitmap Font，根本无法适应这种需求。

为了能够实现文字无损的放缩等功能，基于具体字符 Pattern 的 Font 技术必须被放弃，转而由描述字符轮廓(Outline)的 Font 技术所代替。这就是 Outline Font 或者说 Scalable Font。

Scalable Font 核心的思想是用一系列的点来标志字符的轮廓。这样点的坐标可以做任何放缩，旋转，扭曲等处理，然后再用这些点构造出变形后的字符的轮廓。TrueType Font 就是一种 Scalable Font。

早期很多公司都开发出了各式各样的 Scalable Font，但就目前而言，Windows 平台下是 TrueType Font 一统天下，同时有少数 Bitmap Font 被保留下来用于对老程序兼容。

2.2.3 Glyph

TrueType Font 中最小的描画单位称为 Glyph。Glyph 并不等于字，尽管一个字一定是一个 Glyph。Glyph 有 2 种，简单 Glyph 和复合 Glyph。简单 Glyph 直接包含的就是轮廓等描画信息，而复合 Glyph 则包括了多个简单 Glyph 的 ID，以及它们如何组合的操作信息^[8]。

2.2.4 TrueType 字体的历史

上世纪 80 年代，Adobe 的 Font 技术是最先进的。当时 Adobe 寻找 Apple 和 Microsoft，试图让他们在各自最新的操作系统中加入对 Adobe 提出的 Font 格式的支持。尽管 Adobe 的技术不管是屏幕显示，还是打印机输出都能得到当时最好的效果，但是 Apple 和 Microsoft 考虑到 Font 技术上受制于其它公司的后果，以及因此而必须付出的大笔版权金，基于长远打算就都拒绝了 Adobe 的提案。对于 Apple 来说，其另外一个目的是垄断 Apple 电脑的打印机市场

但是 Adobe 的技术优势是明显的，所以 Apple 和 Microsoft 合作，试图开发一种能够取代 PostScript 的格式的字体。当时的分工是，Microsoft 负责图像引擎，模块命名 TrueImage，Apple 负责字体引擎，模块命名 TrueType。结果现在大家都知道了，TrueType 变成了 MacOS/Windows 的标准字体技术，而 TrueImage 这个定位不明确的东西后来迅速走入了垃圾堆，或许今天的 GDI/DDI 里面还有它的影子？

1987 年到 1989 年，Apple 的总工程师 Sampo Kaasila 带领一个团队完成了 TrueType 的几乎所有工作。1991 年 3 月，Apple 正式发布 TrueType 技术，给 Macintosh System 6.0 发布升级包以支持 TrueType，同时推出三种 TrueType 字体，分别是 Times Roman，Helvetica，Courier。1992 年 Microsoft 发布了 Windows 3.1，Windows 平台也开始支持 TrueType，同时推出的三种 TrueType 字体，分别是 Time New Roman，Arial，Courier。时至今日，Windows 里面依然还有这三种字体^[15]。

2.2.5 TrueType 字体的优点

在当时看来，TrueType 字体技术确实超越包括 Adobe 在内的其他厂商的字体技术。

首先，它在绝大多数情况下显示的字型更加美观，特别是小字体低分辨率时更是优势明显。这是因为对于 Scalable Font 来说，最大问题在于放缩等数学处理之后，当连续的点和线要在离散的实际光栅设备上输出时，必须做舍入。这个时候，很多数学上正确的结果实际看上去非常别扭，丧失了美感，尤其是小字体。因此，任何向量型字体，在放缩之后，都需要经过一定的调整才能输出到实际的光栅设备如显示器，打印机等，这种调整称为指令化（Gridfitting 或 Hinting）。TrueType 对于指令化支持得非常出色，允许每一个独立的 Glyph 拥有独自の指令化算法，而这部分被一种类似汇编的语言所书写，并被嵌入在 Glyph 的数据中。

而且，TrueType 字体的轮廓采用的是直线和 2 阶 Bezier 曲线结合的描画方式，这几乎是最快的曲线计算方法。因此 TrueType 字体的性能表现也让人满意。

最后，它是一种完全开放的格式。所谓的开放，不仅仅表现在它格式的公开，而且在于 TrueType 字体的文件存储格式采用的是表格嵌套的思路，任何厂商都可以自由的添加一些自己独特的表来完成特殊的功能，只要解释程序能认识就可以了。作为 TrueType 字体可扩展性的最好例子，就是 Microsoft 的 OpenType。OpenType 实际上就是在原始的 TrueType 基础上加入了一些特别的表，例如给某些 Font 内嵌 Bitmap 点阵等等。

2.3 True Type 字体文件解析

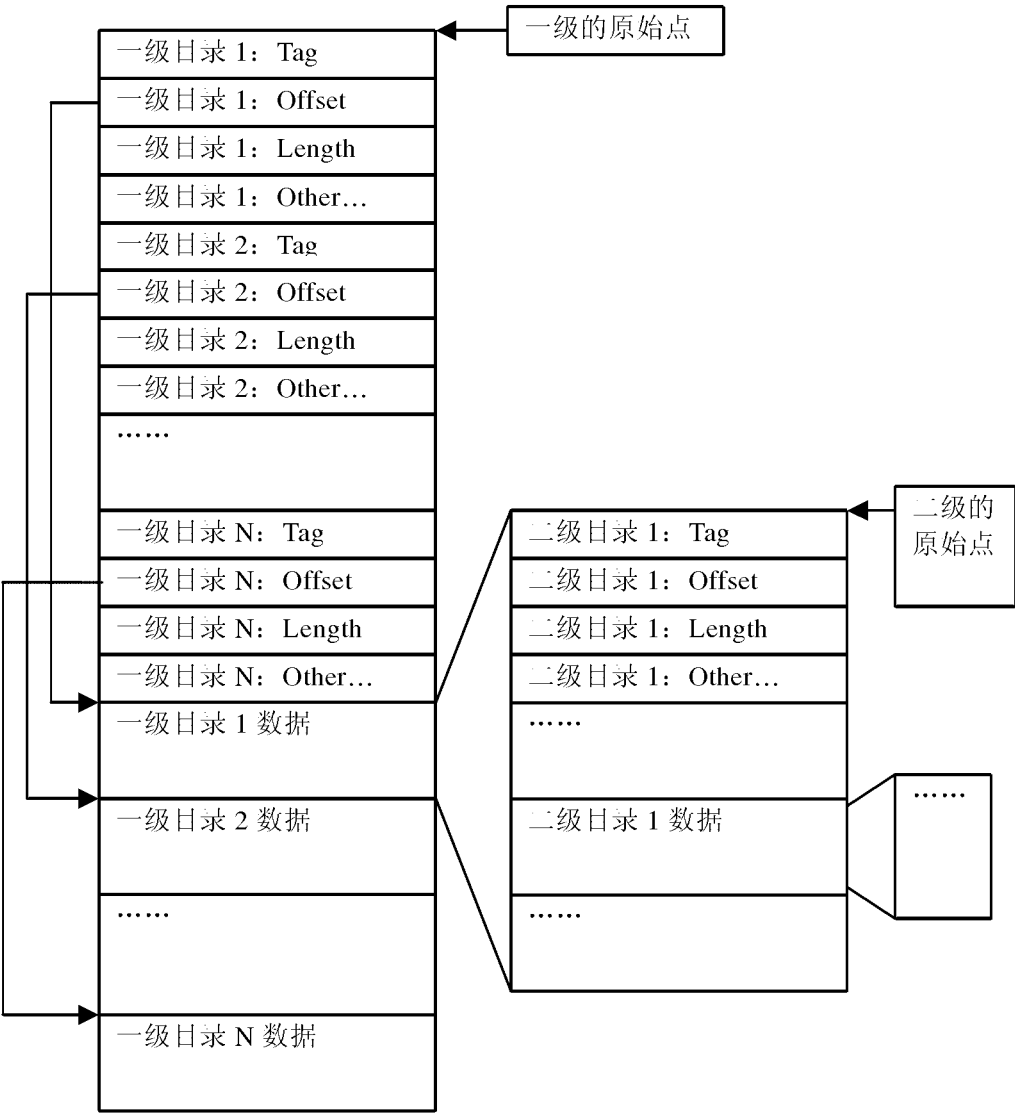


图 2-1 TrueType Font 文件结构

2.3.1 基本结构

TrueType 字体文件的结构如图 2-1 所示是表格目录式。如此嵌套形势组成了整个文件。其中需要注意的是，作为 Offset，都是只相对于当前级别的开头的。换句话说在一级目录中的寻址都是相对于一级目录的原始点(一般是文件的最先端)，再加上需要的 Offset 得到；而在二级目录中寻址的时候，原始点就是二级目录的起始端。

这样做的好处是，各个目录里面的内容是相对独立的，不管是目录前后移动，还是从中抽出一部分单独处理，都不需要牵扯到太多的修改。

2.3.2 数据类型

标准数据类型有：

BYTE	unsigned char;	CHAR	signed char
USHORT	unsigned short;	SHORT	signed short
ULONG	unsigned long;	LONG	signed long

扩展数据类型：

FIXED	16.16 型有符号定点小数，四则运算基本可以采用整数运算
F2DOT14	2.14 型有符号定点小数，四则运算稍微比 FIXED 复杂一点

特殊数据类型：LongDataTime 64Bit^[7]

2.3.3 文件内容简介

为了简单起见，以 Arial.ttf 作为例子。

2.3.3.1 TrueType 字体文件的开头

TrueType Font 文件开头是如表 2-1 所示结构：

表 2-1 TrueType Font 文件头

Type	Name	Description
Fixed	sfnt version	0x00010000 for version 1.0.
USHORT	NumTables	Number of tables.
USHORT	searchRange	(Maximum power of 2 numTables) x 16.
USHORT	entrySelector	Log2(maximum power of 2 numTables).
USHORT	RangeShift	NumTables x 16-searchRange.

目前 sfnt version 是固定值都是 1.0，numTable 标志了这个 TrueType 文件有多少个

表。其余的表项在快速定位表的时候有可能会用到。

对于 Arial 字体，该结构 dump 出来的内容是：

0x0-0xb 部分： 00 01 00 00 00 17 01 00 00 04 00 70

其中最重要的是 numTable = 0x17，即 Arial.ttf 共有 0x17，23 个表。

2.3.3.2 第一级目录

在 TrueType Font 文件开头 12 个 Bytes 之后的数据内容包含的是第一级目录的信息，个数由 numTable 的值决定。描述第一级目录的结构如下表 2-2：

表 2-2 第一级目录结构

Type	Name	Description
ULONG	Tag	4 -byte identifier.
ULONG	Checksum	CheckSum for this table.
ULONG	Offset	Offset from beginning of TrueType font file.
ULONG	Length	Length of this table.

Tag 是表的名称，下文常提到的某某表，就是指的这个 Tag。对于 Arial 字体，共有 23 个表。下面以 Arial 字体为例子简单介绍一些表的含义。

(1) glyf 表：0xcc-0xdb 部分： 67 6C 79 66 0E F7 8F EC 00 01 1A FC 00 03 E7 62

“67 6C 79 66”对应的 ASCII 码是“glyf”，因此该表就叫做 glyf 表，包含了所有 Glyph 的数据，位置从整个文件的 0x11AFC 处开始，共 0x3E762 个 Bytes。

(2) cmap 表：0x8c-0x9b 部分： 63 6D 61 70 E7 40 6A 3A 00 00 D1 C4 00 00 17 6A

“63 6D 61 70”对应的 ASCII 码是“cmap”，因此该表就叫做 cmap 表，位置从整个文件的 0xD1C4 处开始，共 0x176A 个 Bytes。cmap，意思是“Character To Glyph Index Mapping Table”。有了这个表，才能在得到字符的编码后知道去寻找对应的哪一个 Glyph。

(3) cvt 表：0x9c-0xab 部分： 63 76 74 20 96 2A D2 76 00 00 FA A0 00 00 06 30

“63 76 74 20”对应的 ASCII 码是“cvt”，因此该表就叫做 cvt 表，位置从整个文件的 0xFAA0 处开始，共 0x630 个 Bytes。cvt，意思是“Control Value Table”，保存了指令化中需要的一些数据。

(4) 其它表

其余的表还有“head”，“maxp”，“fpgm”，“prep”，“hhea”，“hmtx”，“vhea”，

“vmtx”等等。其中“head”给出了字体的基本信息，“maxp”给出了字体中一些数组变量的最大长度(可以方便分配空间)，“fpgm”和“prep”都是指令化相关的表，“hhea”“hmtx”“vhea”“vmtx”这四个表给出了字体在横写和竖写时的一些特征。基本上可以说，最重要的表就是“head”，“maxp”，“cmap”，“cvt”，“glyf”这五个。下面将对“glyf表”和“cmap表”进行详细阐述。

2.3.4 glyf 表详解

如前面所提到的，glyf 表包含了字体文件中所有字符的数据。glyf 表由简单 glyph 子表和复合 glyph 子表组成。

Glyph 子表的个数可以从 maxp 表中读取，maxp 表中的第 5,6 个字节就是 glyph 子表的个数 numGlyphs，它是一个 USHORT 型数据。

那么如何定位每个 glyph 子表的位置呢？

其实每个 glyph 子表相对于 glyf 表的偏移量在表 loca 中可以读取。这些位置索引以 ULONG offsets[numGlyphs+1]或是 USHORT offsets[numGlyphs+1]的数组形式存储在 loca 表中。为了确定 offsets[]数组的数据类型，必须先从 head 表中读取 indexToLocFormat，它位于 head 表中的第 51 和 52 个字节，是个 SHORT 型数据。如果 indexToLocFormat 为 1，offsets[]为 ULONG 型数组；如果 indexToLocFormat 为 0，offsets[]为 USHORT 型数组。Offsets[]数组的最后一个额外的偏移量并不实际指向一个 glyph 子表的位置偏移，而是为了计算最后一个 glyph 子表的长度而设的，它指向最后一个 glyph 子表末的下一个位置。每个 glyph 子表的长度等于 offsets[i+1] - offsets[i]。

这里需要说明的是，位置索引表中的每一个索引以无符号短整数对齐的，如果使用了短整数格式，索引表实际存储的是 WORD 偏移量，而不是 BYTE 偏移量。所以，当使用了短整数格式时，实际偏移量是所读取的数据和 2 乘积。

在得到每个 glyph 子表的位置偏移量（相对于 glyf 表），就可以很快地定位每个子表。一般，每个 TrueType 文件的第一个 glyph 子表是 MISSING CHARACTER GLYPH，其意义是当某一字符在 glyph 表中没有找到相应的 glyph 数据时，就使用第一个 glyph 子表的数据。第一个子表的位置偏移量是 0。

2.3.4.1 Glyph 描述部分

Glyph description 存在于所有的 glyph 子表（除了长度为 0 的子表）表的开头

部分，它包括 10 个字节，其数据结构如表 2-3：

表 2-3 glyph 表头

Type	Name	Description
SHORT	numberOfContours	If the number of contours is greater than or equal to 0 this is a single glyph; if negative, this is a composite glyph.
FWord	xMin	Minimum x for coordinate data.
FWord	yMin	Minimum y for coordinate data.
FWord	xMax	Maximum x for coordinate data.
FWord	yMax	Maximum y for coordinate data.

X, Y 的最小值和最大值限制了当前子表中坐标数据的取值范围，而且定义了当前 glyph 的 bounding box。bounding box 用于在发生任何指令化操作前的原始 glyph 轮廓^[15]。

紧跟在 glyph 描述部分后面的是当前子表的简单 glyph 数据或复合 glyph 数据。

2.3.4.2 简单 glyph

头结构之后就是 Glyph 存储的数据，其数据结构如表 2-4：

表 2-4 简单 glyph 表头

Type	Name	Description
USHORT	endPtsOfContours[<i>n</i>]]	Array of last points of each contour; <i>n</i> is the number of contours.
USHORT	instructionLength	Total number of bytes for instructions.
BYTE	instructions[<i>n</i>]	Array of instructions for each glyph; <i>n</i> is the number of instructions.
BYTE	flags[<i>n</i>]	Array of flags for each coordinate in outline; <i>n</i> is the number of flags.
BYTE or SHORT	xCoordinates[]	First coordinates relative to (0,0); others are relative to previous point.
BYTE or SHORT	yCoordinates[]	First coordinates relative to (0,0); others are relative to previous point.

根据 `endPtsOfContours[n]` 可以得出 `glyph` 的点数：`endPtsOfContours[n-1]+1`。简单 `glyph` 中关键是分析 `flags[n]` 数组，点的 X, Y 坐标是结合与其相对应的 `flags` 值的解析来得到正确值的。理论上，`flags[]`，`xCoordinates[]`，`yCoordinates[]` 数组的个数应该是和 `glyph` 点的个数是相等的，但在实际的字体文件中，为了节省空间，采用了一定的压缩方法，所以上述数组的个数不一定和 `glyph` 点的个数是相等。在分析处理 `glyph` 时，我们可以分配和 `glyph` 点的个数相等的 `Tempflags[]`，`TempxCoordinates[]`，`TempyCoordinates[]` 数组，把解析出来的 `flags`，`xCoordinates`，`yCoordinates` 和每一个点对应起来，这样处理起来较为方便。下面用 Arial 字体的“1”来具体说明。这部分的数据位于 Arial.ttf 文件中的 0x12a0e-0x12ae9 处，长度为 220Bytes。将这部分数据完全 Dump 出来就是：

```
0x12a0e: 00 01 00 df 00 00 02 fb 05 c0 00 0a 00 af 40 20
0x12a1e: 03 40 0d 11 34 6b 04 7f 02 8f 02 99 08 04 ac 04
0x12a2e: 01 09 00 06 05 02 03 09 05 01 0c 02 01 ca 0a 00
0x12a3e: b8 ff c0 40 0a 21 23 34 30 00 01 20 00 01 00 b8
0x12a4e: ff e0 b4 10 10 02 55 00 b8 ff ea 40 11 0f 0f 02
0x12a5e: 55 00 1c 0c 0c 02 55 00 0e 0d 0d 02 55 00 b8 ff
0x12a6e: f0 40 19 0f 0f 06 55 00 10 0c 0c 06 55 00 10 0d
0x12a7e: 0d 06 55 00 1a 0c 05 40 0d 0f 34 05 b8 ff c0 40
0x12a8e: 0e 21 23 34 30 05 01 20 05 40 05 02 05 19 0b ba
0x12a9e: 01 3c 01 85 00 18 2b 4e 10 e4 5d 71 2b 2b 10 f6
0x12aae: 2b 2b 2b 2b 2b 2b 2b 5d 71 2b 3c 4d fd 3c 00 3f
0x12abe: 3f 17 39 01 11 39 31 30 01 5d 00 5d 2b
0x12ace:                02 fb b4 41 d3 54 97 e2
0x12ade: 2f 74 04 7b 3e 7c 1f ae 47 ca 5f 00
```

表 2-3 中已经说的很清楚了，`numberOfContours` 表示了轮廓的数目。在这个例子中，Arial 的 1 是由一条轮廓组成的，这里 `numberOfContours` 的值就是 1；如果是 A，那么 `numberOfContours` 的值就是 2；等等。看 0x12a0e 开始的 10 个 Byte，就是表 4.3 这个头结构。这里只看 `numberOfContours`，其余的量暂时不考虑。当 `numberOfContours` 小于 0 的时候，表示是一个复合 `glyph`，稍后会讲到。先看这个简单 `glyph`。这里有一些东西解释的比较绕。

首先是一个 `USHORT` 数组 `endPtsOfContours`，表示的是各个轮廓的终点。由于起点可以从前一个轮廓的终点加 1 得到，而第一条轮廓的起点是 0，因此只需要

存储终点就可以了。比如说，这里第 1 条轮廓，数值是蓝色的 00 0a，那么意思就是从点 0 开始到点 0x0a 为止是轮廓 1 的点。同样的，如果有 2 条轮廓，接在头文件之后的数据是 00 03 00 07 的话，那么意思就是从点 0 开始到点 3 是轮廓 1，从点 4 开始到点 7 是轮廓 2。这里还隐含了点的数目信息。最后一条轮廓的终点加上 1 就是描述全部轮廓的点的数目。在本例中是 0x0b(11)个点。

其次是 `instructionLength` 和 `instructions`，这个就是指令化的代码。前面一个 `Byte` 表示操作码的总长，后面则是具体的操作码，这里先跳过去。

再下来是 `flags` 数组，以及 `xCoord`，`yCoord` 数组。`TrueType` 字体文件中采用的是一种非常紧密的压缩方式，因此这里数组长度都是不定的^[15]。

先来看 `flags` 数组，这是一个标志数组，其 0 到 7Bit 表示的意义如表 2-5：

表 2-5 `flags` 数组

Flags	Bit	Description
On Curve	0	If set, the point is on the curve; otherwise, it is off the curve.
x-Short Vector	1	If set, the corresponding x-coordinate is 1 byte long, not 2.
y-Short Vector	2	If set, the corresponding y-coordinate is 1 byte long, not 2.
Repeat	3	If set, the next byte specifies the number of additional times this set of flags is to be repeated. In this way, the number of flags listed can be smaller than the number of points in a character.
This x is same (Positive x-Short Vector)	4	This flag has two meanings, depending on how the x-Short Vector flag is set. If x-Short Vector is set, this bit describes the sign of the value, with 1 equalling positive and 0 negative. If the x-Short Vector bit is not set and this bit is set, then the current x coordinate is the same as the previous x coordinate. If the x-Short Vector bit is not set and this bit is also not set, the current x-coordinate is a signed 16-bit delta vector.
This y is same (Positive y-Short Vector)	5	Fro y ,the meaning is the same as bit4
Reserved	6	This bit is reserved. Set it to zero.
Reserved	7	This bit is reserved. Set it to zero.

实际上，Flag 的数目应该等于所有点的数目，但是 Flag 中连续的相同 Flag 可以通

过 Bit3 位置设成 1 而压缩存储，因此 Flag 数组的长度有可能会小于点的数目。本例中没有出现重复的 Flag，因此 Flag 的长度等于点的数目 11，橘红色的数据部分。

Flag 数组之后是所有表示轮廓的点的横，纵坐标。其中，每一点的 X，Y 的值是根据 Flag 对应 Bit 的设定而得到的。所有的数值都是相对值，第 0 点相对于 0。比如说这里的横坐标，如果要把所有数值都解开的话，应该是表 2-6：

表 2-6 横坐标

	Flag	Bit1	Bit4	表示意义	参照 Data	最终数值
0	21	0	0	signed short 值	02 fb	0x02fb
1	23	1	0	1Byte 的负位移	b4	0x0247
2	11	0	1	重复 x		0x0247
3	06	1	0	1Byte 的负位移	41	0x0206
4	06	1	0	1Byte 的负位移	d3	0x0133
5	07	1	0	1Byte 的负位移	54	0x00df
6	35	0	1	重复 x		0x00df
7	36	1	1	1Byte 的正位移	97	0x0176
8	36	1	1	1Byte 的正位移	e2	0x0258
9	37	1	1	1Byte 的正位移	2f	0x0287
10	33	1	1	1Byte 的正位移	74	0x02fb

同样，可以顺次把纵坐标的 11 个点坐标值也都解开如表 2-7：

表 2-7 纵坐标

	Flag	Bit2	Bit5	表示意义	参照 Data	最终数值
0	21	0	1	重复 y		0x0000
1	23	0	1	重复 y		0x0000
2	11	0	0	signed short 值	04 7b	0x047b
3	06	1	0	1Byte 的负位移	3e	0x043d
4	06	1	0	1Byte 的负位移	7c	0x03c1
5	07	1	0	1Byte 的负位移	1f	0x03a2
6	35	1	1	1Byte 的正位移	ae	0x0450
7	36	1	1	1Byte 的正位移	47	0x0497
8	36	1	1	1Byte 的正位移	ca	0x0561

9	37	1	1	1Byte 的正位移	5f	0x05c0
10	33	0	1	重复 y		0x05c0

最后结束时，因为整个数据的长度是奇数，所以用 0 补齐。

求横坐标的相对坐标的伪代码如下：

```

if (bit1)    //该点的 X 相对坐标在 font 文件中占有 1 字节数据
{
    if (bit4)    //该点的 X 相对坐标为正值
        TempCoordinates[i] = xCoordinates[i];
    else    //该点的 X 相对坐标为负值
        TempCoordinates[i] = -xCoordinates[i];
} else    //该点的 X 相对坐标在 font 文件中占有 2 个字节数据
{
    if (bit4)    //该点的 X 绝对坐标和前一点的 X 绝对坐标相同，font 文件//
                中没有该点 X 相对坐标数据，它被压缩
        TempCoordinates[i] = 0;
    else    //该点的 X 相对坐标为 font 文件中的数据
        TempCoordinates[i] = xCoordinates[i];
}

```

求纵坐标相对坐标的方法同求横坐标的一致，纵坐标相对坐标由 bit2 和 bit5 决定。根据 flag，可以知道点 0, 1, 2, 5, 6, 9, 10 在轮廓上，点 3, 4, 7, 8 不在轮廓上。他们的坐标分别是：

(763, 0) (583, 0) (583, 1147) (518, 1085) (307, 961) (223, 930)
(223, 1104) (374, 1175) (600, 1377) (647, 1472) (763, 1472)

2.3.4.3 复合 glyph

复合 glyph 的描述部分 number Of Contours 的值为 负一，它是由两个或以上的简单 glyph 复合而成的。复合 glyph 的数据包括一系列的 component glyph part description 和一系列指令集。Component glyph part description 的个数由组成当前复合 glyph 的简单 glyph 个数决定，它没有明显地给出，而是隐含在 component glyph part description 的 flags 当中。Component glyph part description 的数据结构如表 2-8 所示。

表 2-8 复合 glyph 表头

Type	Name	Description
uint16	flags	Component flag
uint16	glyphIndex	Glyph index of component
int16, int16, int8 or uint8	argument1	X-offset for component or point number; type depends on bits 0 and 1 in component flags
int16, int16, int8 or uint8	argument2	Y-offset for component or point number type depends on bits 0 and 1 in component flags
transformation option		One of the transformation options

复合 glyph 用一个经变换的 glyph 序列定义。每个经变换的 glyph 的定义包括三个部分：一个标志、一个 glyph 索引和一个变换矩阵。标志字段决定了变换矩阵的编码方式。编码的目的也是为了节省一些空间，外加还说明了是否已到达序列的终点。

也举个例子，Arial 的“Š”。这是由“S”和“y”两个 Glyph 组成的，这部分数据位于 Arial.ttf 文件中的 0x1f670e-0x1f673d 处，长度为 48Bytes。将这部分数据完全 Dump 出来就是：

```
0x1f670e: ff ff 00 5c ff e7 04 eb 07 26          00 00
0x1f671e:          01 28 01 64 00 19 40 0c 01 f0 31 01
0x1f672e: 31 16 12 48 2b 01 01 34 b9 02 21 00 29 00 2b 01
0x1f673e: 2b 5d 35 00
```

首先开头 10Byte 和简单 Glyph 一样，也是 Glyph 的头。其中的 numberOfContours 这里是-1，表示是复合 Glyph。接下来 2 个 Byte 分别表示 Flag 和 GlyphIndex。其中 GlyphIndex 是指组成复合 glyph 的简单 glyph 的索引值，而非偏移量；argument1 和 argument2 有可能是 X 和 Y 的偏移量，也有可能是点的个数，它们的数据类型也不确定，这些都由 component flags 决定。Flag 的意义如表 2-9：

表 2-9 component flags 意义

Flags	Bit	Description
ARG_1_AND_2_ARE_WORDS	0	If this is set, the arguments are words; otherwise, they are bytes.
ARGS_ARE_XY_VALUES	1	If this is set, the arguments are xy values; otherwise, they are points.

ROUND_XY_TO_GRID	2	For the xy values if the preceding is true.
WE_HAVE_A_SCALE	3	This indicates that there is a simple scale for the component. Otherwise, scale = 1.0.
RESERVED	4	This bit is reserved. Set it to 0.
MORE_COMPONENTS	5	Indicates at least one more glyph after this one.
WE_HAVE_AN_X_AND_Y_SCALE	6	The x direction will use a different scale from the y direction.
WE_HAVE_A_TWO_BY_TWO	7	There is a 2 by 2 transformation that will be used to scale the component.
WE HAVE INSTRUCTIONS	8	Following the last component are instructions for the composite character.
USE_MY_METRICS	9	If set, this forces the aw and lsb (and rsb) for the composite to be equal to those from this original glyph. This works for hinted and unhinted characters.

对于本例来说，可以得到：Flag1 = 0x0226，GlyphIndex1 = 0x0036

看 Flag 的各个 Bit，可以得到表 2-10：

表 2-10 Flags 数组示例

Bit	值	意义	参照 Data	结果
0	0	Arg1 和 Arg2 都是 Byte 型	00 00	Arg1 = 0, Arg2 = 0
1	1	Arg1 和 Arg2 是 x 和 y 的位移		
2	1	对 Arg1 和 Arg2 相对网格取整		
3	0	非简单 Scale		
4	0	Reserved		
5	1	之后还有一个 Glyph		
6	0	没有 x 和 y 相异的 scale		
7	0	没有 2x2 的变换		

8	0	Bit5 = 1, 不考虑		
9	1	用复合 Glyph 的某些值代替原来的		

到此为止是复合 Glyph 中第一个 Glyph 的数据已经结束。紧跟的是第二个 Glyph。其 Flag2 = 0x0107, GlyphIndex2 = 0x00df, 同样的格式分析可以得到表 2-11:

表 2-11 Flags 数组示例

Bit	值	意义	参照 Data	结果
0	1	Arg1 和 Arg2 都是 Word 型	01 28 01 64	Arg1 = 0x0128, Arg2 = 0x0164,
1	1	Arg1 和 Arg2 是 x 和 y 的位移		
2	1	对 Arg1 和 Arg2 相对网格取整		
3	0	非简单 Scale		
4	0	Reserved		
5	0	之后已经没有 Glyph		
6	0	没有 x 和 y 相异的 scale		
7	0	没有 2x2 的变换		
8	1	有后继的 Instruction		
9	0	用 Glyph 原有的值		

之后 Transformation option 部分在 font 文件中可能不存在, 可能存在一个值或多个值, 这些值都以 SHORT 类型存在。它们可能的组合如表 2-12:

表 2-12 transformation option 部分

Transformation Option	Meaning
transformation entry #1	scale (same for x and y)
transformation entry #2	x-scale
	y-scale
transformation entry #3	xscale
	xscale
	scale01
	yscale

以上组合的条件为:

transformation entry #1: WE_HAVE _A_SCALE 被设为 1;

transformation entry #2: WE_HAVE_AN_X_AND_Y_SCALE 被设为 1;

transformation entry #3: WE_HAVE_A_TWO_BY_TWO 被设为 1。

Transformation option 的值都是 SHORT 型的,在实际应用中要将该值变为 float 型,如: $Xscale = xscale / (float)16384.0$

当 MORE_COMPONENTS 为 1 时,紧跟在当前的 Component glyph part description 后面是下一个 Component glyph part description。

当 WE_HAVE_INSTRUCTIONS 为 1 时,说明该复合 glyph 有指令,则紧跟在最后一个 Component glyph part description 后面的是指令长度 USHORT instructionLength 和指令内容 BYTE instructions[instructionLength]。至此,整个复合 glyph 结束。

对于本例来说由于所有 glyph 都已经结束,开始看指令化的代码。因为最后一个 glyph 中的 Flag 表示有后继的 Instructions,因此后面是这样的一个结构如表 2-13:

表 2-13 instructions

Type	Name	Description
USHORT	instructionLength	Total number of bytes for instructions.
BYTE	instructions[n]	Array of instructions for each glyph; <i>n</i> is the number of instructions.

这里 instructionLength = 0x0019, 后面的 25Bytes 就全部是 Instructions。从这里可以看出,这个复合的 glyph,实际上是由 0x0036 和 0x00df 两个 glyph 组合而成。其中 0x0036 不需要做位移,而 0x00df 则需要做一定的位移。此外组合成 glyph 之后还需要另外用指令化来调整。

2.3.5 Cmap 表详解

因为TTF字库本身有一个排列glyph的顺序,一般并不与系统代码相同,所以使用cmap表来作为系统内码到glyph序号的映射。cmap表由cmap表头、cmap子表描述目录和一系列子表组成。cmap表头长度为4个字节,内容如下:

USHORT cmap_version //cmap表版本号码

USHORT cmap_tables //子表的个数

接下来是cmap子表描述目录,共有cmap_tables个目录入口,每个目录项长度为8个字节,内容如下:

USHORT Platform_ID //平台标识

```

USHORT  Encoding_ID    //编码体系标识
ULONG   offset         //子表位置偏移

```

cmap表使得TTF字体文件可以在不同的平台和译码体系下使用，Platform_ID代码的值一般为3和1，分别代表Microsoft平台和Macintosh平台，它们使用不同的字符集和编码方法。Encoding_ID用来具体选择字符集和编码方法。每一个cmap子表由一组Platform_ID和Encoding_ID唯一确定，并按Platform_ID和Encoding_ID的顺序由小到大排列。如在Windows中，Platform_ID的值为3，日文的SHIFT-JIS、中文的GB-2312以及Unicode的Encoding_ID的值分别为3、2、1。

在子表描述目录后面是每个子表的详细描述。为了支持各种字符编码方案，cmap子表可以有多种描述方式，比较常见的有4种格式：

- 格式0：Apple公司的字符编码，256的索引表，这是最简单的西文索引结构。
- 格式2：双字节编码映射表，适合于东方文字8/16混合单双字节或双字节编码。
- 格式4：微软公司标准的字符映射表，也是西文最常用的索引结构，一般编码空间被分为几个连续的范围，又称为分段映射表。
- 格式6：格式4中段数为1时使用，又称整齐映射表^[15]。

在Windows下，cmap描述子表不止一个，但TTF解释器只访问第一个Platform_ID为3且格式为2或4的cmap描述子表。通常的字符映射方法如图2-2所示：

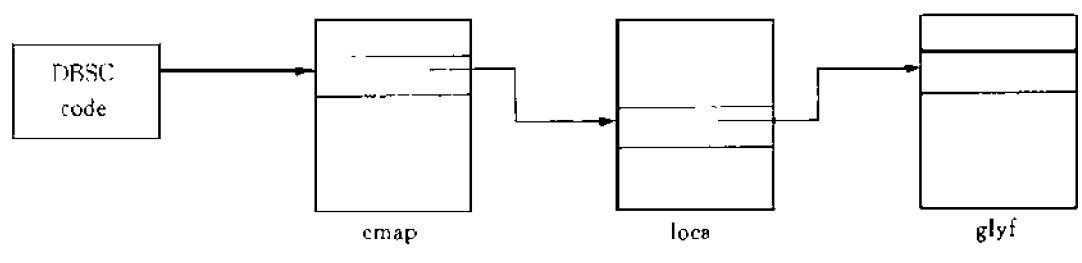


图2-2 字符映射方法

下面以格式4和Arial字体为例具体阐述。Arial字体中格式4的数据结构如表2-14所示：

表2-14 格式4数据结构

Type	Name	Description
USHORT	Format	格式号设为4
USHORT	Length	整个映射表的大小
USHORT	version	版本号
USHORT	segCountX2	段的个数的2倍

USHORT	searchRange	$2 \times (2^{\lceil \log_2(\text{segCount}) \rceil})$
USHORT	entrySelector	$\log_2(\text{searchRange}/2)$
USHORT	rangeShift	$2 \times \text{segCount} - \text{searchRange}$
USHORT	endCount[segCount]	每个段的结束字符编码，最后一个段的结束编码为0xffff
USHORT	reservedPad	设为0
USHORT	startCount[segCount]	每个段的起始字符编码
USHORT	idDelta[segCount]	该段内部所有字符的delta（其实是一种共同属性，数学上的delta？）
USHORT	idRangeOffset[segCount]	0或者是glyphIndex里面的偏移量
USHORT	glyphIdArray[]	glyph索引数组

查找glyph索引的算法如下：

1. 对于给定的字符编码 c ，确定所在的段。首先找到第一个大于或等于 c 的 $\text{endCount}[i]$ 。
2. 如果 $\text{idRangeOffset} == 0$ ，那么 $\text{glyphIndex} = (\text{idDelta}[i] + c) \% 65536$
3. 如果 $\text{idRangeOffset} != 0$ ，那么

$$\text{glyphIndex} = (* (\text{idRangeOffset}[i] / 2 + (c - \text{startCount}[i]) + \&\text{idRangeOffset}[i]) + \text{idDelta}[i]) \% 65536$$

以 Arail 为例，Arail.ttf 这种字体的 format 4 表里面共有 147 个段，这是其中的前十段：

```

Seg No =   1 : Beg = 0020, End = 007e, D =   65507, RO =       0
Seg No =   2 : Beg = 00a0, End = 017f, D =       0, RO =   292
Seg No =   3 : Beg = 0192, End = 0192, D =   65300, RO =       0
Seg No =   4 : Beg = 01a0, End = 01a1, D =    714, RO =       0
Seg No =   5 : Beg = 01af, End = 01b0, D =    701, RO =       0
Seg No =   6 : Beg = 01cd, End = 01dc, D =    815, RO =       0
Seg No =   7 : Beg = 01fa, End = 01ff, D =   65500, RO =       0
Seg No =   8 : Beg = 02c6, End = 02c7, D =       0, RO =   728
Seg No =   9 : Beg = 02c9, End = 02c9, D =   65039, RO =       0
Seg No =  10 : Beg = 02d8, End = 02dd, D =       0, RO =   728

```

如果我们要查找 0x41 所对应的 glyph 索引：

1. 首先发现 0x41 小于 0x7e, 那么该字符在第一个段里。
2. IdRangeOffset 等于 0, 所以 $\text{glyfindex} = (0x41 + 65507) \% 65536 = 36$ 。

如果查找 0xa1 所对应的 glyf 索引:

1. 首先发现 0xa1 小于 017f, 而且大于 0xa0, 可以确定在第二个段里。
2. IdRangeOffset 等于 292, 不等于 0, 那么我们先计算

$\text{temp} = \text{idRangeOffset}[i] / 2 + (c - \text{startCount}[i]) = 292 / 2 + 0xa1 - 0xa0 = 147$,

然后 $147 + \text{idRangeOffset}[i]$, 这时候我们会发现问题来了。因为该字体本身只有 147 个段, 指针 idRangeOffset 偏移 148 已经超出了范围, 按道理来说取该地址的数据应该属于非法操作, 不过由于这个数组后面紧接着的是 glyindex 数组, 因此这里面用了个小技巧, 取的正好是 glyindex 里面的数据, 然后加上 idDelta[i], 再 %65536 就可以得到正确的 glyf 表里面对应的字符了。

当用户使用TTF汉字时, 只需给出该汉字的内码, TTF解释器通过查找cmap表得到该汉字在loca表中的文字序号, 再从loca表中获得对应汉字轮廓数据的存放地址。有些双字节代码不对应任何汉字, 则cmap表将它们统统映射为loca表中的序号0, 称为丢失字符(missing character), 其对应轮廓数据往往解释出来是空心方框。

采用“cmap → loca”、“loca → glyf”两级映射机制, 是TrueType技术的特色之一。它为不同应用对TTF文件的修改提供了操作独立性和灵活性。其特性如下:

1、由于cmap表是系统内码到TTF文件glyph序号的映射, 所以只要能相应设置多张cmap描述子表就可可在不同平台, 不同字符集下使用同一套TTF字形数据。

2、由于glyph序号到实际字形数据的映射是由loca表完成的, 所以对于字形数据的修改不致影响cmap表, 即cmap表一般可由平台标识、字符集标识和描述格式来确定, 与字形数据的大小和排列次序无关。由于cmap表的组织涉及到较多的计算, 因此它的相对固定就减少了制作字库或修改字形数据时的工作量^[18]。

当用户使用TTF汉字时, 只需给出该汉字的内码, TTF解释器通过查找cmap表得到该汉字在loca表中的文字序号, 再从loca表中获得对应汉字轮廓数据的存放地址。有些双字节代码不对应任何汉字, 则cmap表将它们统统映射为loca表中的序号0, 称为丢失字符(missing character), 其对应轮廓数据往往解释出来是空心方框。

第三章 坐标空间

坐标空间定义了所有描画所发生的场所。它决定了出现在页面上的文字、图形、图像的位置、方向和大小。这一章将讨论几种常见的坐标空间亦即它们之间的相互关系。

3.1 各种坐标空间

在笛卡尔平面中轨迹和位置是以一系列成对的坐标对定义的。一个坐标对是一对实数 (x,y) ，它确定了一个点在一个二维坐标空间里的横向和纵向的位置。对于某一确定的页面，它的坐标空间由以下三要素决定：

- ① 坐标原点的位置；
- ② 坐标轴 X 轴和 Y 轴的方向；
- ③ 每条坐标轴的长度。

3.1.1 设备空间 Device Space

页面上的内容最终要显示在一个光栅输出设备上，例如：显示器或者打印机。这些设备在它们的显示区域内是以像素为基础建立它们内建的坐标系统的。一个特定设备的坐标系统被称作设备空间。设备空间的坐标原点在不同的设备上可能不同。对于显示器它的坐标原点在很大程度上依赖于它的视窗系统。因为纸张或其它输出介质在不同打印机或成象系统中可能会以不同方向移动，它们的设备空间的坐标轴的方向可能也不同。比如，在一些设备上竖直方向的坐标值是从页面的顶端向底端增加的，而在有的设备上则是从底端向顶端增加的。而且不同设备之间的分辨率也有不同，有的设备甚至在垂直方向和水平方向上的分辨率也不同。

如果在 TrueType 字体引擎中以设备空间建立坐标，那么所解析的文件就必须是和设备相关的，而且同一文件在不同的设备上显示的结果也会不一样，这显然是不行的。例如：对于同一个图象分别在典型的 72DPI（每英寸 72 个像素）的显示器和 300DPI（每英寸 300 个点）的打印机上显示的结果在大小上的差别将在 8 倍以上；一条在显示器上是 8 英寸的线段在打印机上显示的长度要比 1 英寸还短。图 3-1 显示了，由设备空间指定的同一图形对象，在不同输出设备上光栅化时，所

表现出来的巨大差异。

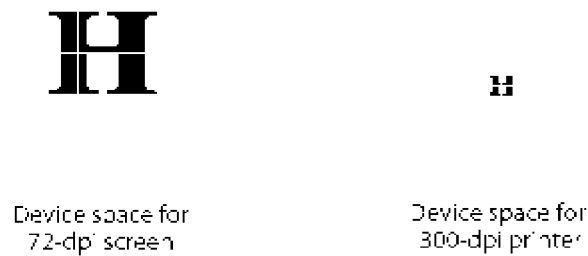


图 3-1 设备空间

3.1.2 用户空间 User Space

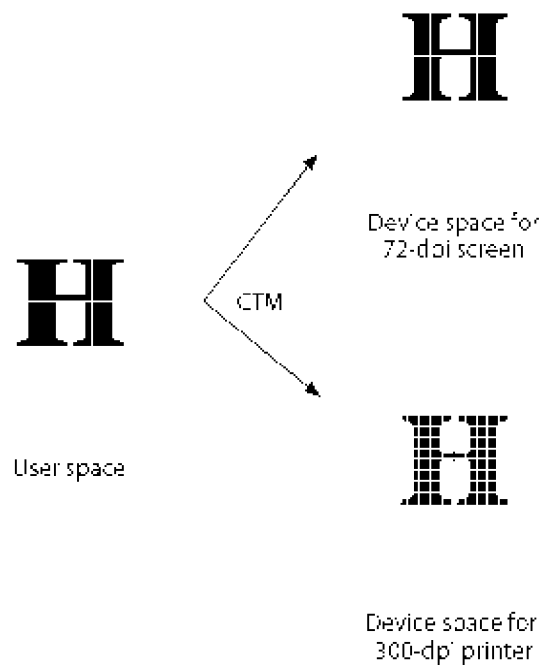


图 3-2 用户空间

为了避免在设备空间中指定对象所造成的设备相关的影响，我们定义了一种独立于输出设备的坐标系统。对于当前处理的页面，无论是在输出设备上打印还是显示，这种坐标系统认为分辨率都是相同的。这种独立于设备的坐标系统被称作用户空间。

用户空间的坐标系统为文件的每一页都初始化为一个默认的状态。和标准的数学坐标一样，这个默认状态是 X 轴的正方向为沿水平方向向右，Y 轴的正方向为沿竖直方向向上。从理论上讲，用户空间是一张无限延伸的平面，而输出设备的显示区域只对应了其很小的一部分。对于用于显示或打印的每个页面在用户空间中的默认显示区域可以是不同的^[19]。

从用户空间向设备空间的转化是通过 CTM 矩阵（Current Transformation Matrix）来实现的。应用程序可以根据特定输出设备的分辨率，通过调整 CTM 矩阵，来获得独立于设备的页面描述。如图 3-2 所示，一个由用户空间定义的对象，无论光栅化它的输出设备的分辨率是多少，最后所得到的大小都是相同的。

3.1.3 文本空间 Text space

文本的坐标定义在文本空间中。文本空间向用户空间的转换，仍然是通过相应的矩阵运算来实现的。文本空间的主要目的是为了将连续的多个相同字体的文字放在同一个坐标系统下来研究它们的相对位置。

3.1.4 轮廓空间 Glyph space

字体中字符的轮廓定义在轮廓空间中。轮廓空间完全是针对单个字符而言的，它是将一个字符的轮廓看成一个二维的几何图形，来研究它各个部分的位置关系和几何外形。在轮廓空间中形成字符轮廓后，在将该轮廓放在文本空间下，轮廓空间向文本空间的转换，仍然是通过相应的矩阵运算来实现的。本文所讨论的 TrueType 字体引擎将主要在轮廓空间下讨论问题。

3.1.5 其他空间

图形空间 Image space: 针对简单图形设计的一种坐标系统。

模式空间 Pattern space: 针对多重颜色描绘的复杂图形设计的一种坐标系统。

3.1.6 各个坐标空间之间的转化

图 3-3 表示了以上几个坐标空间之间转化的关系。每个箭头表示了从一个坐标空间向另一个坐标空间的转化。

由于某个对象在一个坐标空间中的定义和在另一个空间中的表示有关，所以改变一个变换矩阵会影响对象在其他坐标空间中的表示。比如：改变从用户空间

到设备空间的 CTM 矩阵，就会影响到对象在文本空间、图形空间、模式空间中的表示。

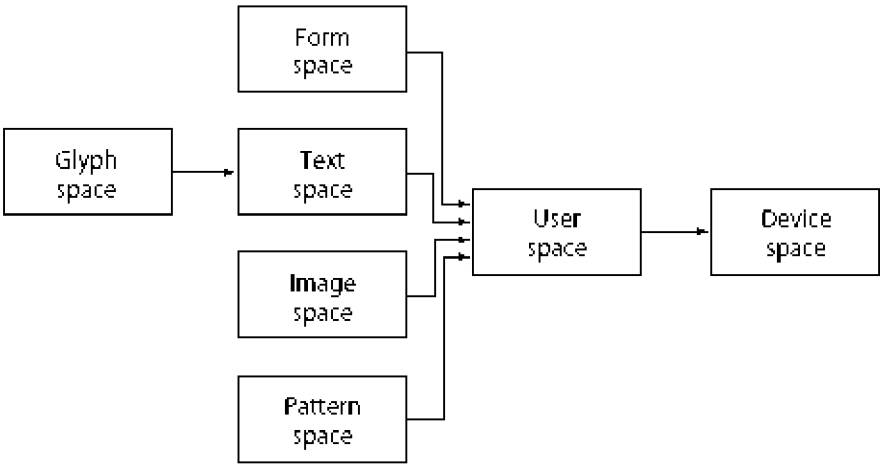


图 3-3 各坐标空间间的转化

3.2 坐标变换的数学原理

本节对变换矩阵的讨论是基于以下两个基本点：

- ① 坐标变换是对坐标系统的改变，而不是对几何图形对象的改变。所有在坐标变换前描绘的坐标都不会受到当前坐标变换的影响。与这个基本点相对应的是，变换图形对象。虽然两者在理论上都是正确的，但是在计算的细节上还是有一些差别。
- ② 变换矩阵定义变换是从新的坐标系统（变换后的坐标系统）到原始的坐标系统（变换前的坐标系统）的变换。

为了能使一个定义在二维空间内的点 (x,y) 在下面将要讨论的矩阵变换中能够方便使用，我们用一个第三个元素是常量 1 的向量 [x y 1] 来表示这个点。

用来表示在两个坐标系统间变换的是一个如下所显的 3 乘 3 的矩阵：

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

因为坐标变换矩阵只有 6 个元素是可变的，所以我们用一个有 6 个元素的数组 [a b c d e f] 来表示它。

坐标变换以矩阵乘积的形式表示：

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

因为变换矩阵定义的变换是从变换后的坐标系统到原始的未变换的坐标系统，所以 x' 和 y' 在这个等式中表示的坐标是在未变换的原始坐标系统中的值， x 和 y 表示的坐标是在变换后的坐标系统中的值。将上面的等式展开得到：

$$\begin{aligned} x' &= a \times x + c \times y + e \\ y' &= b \times x + d \times y + f \end{aligned}$$

如果要想实现一系列坐标变换，可以将对应每个独立变换的变换矩阵进行连乘得到一个等效的变换矩阵来实现复杂的坐标变换。

矩阵乘法不满足交换律，所以对每个独立变换的变换矩阵相乘的顺序是很重要的。假设要进行两个坐标变换，先在用户空间中进行放缩变换，然后再进行从用户空间到设备空间的变换。设 M_s 为实现在用户空间中进行放缩变换的矩阵， M_c 为实现从用户空间到设备空间变换的变换矩阵。由于坐标总是定义在坐标变换后的坐标系统中的，正确的变换顺序一定是先在默认的用户空间中完成放缩变换，然后再进行从用户空间到设备空间的转化，其对应的表达式如下：

$$X_D = X_U \times M_C = (X_S \times M_S) \times M_C = X_S \times (M_S \times M_C)$$

其中：

X_D 为设备空间中的坐标；

X_U 为默认用户空间中的坐标；

X_S 为放缩后的默认空间中的坐标。

由此可以看出当一个新的坐标变换要实施在一个已有的坐标变换之后时，表示新变换的矩阵必须左乘表示已有变换的矩阵。既：当要实现一组坐标变换时，表示组合变换的矩阵（ M' ）是通过将表示新增变换的矩阵（ M_T ）左乘到表示所有已有变换的矩阵（ M ）上得到的。既如下表达式：

$$M' = M_T \times M$$

当光栅化一个图形对象时，应用程序有时需要实现上述变换的逆变换，既：通过已知的设备空间中的坐标，来求得相应的用户空间中的坐标。但是，并不是所有的变换都是可逆的。比如，如果在变换矩阵中 a 、 b 、 c 和 d 全为 0，则所有的用户空间中的坐标经过变换后都会和设备空间中得到相同的坐标，因此它就没有唯一的逆变换。这种由例如放缩 0 倍等操作引起的不可逆变换没有多大的实际意义。在描绘图形对象时，使用不可逆矩阵会造成不可预测的动作^[20]。

3.3 几种基本的坐标变换矩阵

坐标变换矩阵确定了两个坐标系之间的转化关系，通过改变坐标变换矩阵中元素的值，可以完成放缩、旋转、平移等基本操作。如上节所述，我们用一个含有 6 个元素的数组[a b c d e f]来表示一个坐标变换矩阵，它可以确定任何从一个坐标系到另一个坐标系的线性变换。

3.3.1 几种基本的坐标变换矩阵

- ① 平移 (Translation): $(1 \ 0 \ 0 \ 1 \ t_x \ t_y)$ ，其中 t_x 和 t_y 分别是坐标原点在水平方向和竖直方向上的位移。如图 3-4:

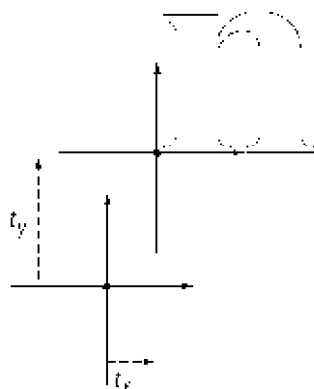


图 3-4 平移

- ② 放缩 (Scaling): $(s_x \ 0 \ 0 \ s_y \ 0 \ 0)$ ，在放缩后的坐标系中水平方向和竖直方向上 1 单位长度分别相当于在放缩前的坐标系中 s_x 和 s_y 单位长度。如图 3-5:

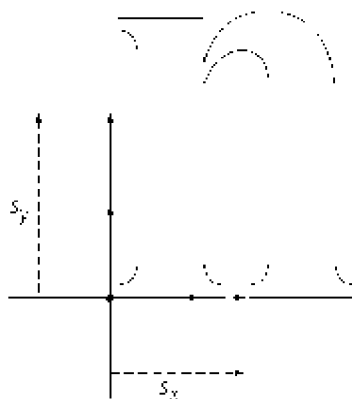


图 3-5 放缩

- ③ 旋转 (Rotation): $(\cos \theta \ \sin \theta \ -\sin \theta \ \cos \theta \ 0 \ 0)$ ，将坐标系统的坐标轴按逆时针方向旋转 θ 度。如图 3-6:

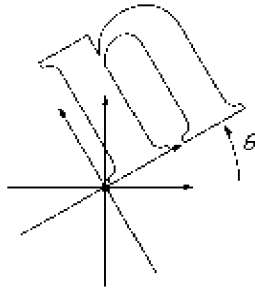


图 3-6 旋转

- ④ 倾斜 (Skew): $\begin{pmatrix} 1 & \tan\alpha & 0 \\ 0 & \tan\beta & 1 \end{pmatrix}$, 将坐标轴 X 倾斜 α 度, 将坐标轴 Y 倾斜 β 度^[22]。如图 3-7:

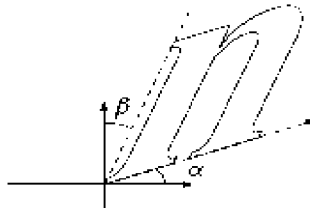


图 3-7 倾斜

3.3.2 复合变换

如果要想实现几种坐标变换的组合, 那么执行它们的顺序是很重用的。例如: 对 X 轴先放缩再平移和先平移再放缩是不一样的。一般的, 为了获得期望的转化结果应该遵循下列的转化顺序:

- 1、平移;
- 2、旋转;
- 3、放缩或倾斜。

例: 假设有如下的变换操作:

- 沿水平方向平移 10 单位的距离, 沿竖直方向平移 20 单位的距离;
- 旋转 30 度;
- 将 X 轴上的坐标放大 3 倍。

图3-8表示了以“放缩—旋转—平移”的顺序进行的复合变换。

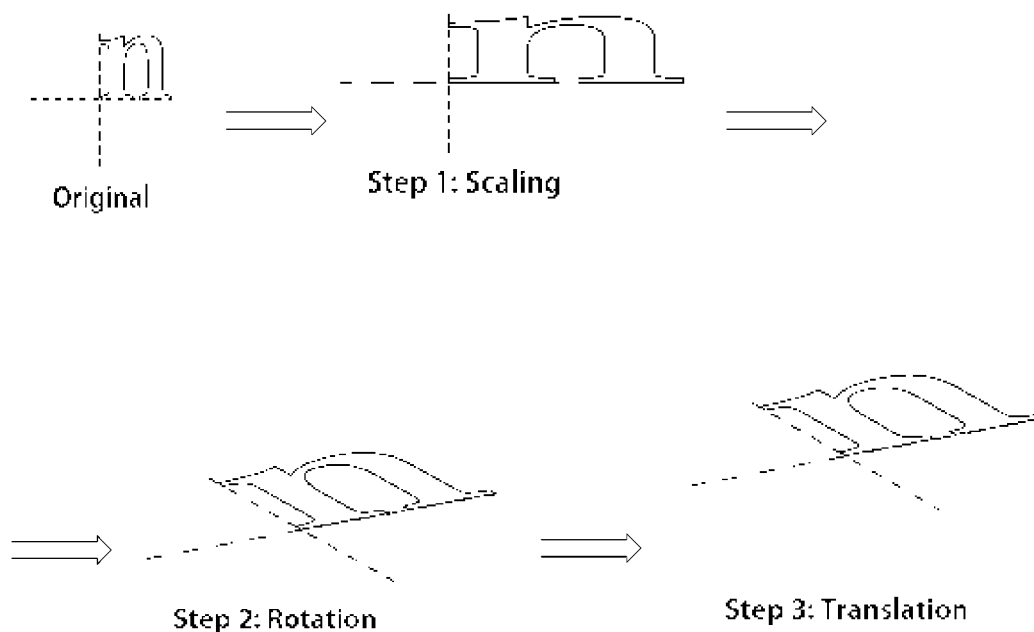


图3-8复合变换

图3-9表示了以“平移—旋转—放缩”的顺序进行的复合变换。如图以“放缩—旋转—平移”的顺序进行的复合变换所得到的结果出现了失真，它使变换后的X轴和Y轴不再正交；而以“平移—旋转—放缩”的顺序进行的复合变换得到的结果没有出现失真。

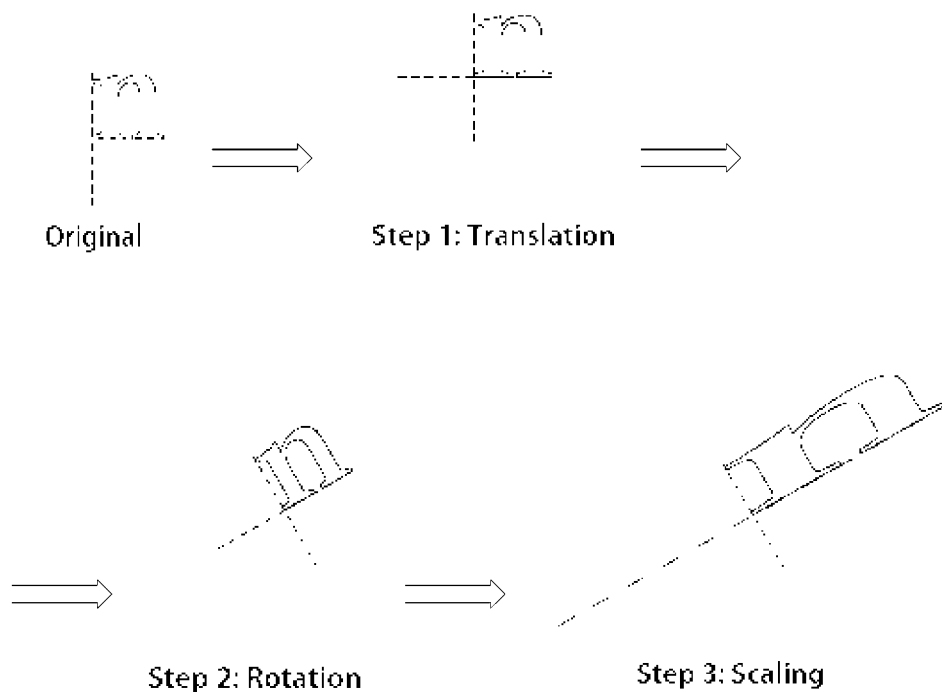


图 3-9 复合变换

第四章 TrueType 字体引擎的基本原理

4.1 Font 文件的生成

Font 文件的生成一般分成三个步骤如图 4-1:

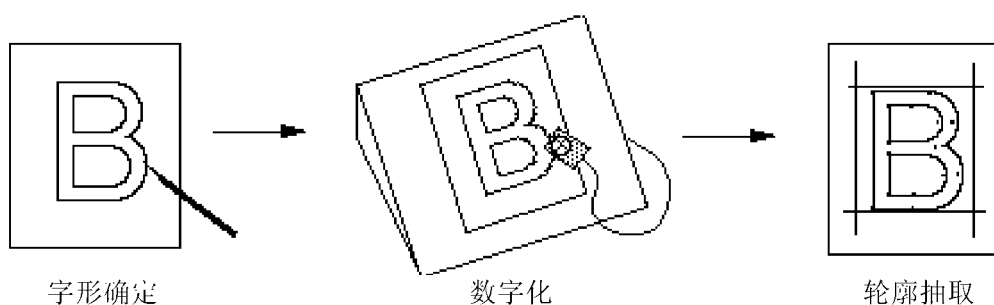
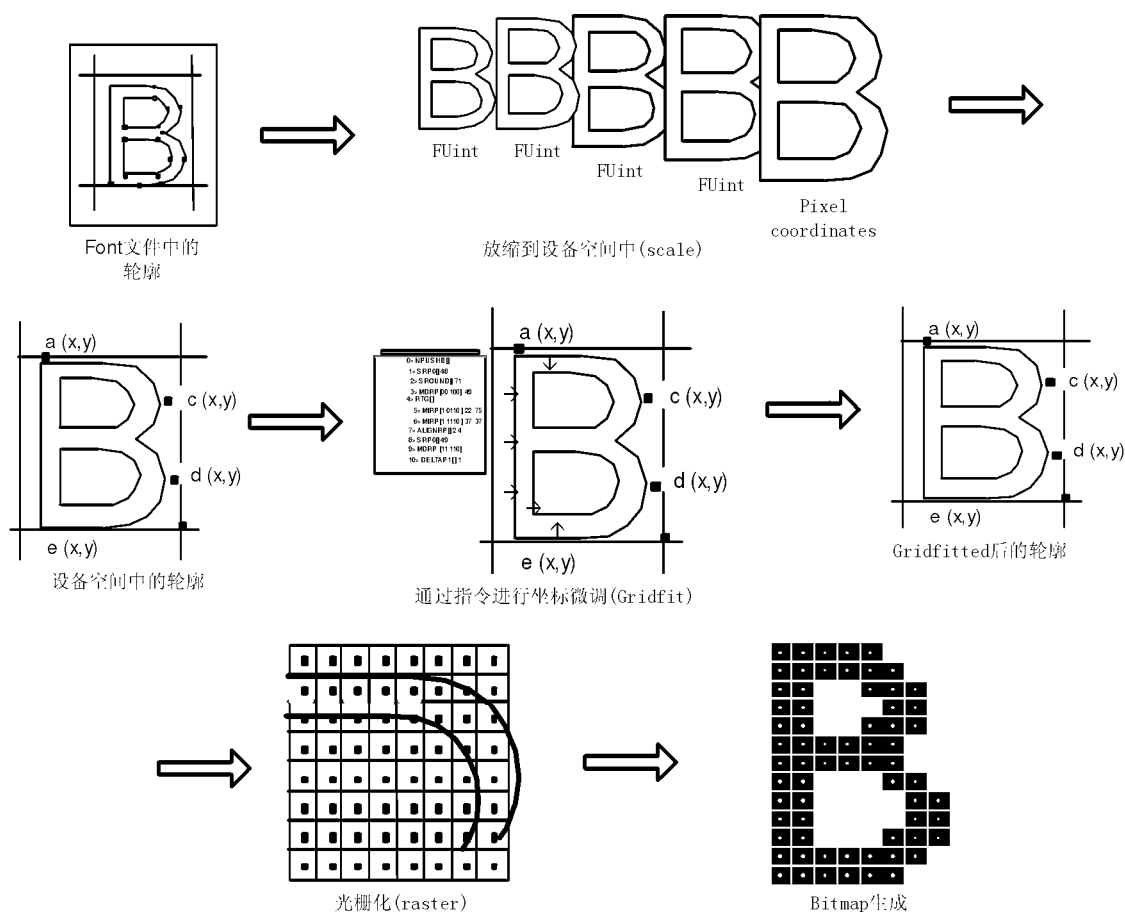


图 4-1 三个步骤

- ① 确定字体风格并生成该字体包括的所有字符的字形。例如，可以请书法家在纸张上创作所有字符的字形。
- ② 数字化已经得到的所有字形。例如，将写在纸张上的字形扫描到电脑中，生成数字化的字形文件。
- ③ 从字形文件中抽取轮廓，并以坐标点的形式表示轮廓。

4.2 从 Font 文件到纸面

一个字符要打印在纸面上或者显示在显示器上，大概要经过以下过程。首先是 **Scale**，即要把存储在 TrueType 字体文件中的描绘该字符 **glyph** 的轮廓放缩成需要的尺寸。实际上就是将描述轮廓的点进行坐标变换。变换后的点坐标不再是以 **FUnit** 为用户空间中的坐标，而是以像素为单位的设备空间中的坐标。然后是指令化既 **GridFit**，就是通过指令将描述轮廓的点的坐标进行微调。指令化程序要执行关于该字符 **glyph** 的指令，这些指令是用来使字符能更好的在光栅设备中显示。最后是光栅化即 **Raster**，就是将指令化的轮廓填充后生成 **Bitmap** 位图显示在光栅设备上^[28]。整个过程如图 4-2:



图：4-2 显示字符的过程

4.3 TrueTypeFont 技术相关的基本概念

4.3.1 轮廓 (Outline)

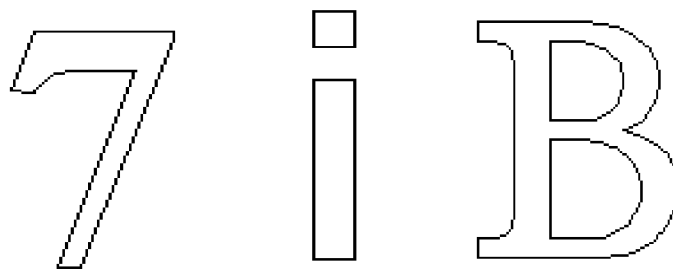


图 4-3 轮廓

在 TrueType 字体中字形 (glyph) 是通过轮廓来描述的。一个 glyph 由一系列的笔画 (Contour) 组成。glyph 可分为简单 glyph 和复合 glyph。一个简单 glyph 可以仅仅包含一个 Contour，较复杂的简单 glyph 则可以包含几个 Contour。图 4-3

分别表示了 1 个、2 个、3 个 contour 的 glyph。复合 glyph 则是由两个或两个以上简单 glyph 组合而成。

Contour 由直线和曲线组成。组成 contour 的曲线是通过一组定义二阶贝塞尔样条 (Bezier-spline) 的点描述的。Bezier-spline 在 TrueType 字体中由两类点描述，一类是在轮廓上的点，一类是不在轮廓上的点。定义曲线时任何在与不在轮廓上的点的组合都是合法的。直线则是由两个连续的在轮廓上的点定义的。图 4-4 是一个用点描述的 glyph 的示例。其中点 0 至点 44 为轮廓上的点，空心点为不在轮廓上的点。

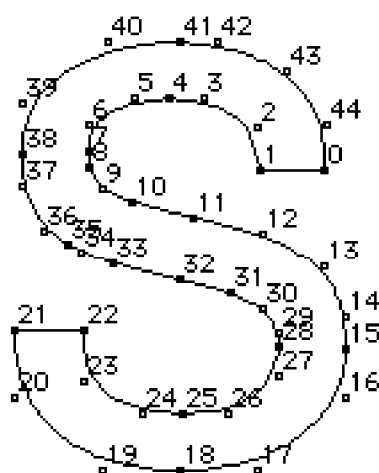


图 4-4 点描述的轮廓

对于同一个字符的 glyph，描述组成它的直线和曲线的点的编号必须是连续且有序的。至于是升序还是降序，将决定轮廓的方向；亦即在填充轮廓时，填充区域是在轮廓的左边还是轮廓的右边。

4.3.2 Funits 和 EM

在 TrueType 字体文件中，点的坐标值是以 Funits 为单位的，Funits 是 EM 中最小的量度单位，而 EM 是一个虚构的用来排列和量度 glyph 的方框。在排版时要是没有额外的行距，上下两行可能重叠，为了避免这种重叠，在 EM 中包括了一定长度的空隙，所以 EM 典型的长度等于整个 glyph 的长度加上这种额外加入的空隙的长度的和。

在铅字印刷时代，glyph 的大小必须被限定在 EM 之中；但对于数字字体来说，glyph 是可以超出 EM 框的。EM 可以定义得足够大，使所有的 glyph 都能被包含在 EM 框中；但是如果 glyph 超出 EM 框对于某些实现更加方便的话，glyph 也可

以超出 EM 框。TrueType 字体对这两种方式都支持，选择哪种方式完全取决于 TrueType 字体的制造者。如图 4-5 是这两种方式的一个例子，左边的为 EM 框包含整个 glyph 的情况，右边的为 glyph 超出 EM 框的情况。

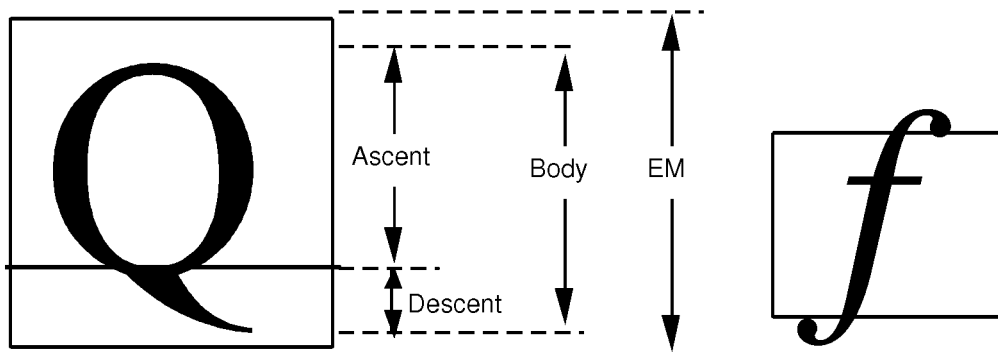


图 4-5 EM

4.3.3 Funits 和格栅 (grid)

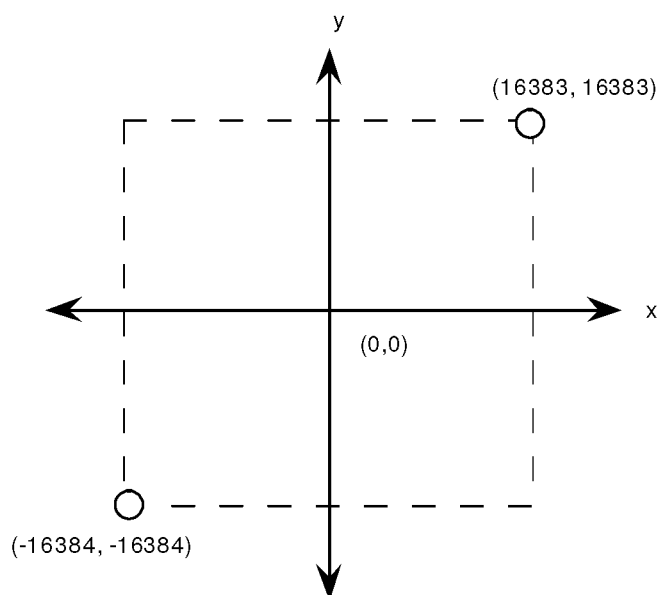


图 4-6 格栅

数字化一个 Font 关键的一点是决定组成 glyph 的点在什么分辨率下描述。格栅 (grid) 就是对这种分辨率的具体体现。格栅是一个二维的坐标系统，以 FUnit 为最小单位，(0, 0) 点为坐标原点，X 轴表示水平方向上的位移，Y 轴表示竖直方向上的位移。格栅是一个有限平面，每个点的坐标必须在 -16384 到 +16384 FUnit 之间，如图 4-6。

虽然 EM 的原点可以和轮廓的位置没有任何关系，但是在应用程序的开发中对于给定的字体轮廓位置的确定，有一些约定俗成的原则。例如 Roman 字，它被

故意放置在水平方向上。与 X 轴重叠的直线 Y=0，被当成这种字体的基线 (baseline)，直线 X=0 没有特殊的意义。但是制造商可以为原点的横坐标挑选一个自己的标准，来改进其应用程序的实现。对于非 Roman 字，原点的 X 坐标和 Y 坐标所代表的意义可能就会不同。

每个 EM 所包含的 FUnit 数被叫做坐标格栅的粒度。这种粒度是有 Font 制造商决定的，当粒度是 2 个幂时，比如 2048，在放缩轮廓时会较快。EM 被以 FUnits 为单位分割成一个坐标系统，所有定义在这个坐标系统中的坐标点都有一个整数的坐标。一个 EM 所包含的 untis 越多，EM 中定义的点的位置就越精确。如图 4-7

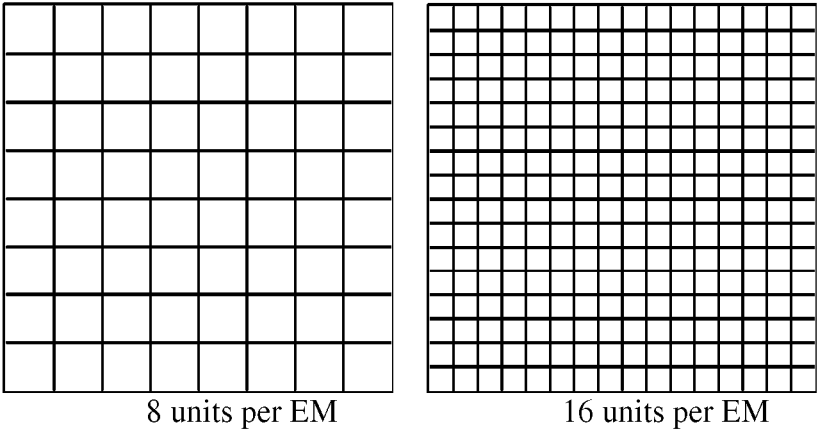


图 4-7 EM

FUnits 的大小是随着 EM 大小的变化而变化的。而对于一种 Font 而言，无论该 Font 的磅值 (point) 是多少，EM 所包含的 untis 数都是保持不变的；所以 EM 的磅值是随着 glyph 的磅值变化的，当该 glyph 以 9 磅的大小显示的时候，EM 的大小就是 9 磅，当 glyph 以 10 磅的大小显示的时候，EM 的大小就是 10 磅。因为 EM 所包含的 untis 数是不随 Font 显示的磅值变化的，所以 FUnits 的绝对大小是随着磅值变化而变化的。如图 4-8：

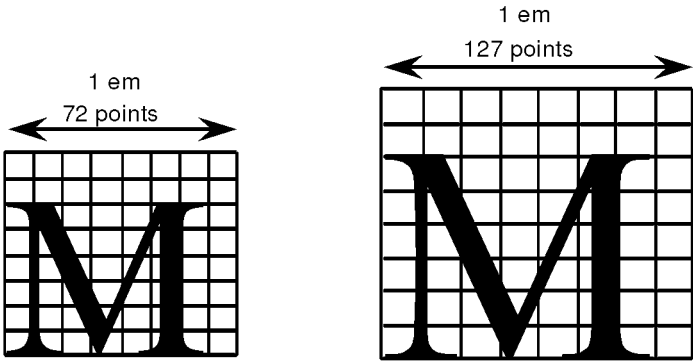


图 4-8 不同大小的 M

72 磅的 M 和 127 磅的 M 在 8 units 的 EM 中显示。因为 FUnits 和 EM 是相互关联的，所以无论一个 Font 以什么样的磅值显示，glyph 中的一个点，以 FUnits 为单位，都有相同的坐标值。这对应用程序的实现是一个很便利的特性，因为我们只考虑原始轮廓和最终光栅化字符之间的变化^[29]。

4.4 放缩 (Scale) glyph

放缩的目的是将存储在 Font 文件中的 glyph 轮廓坐标转化为应用程序所需要的坐标。无论定义 glyph 轮廓的 EM 的分辨率是多少，在 glyph 被显示出来之前必须根据显示它的输出设备的特性进行适当的放缩。放缩后的轮廓必须以绝对距离而非相对距离来描述轮廓，因此构成 glyph 轮廓的点都是以一系列像素来描述的。直观上看，像素就是会被显示器或打印机实际输出的点。为了能够处理高精度的轮廓，TrueType 以 1/64 个像素长度为单位来建立像素坐标。

4.4.1 从 FUnits 到 Pixels

从 EM 的 FUnits 到 Pixels 值的转换公式如下：

$$Point\ Size \cdot \frac{Resolution}{72 \cdot Units_Per_EM}$$

其中 PointSize 是 glyph 在显示设备上要显示的大小（磅值）；Resolution 是输出设备的分辨率 dpi，即每英寸多少 Pixels；分母上的 72 是 1 英寸的磅值，即 72 磅长的线段长 1 英寸；Units_Per_EM 表示 EM 包含多少 FUnits。例如：某 Font 的每个 EM 包含 2048 个 Units，在 EM 中长 600FUnits 的 glyph 要在一个分辨率为 96dpi 的屏幕上以 22 磅的大小显示，则该 Font 中的每个 FUnits 在该屏幕上所占的 Pixels 为：

$$22 \times \frac{96}{72 \times 2048} \approx 0.014323$$

而这个 glyph 在屏幕上就应该占 $0.014323 \times 600 = 8.59$ 像素长^[29]。

4.4.2 光栅设备的特性

对任何特定的光栅设备，dpi 都表示其每英寸有多少 Pixels，例如：VGA 在 OS/2 和 Windows 下是 96dpi，大多数激光打印机则是 300dpi。而有些设备如 EGA，在水平方向和垂直方向上有不同的分辨率，EGA 的分辨率就是 96x72。

EM 在光栅设备上所占的 Pixels 数和输出的光栅设备有密切关系。一个 18 磅的字符的 EM，在 72dpi 的设备上占 18pixels；在 300dpi 的设备上占 75pixels；在 1200dpi 的设备上占 300pixels。如图 4-9：是 18 磅的“8”在 72dpi，300dpi，1200dpi 的光栅设备上的显示结果。

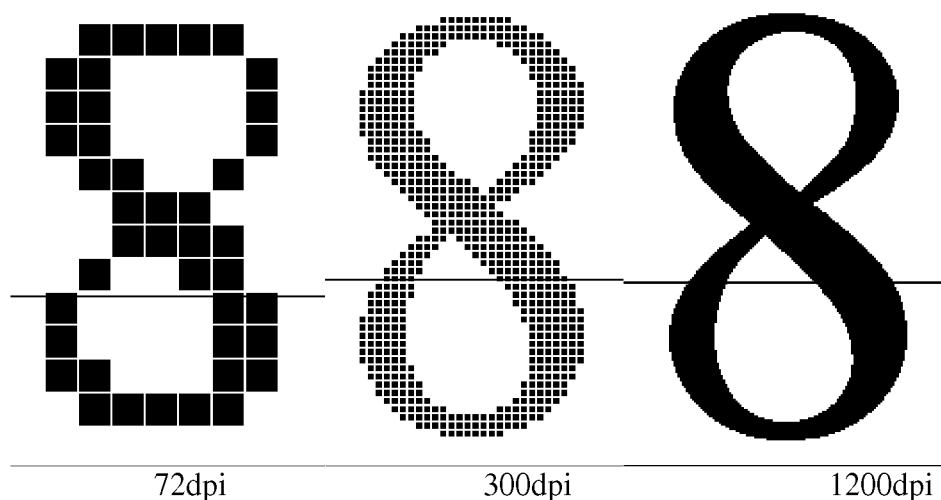


图 4-9 不同 dpi 的 8

在某个具体设备上以某个磅值显示某个字符的有效分辨率是以 PPEM (pixels per em) 来衡量的。PPEM 计算公式如下：

$$PPEM = Point\ Size \cdot \frac{Resolution}{72 \cdot Units_Per_EM} \cdot Units_Per_EM$$

$$= Point\ Size \cdot \frac{dpi}{72}$$

那么在 300dpi 的激光打印机上，一个 12 磅的 glyph 的 PPEM 为 $12 \times 300 / 72 = 50$ ；在 2400dpi 排字机上的 PPEM 为 $12 \times 2400 / 72 = 400$ ；在 VGA 中 12 磅的 glyph 的 PPEM 为 $12 \times 96 / 72 = 16$ 。当然，12 磅的字符在 72dpi 的设备上的 PPEM 为 $12 \times 72 / 72 = 12$ ，这说明在 72dpi 的光栅设备上字号的磅值和 PPEM 是相等的；不过应当注意的是在传统的印刷样式中每英寸包含 72.2752 个 points 而非 72，这样一来每个 points 为 0.013836 英寸^[30]。

如果已知 PPEM，将 EM 中的坐标（以 FUnit 为单位）转换成设备空间中的坐标（以 pixel 为单位）的公式如下：

$$Pixel_coordinate = EM_coordinate \cdot \frac{PPEM}{Units_Per_EM}$$

那么在 Units_Per_EM 为 2048，PPEM 为 12 磅的情况下，一个在 EM 中坐标为 (1204, 0) 的点，在设备空间中的坐标应为 (6, 0)。

4.5 指令化 (Gridfitting)

为了保证字符在不同输出设备上以不同磅值都能正常的显示，我们需要在一些情况下对一些字符的显示进行调整。这些调整一般包括，对主要笔画宽度的调整，颜色或者留白的调整和对遗失点 (dropouts) 的补充。

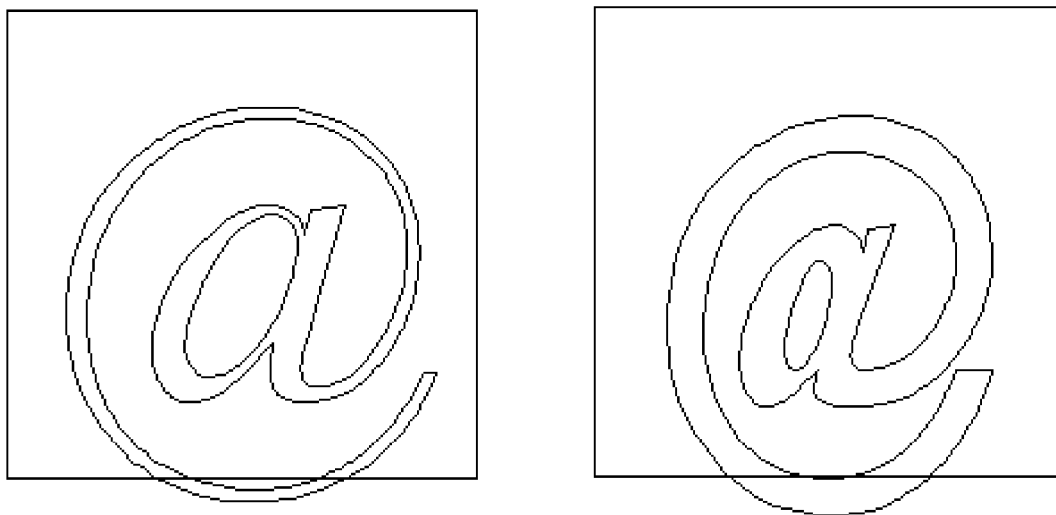


图 4-10 调整前后的 12 磅的轮廓

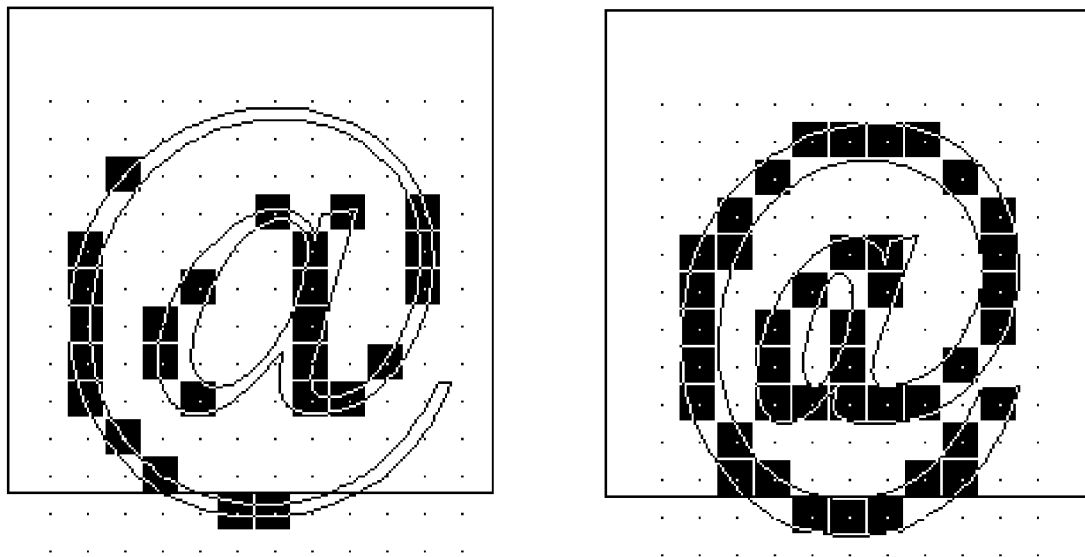


图 4-11 gridfitting 的效果

从根本上讲，保证正确的像素点在 glyph 光栅化时被点亮，就能确保字符正常显示，因为点亮的像素和最终产生的 glyph 的 bitmap 图像是一一对应的。Glyph 的轮廓形状决定了给定大小字符的 Bitmap 图像，所以为了保证产生高质量的 Bitmap

图像，我们要对 glyph 的原始轮廓进行恰当的改变。这种恰当的改变被称作指令化或轮廓调整（Gridfitting）。图 4-10 是一个 TrueType glyph 轮廓在 Gridfitting 前后的例子，其中左边的为 glyph 的原始轮廓，右边为 Gridfitting 后 glyph 的轮廓。继续上面的例子，如图 4-11，左边是没有进行 Gridfitting 直接进行填充的结果，右边是 Gridfitting 后填充的结果^[31]。

Gridfitting 是通过修改描述轮廓的点的坐标来实现，点的编号不变。对点坐标的修改又是通过执行保存在字体文件中对应字符的指令集中的命令来完成的。

4.5.1 指令

TrueType 指令集提供了一系列命令在 glyph 被放缩（scaled）后来完成字符的光栅化。换句话说就是，指令决定了特定大小的 glyph 轮廓在特定 dpi 下 Gridfitting 的方法。指令可以针对特定大小的 glyph 在给定的显示设备上 Gridfitting，修改它的轮廓使其包含正确的像素点，修改轮廓说到底就是根据指令移动描述轮廓点的位置。

TrueType 字体应用指令与否可以被设定。非指令化的 Font 在充分高的显示分辨率和充分大的磅值下一般是会产生较好的结果。对于非指令化的 Font 要得到较好的结果，Font 设计不同对显示分辨率和磅值的要求也是不一样的；Font 被运用的场合也是一个决定该 Font 是否指令化的一个要素。一般来说，在低分辨率小磅值的情况下，都推荐指令化。

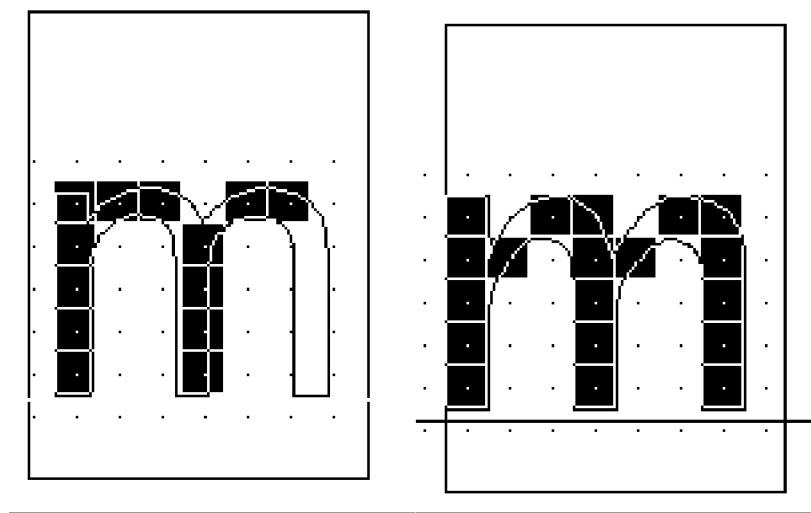


图 4-12 指令的效果

对于 Font 指令化有两方面的问题需要考虑，一是分析 glyph 的关键要素，二是在运用 TrueType 指令时保持这些关键要素。TrueType 指令可以做到足够的灵活，

使字符特征能在较小的磅值下，以足够的像素，大致得表现原始设计的风味^[32]。

但是为了产生想要的结果，TrueType 指令解释器怎么知道，哪个轮廓需要修正呢？相关的这些信息包含在 Font 文件中每个字符的指令集中。指令可以指定字符在最初设计上的某方面的特征，必须要在放缩（scaled）中被保持；可以控制 Font 文件中单个或者所有字符的高度；可以针对一个字符，对其中的关键笔划间位置的关系或它们的笔划宽度进行调整。

如图 4-12 示例了指令对于一个 9 磅值的 Arial 字体的小写“m”在小分辨率的显示设备上显示时所起到的作用。左边的“m”由于没有经过指令来修正，没有表示第三根竖线的像素点被点亮，右边的“m”通过指令来修正避免了这种情况。需要说明的是：一个像素点是否被点亮，我们是按照该像素的中心点是否在轮廓中这个基本规则来判定的。

4.5.2 TrueType 指令解释器

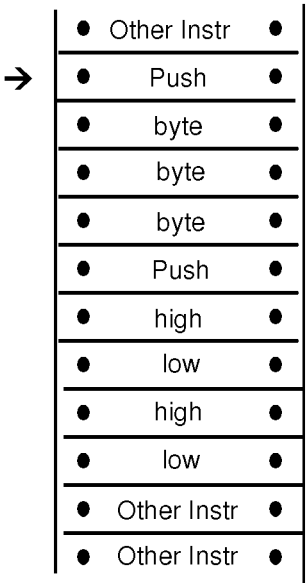


图 4-13 指令栈

指令解释器顾名思义就是用来解释或者执行指令的。具体的说，它是对一系列指令流进行处理并维护了一个堆栈。一般情况下，指令从解释器堆栈中取出数据作为自己的参数，并把执行结果放回堆栈；也有很小一部分的指令是用来向堆栈中压入数据的，这些指令则是从指令流中获取它们的参数的。

解释器所有动作都是为了描述“图形状态”（Graphics State）中的内容。“图形状态”是一个变量集合，这些变量是指令运行的依据，并决定指令运行的效果。

解释器的动作可以归纳为以下几个要点：

1、解释器从指令流中取得指令。

指令流是由操作码和操作数组成的有序队列，其中每个操作码一个字节 byte 长，一个操作数可以是一个字节长或者一个字 word 长（两个字节）。如果操作数是一个 word 型数据，那么在指令流中的第一个字节为该 word 的高字节，指令流中的第二个字节为该 word 的低字节。如图 4-13 是一个解释器堆栈的例子，箭头指示的是下一个将要执行的指令。

2、指令的执行

- 如果是一个向堆栈中压入数据的指令，它将从指令流中获得它的参数。
- 其它指令将从堆栈中弹出的数据，作为自己需要的参数。如图 4-14

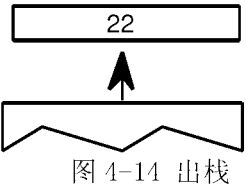


图 4-14 出栈

- TrueType 解释器维护的是一个堆栈，而堆栈是一个“先进后出”（LIFO）的数据结构。指令从堆栈中获得的数据都是最后进入堆栈的，从堆栈中弹出栈顶元素的命令是“pop”。当一个指令产生一些结果，并将这些结果压入堆栈的栈顶时，这些数据很有可能是下一条指令的输入。
- 指令集中包含所有对堆栈操作的命令，如：向栈顶压入元素，弹出栈顶元素，清空栈，复制栈中元素等等。
- 指令执行的效果取决于构成 Graphics State 中变量值的设定。
- 指令可以改变 Graphics State 中变量的值，如图 4-15，Graphics State 中 rp0 的值被堆栈中的弹出的值更新了。

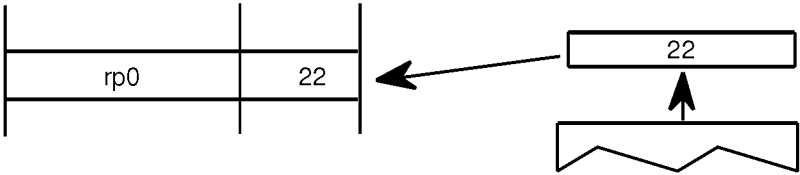


图 4-15 Graphics State 的更新

3、解释器将一直执行指令流中的指令，直到指令流中所有指令都执行完^[33]。

4.5.3 指令的运用

指令可以在许多组成 TrueType 字体的表中存储，它可以作为 Font 程序的一部分，也可以是 CVT 程序或者是 glyph 数据的一部分。出现在 Font 程序和 CVT 程

序中的指令是针对整个 Font 起作用的，而存储在 glyph 数据中的指令只是针对单个 glyph 的。

4.5.3.1 Font 程序

Font 程序由一个只执行一次的指令集合构成。当这个 Font 第一次被应用程序访问的时候，Font 程序被执行。Font 程序被用来创建函数定义（FDEFs）和指令定义（IDEFs）。函数定义和指令定义将在 Font 文件中的其他地方用到。

4.5.3.2 CVT 程序

每当字符的磅值发生变化时 CVT 程序都会被执行，CVT 程序负责的是整个 Font 的变化而不管管理单个的 glyph。它被用来创建 CVT 表（Control Value Table）中的值。建立 CVT 表的目的是为了简化在指令化 Font 的时候对一致性的管理，表项中的值是对于许多 glyph 来说是相同的，这些值将被两种间接指令(MIRP 和 MIAP)引用。例如 CVT 表中调整笔化宽度的项。表 4-1 是一些 CVT 表中的值。

表 4-1 CVT 表值

Entry #	Value	Description
0	0	upper and lower case flat base (base line)
1	-39	upper case round base
2	-35	lower case round base
3	-33	figure round base
4	1082	x-height flat
5	1114	x-height round overlap
6	1493	flat cap
7	1522	round cap
8	1463	numbers flat
9	1491	numbers round top
10	1493	flat ascender
11	1514	round ascender
12	157	x stem weight
13	127	y stem weight
14	57	Serif
15	83	space between the dot and the I

引用 CVT 表中值的指令被称作间接指令；相对的从 glyph 轮廓中得到数据的指令叫直接指令。作为 TrueType 字体文件的一部分，CVT 表中的值是以 FUnits 为单位的，当放缩时从以 FUnit 为单位转化成以像素为单位时，CVT 表中的值也要相应的变化^[35]。从 CVT 表中读数据的时候解释器都会进行相应的变化，使输出的数据

都是以像素为单位的。

4.5.4 存储区域

表 4-2 存储区域中的值

Address	Value
0	343
1	241
2	-27
3	4654
4	125
5	11

解释器还维护了一个存储区域来存储解释器堆栈中传过来的数据，指令可以从该存储区读数据或向它写新的数据。存储区的范围 N 是一个 32 位的数值，这个值存储在 Font 文件中的 maxProfile 表的 maxStorage 项中。表 4-2 是一些存储区域中的值。

4.5.5 图形状态 (The Graphics State)

图形状态是一个表，表中的值是描述图形状态的，有一系列初值，指令是以此为根据来调整 glyph 的，其中一些值也可以通过 CVT 程序来修改。无论图形状态表的初值是什么，当解释器针对单个 glyph 来执行指令的时候，图形状态表都会被重建。换句话说就是，图形状态表在 glyph 间没有共用的存储区。在处理某个 glyph 时，改变图形状态表中的值，只会影响这个 glyph 的结果。

4.6 光栅化

扫描转化也就是前面提到的光栅化 (Raster) 即将指令化后的轮廓填充后生成 Bitmap 显示在光栅设备上。基本的 TrueType 扫描转化模式是运用一种简单的逻辑来判断哪个像素是 glyph 的一部分，即点亮那个像素。

其规则如下：

Rule 1: 如果一个像素的中心点落在 glyph 轮廓内，那么这个像素被点亮。

Rule 2: 如果一个像素的中心点正好落在 glyph 轮廓上，那么这个像素被点亮。

我们希望 TrueType 解释器能够修正 glyph，使它在任何磅值和任何变换后，根据 Rule1 或 2 都能正确点亮我们希望的像素^[37]。但是要预先知道所有 glyph 可能的变换是

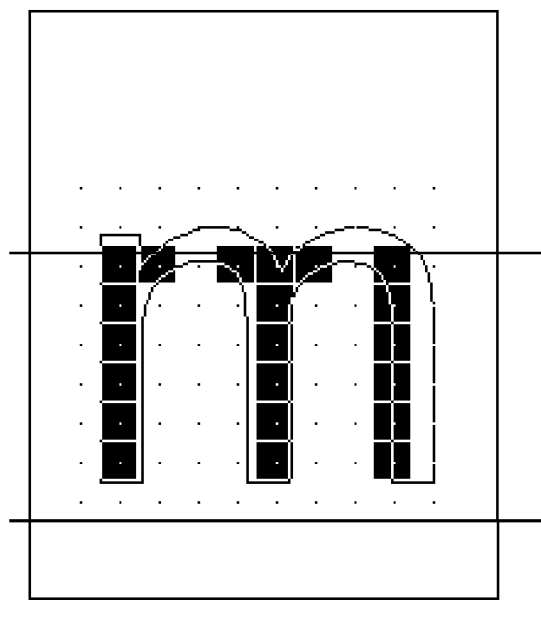


图 4-16 有缺陷的 m

非常困难的，所以 glyph 在指令化时对于每种可能的变换都得到希望的修正也是很困难的。对于复杂 glyph 在小 PPEM 的光栅设备上显示时，这种问题尤其明显。这种问题导致的结果就是，显示出来的 glyph 会包含像素丢失（dropouts）。具体的

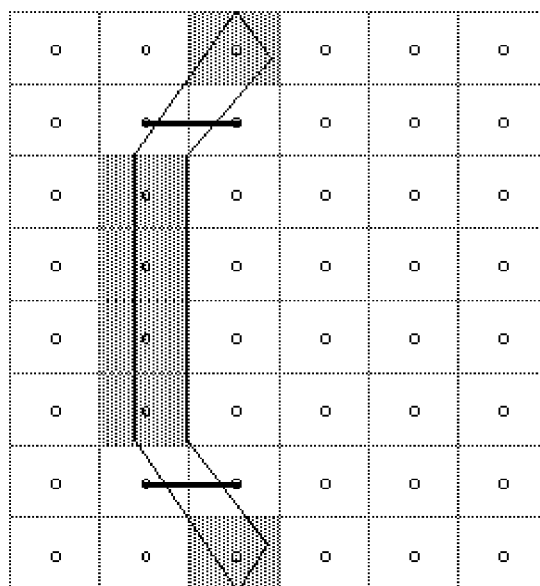


图 4-17 断笔

讲，像素丢失是指在 glyph 内部的连接区域内，两个被点亮的像素间没有被连接起来，而出现了空白，就好象笔化断了似得，如图 4-16。为了避免这种缺陷，在判断某个像素是否点亮时增加了“像素丢失控制模式”。“像素丢失控制模式”启

动与否可以通过设置“图形状态表”中的 **SCANCTRL[]** 值来控制。

判断那些地方可能出现“像素丢失”的方法如下。将两个相邻像素的像素中心用一条假想的线段连接起来，如果这条线段和同一轮廓有两个交点，则这两个像素为可能出现“像素丢失”的像素；如果和这条线段相交的两条边继续以原方向和其他像素产生的类似线段相交，那么这两个像素之一为是“像素丢失”的像素，如图 4-17。如果这两条边和假想线段相交以后马上相交，那么将产生“stub”，而不会产生“像素丢失”。“stub”不会使笔划看起来不连贯，而是使得到的笔划比希望的要短一点，如图 4-18。

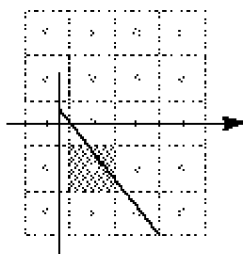


图 4-18 stub

“像素丢失控制模式”可以既填充“像素丢失”的像素，又填充“stub”的像素。其规则如下：

Rule 2a: 如果同一轮廓的两条边和连接两个相邻像素中心点的水平扫描线都相交，且两个像素都没有被 **Rule 1** 点亮，那么将最左边的像素点亮。

Rule 2b: 如果同一轮廓的两条边和连接两个相邻像素中心点的竖直扫描线都相交，且两个像素都没有被 **Rule 1** 点亮，那么将最下边的像素点亮。

“像素丢失控制模式”也可以只填充“像素丢失”的像素，不填充“stub”的像素。其规则如下：

Rule 3a: 如果同一轮廓的两条边和连接两个相邻像素中心点的水平扫描线都相交，且继续和它的水平扫描线相交，且两个像素都没有被 **Rule 1** 点亮，那么将最左边的像素点亮。

Rule 3b: 如果同一轮廓的两条边和连接两个相邻像素中心点的竖直扫描线都相交，且继续和它的竖直扫描线相交，且两个像素都没有被 **Rule 1** 点亮，那么将最下边的像素点亮。

第五章 True Type 字体引擎的实现

本章主要介绍整个 TrueType 字体引擎的整体设计和内部模块的设计及各模块之间的关系。最后较为详细的介绍了 GRIDFITTING 模块，而 RASTERIMAGE 模块作为本文的重点将在最后一节重点详细阐述。

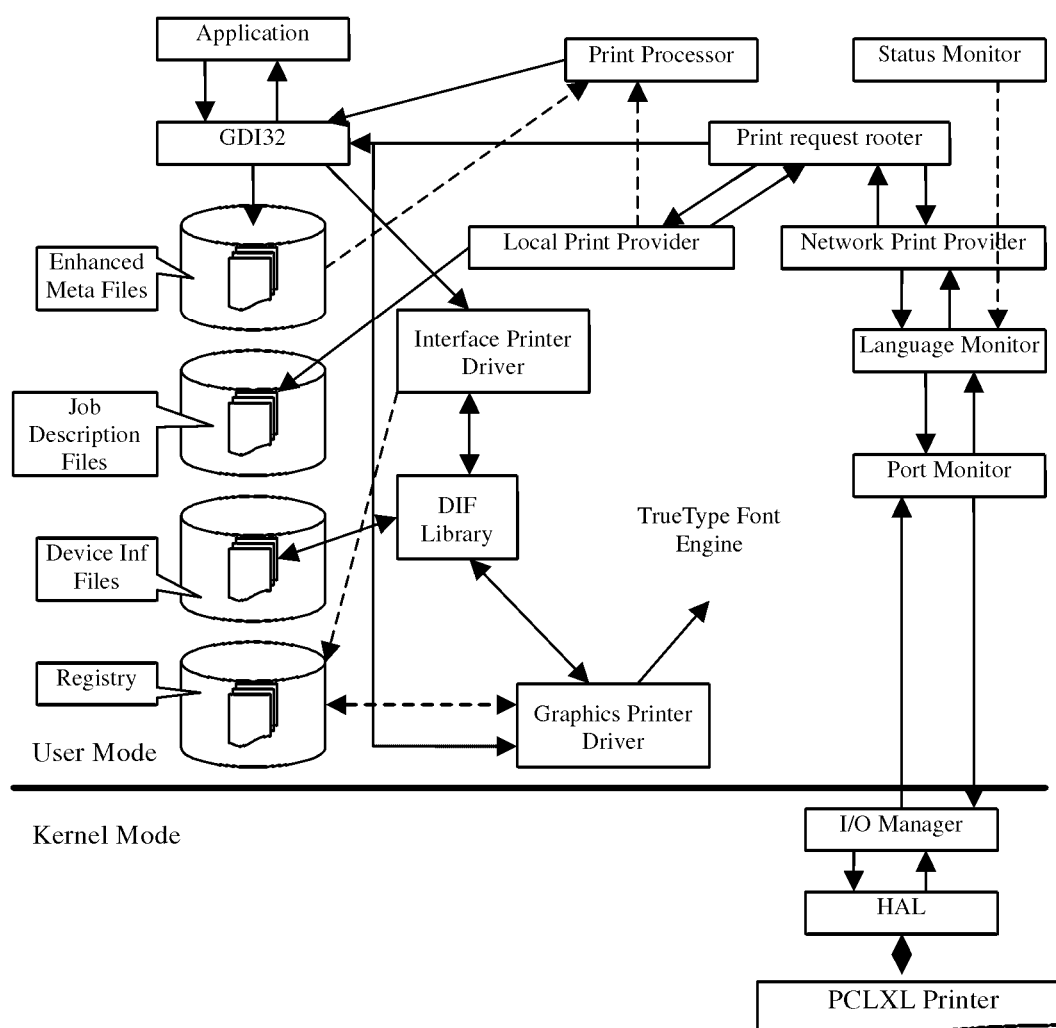


图 5-1: TrueTypeFont Engine 在打印机驱动中的位置

一个字符要打印在纸面上或者显示在显示器上，大概要经过以下过程。首先是 Scale，即要把存储在 TrueType 字体文件中的描绘该字符 glyph 的轮廓放缩成需要的尺寸。实际上就是将描述轮廓的点进行坐标变换。变换后的点坐标不再是以 FUnit 为单位的用户空间中的坐标，而是以像素为单位的设备空间中的坐标。然后

是 Gridfit，即通过指令将描述轮廓的点的坐标进行微调。解释程序要执行关于该字符 glyph 指令，这些指令是用来使字符更好的在光栅设备中显示。最后是 Raster，即将指令化的轮廓填充后生成对应的点阵显示在光栅设备上。所以一个 TrueType 字体引擎必须实现这三个功能。

5.1 外部设计

基于上面儿章的内容，我们针对打印机驱动程序设计并实现了自己的 TrueType 字体引擎。它在整个打印机驱动中的位置如图 5-1 所示。在打印机驱动程序中被 DrvTextOut()调用；在 DrvTextOut()中，也可以通过调用 EngTextOut()来调用微软的字体引擎来实现同样功能，所以该字体引擎是独立于微软的字体引擎的。在调用时，DrvTextOut()将包括 Font name、Font size、Font type、Font file pointer 在内的 FONTOBJ 信息和包括 Glyph index array 在内的 STROBJ 等信息传给 TrueType 字体引擎；TrueType 字体引擎得到这些信息后经过上面提到的三个过程，最后将产生一个数组返回给 DrvTextOut()，该数组经过简单处理后就可以生成对应字符的位图，传给打印机。

5.2 内部设计

本文设计的 TrueType 字体引擎的内部结构如图 5-2 所示分为 FONTRASTER、GRIDFITTING、RASTERIMAGE、PUB 四个模块。

FONTRASTER 模块起一个承上启下的功能，对上，接收从 Graphics Printer Driver 传来的字符和字体信息；对下启动 TrueType 字体引擎。由于在字体引擎的工作过程中的第一步 Scale，即将描述字符轮廓的点进行坐标变换的工作相对较为简单，所以这部分工作也在 FONTRASTER 模块中完成。FONTRASTER 模块将字体信息和经过变换后的字符坐标传给 GRIDFITTING 模块。还是以前面提到的 Arial 字体的“1”为例，它在 arial.ttf 文件中的坐标（单位为 Funit）是：

(763, 0) (583, 0) (583, 1147) (518, 1085) (307, 961) (223, 930)
(223, 1104) (374, 1175) (600, 1377) (647, 1472) (763, 1472)

如果这个“1”以 6pt 的磅值，在 150dpi 的显示设备上显示的话，那么经过 FONTRASTER 模块进行坐标变换后各点的坐标（单位为 64 倍像素）将变为：

(310, 0) (237, 0) (237, 466) (210, 441) (125, 390) (91, 378)

(91, 449) (152, 477) (244, 559) (263, 598) (310, 598)

GRIDFITTING 模块功能较为单一，就是将字符的坐标通过指令化进行微调，将正确的坐标值传给 RASTERIMAGE 模块。应当注意的是，从 GRIDFITTING 模块传出的坐标值应该是以像素为单位，根据条件进行了调整的字符轮廓坐标，RASTERIMAGE 模块可以直接使用。此外 GRIDFITTING 模块还要将控制 RASTERIMAGE 模块执行模式的参数计算出来，传给 RASTERIMAGE 模块。以上工作都完成以后 RASTERIMAGE 模块被调用，根据轮廓坐标和填充模式，填充字符轮廓，生成一个描述字符的数组。由于该数组和生成该字符的 Bitmap 位图有一一对应的关系，所以该数组中的每一个项仅有 0, 1 两个值；0 表示在 Bitmap 位图上该像素不被点亮，1 表示在 Bitmap 位图上该像素被点亮。最后一个模块是 PUB 模块，它与其他三个模块提供公共的函数。

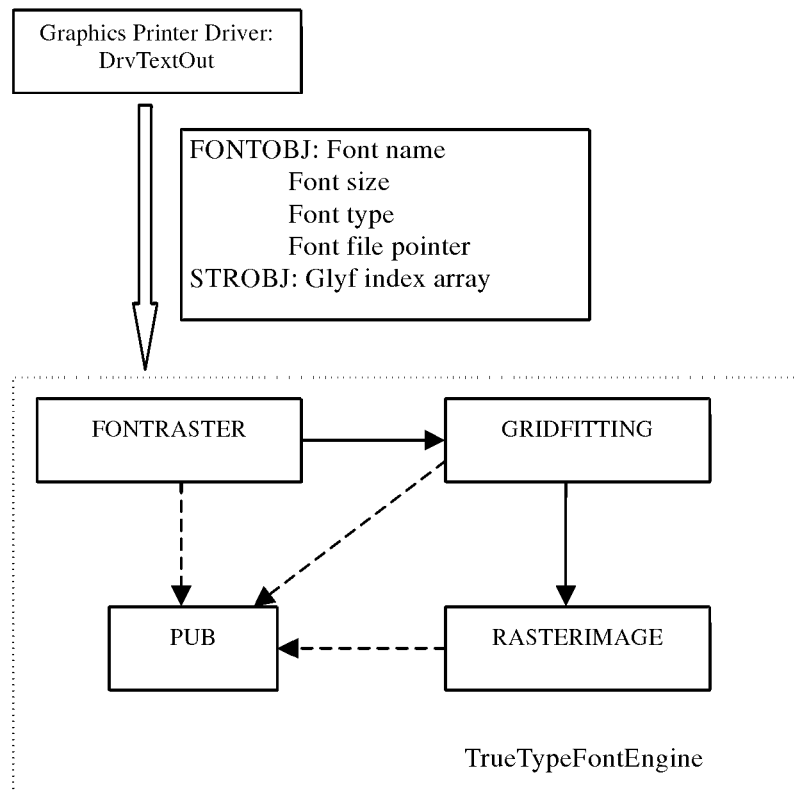


图 5-2: TrueTypeFont Engine 的内部结构

5.3 GRIDFITTING 模块

GRIDFITTING 模块其实就是一个 TrueType 指令解释器，它根据指令将字符进行微调。

5.3.1 指令化 TRUEType 字体的有关概念

5.3.1.1 扫描转换模式与控制舍入

指令化一个TrueType字体的关键因素之一是选择扫描转换模式。字体设计者们可通过设置图形状态变量SCAN CONTROL的值来选择，它共有两种选择模式：快速扫描转换模式与遗漏控制扫描转换模式。TrueType解释器用ROUND STATE来确定数值的舍入方式。图形状态变量ROUND STATE可用指令来设置。其效果映射到输出设备上点的坐标是舍入到像素位置还是舍入到半个像素点位置，是向上舍入还是向下舍入。如果没有满足要求的预定义的舍入选项，指令SROUND将为舍入函数选择相位、门限与周期等来提供非常好的舍入值控制。指令S45ROUND与SROUND的控制作用基本相同，只是S45ROUND要求在X-Y 平面上沿45度角进行移动。许多指令在移动之前要对他们所获得的数值进行舍入，如MDRP、MIRP、MIAP、MDAP和ROUND指令等。这些指令的执行依赖于ROUND STATE图形状态变量和CONTROL VALUE和CUT IN的值。ROFF指令使舍入失效，但允许指令继续观察CUT IN的值。

5.3.1.2 点的概念及其管理

由于TrueType字体文件中，文字的形状是通过它们的轮廓来描述的。一个文字的轮廓由一系列的封闭轮廓线组成。每条封闭轮廓线都由首尾相连的直线和二次B-样条曲线构成，在字体文件中表现为一串轮廓点。这些点分为在线点和线外点两种，在线点在轮廓之上，线外点在轮廓线之外。解释器用区域(ZONE)和参考点(REFERENCE POINT)来对组成当前文字的点集进行移动管理。解释器的任何参照点都在区域1(Z1)和区域0(Z0)这两个区域内。区域1包括当前被执行的文字，区域0用来暂时存贮那些与Z1中文字的点不相符的点坐标。Z0在对文字中不存在的点的处理时能发挥很好的作用，它对记住中间点位置也相当有用。Z0又称过渡(TWILIGHT)区域。TrueType字体文件中的maxp表给出了过渡点(TWILIGHT POINT)的最大数目。从0到TWILIGHT PONTS都被初始化，这些点与Z1的点移动方式相同。解释器有三个区域指针：GEP0、GEP1和GEP2，它们都指向Z0或Z1。初始化时，这三个区域指针都将指向Z1。区域指针提供了对一组点的访问，利用参考点可以访问这组点中的某些特定点。解释器用了三个数字参考点：RP0、RP1和RP2。可根据区域1中文字的任意轮廓点或区域0的任意点对这三个参考点赋值。两个不同的参考点可指向同一个轮廓点。在字体文件中经常要将Z0中的点设置到

关键的度量位置上，此时用指令MIAP和MIRP可令Z0中的点移到指定的位置，并将GEP0指向Z0。总的来说，区域指针和参考点都是图形状态，可使用指令来改变它们的值。许多TrueType指令都依赖于图形的区域指针和参考点来详细地定义它们的操作。TrueType字体缩放器总是在每一个轮廓的末端添加两个幻像点(PHANTOM POINT)。如果一个文字的一条封闭轮廓线需要N个点来表示(假设封闭轮廓线上的点用0到N-1来表示)。缩放器将加上点N和点N+1，这两个点将放在字符基准线上。点N放在字符的起始处，点N+1放在前导宽度点上。点N与N+1都可根据指令化文字的边界和前导宽度效果用TrueType指令控制。用这些幻像点来计算的边界和前导宽度被称为特定设备宽度。特定设备宽度可与线性比例宽度相同，也可不同。相同与否依赖于应用到幻像上的指令。

5.3.2 距离的确定与管理

5.3.2.1 距离的确定

从某种角度上看，文字指令化意味着管理点间距离。管理距离的第一步常常是确定它们的大小。例如，建立CVT表的第一步就涉及到测量字体中关键点间的距离。在一个字体轮廓上测量两点的距离虽不难，但必须考虑某些因素。

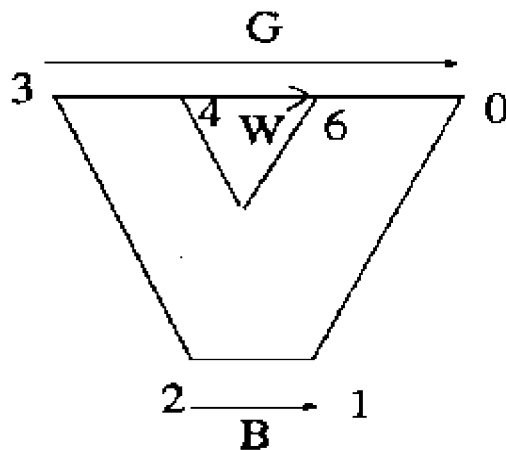


图5-3 三种距离

首先，所有的距离测量都要与投影矢量平行。投影矢量是一个单位矢量，其方向由一个圆的半径来指示。距离被映射到此矢量上并沿着它进行测量。其二，是测量原始字符轮廓上的两点间的距离还是测量网格化轮廓上的两点间的距离，是由测量距离指令(MD)取得的一个布尔值来决定的。此外，TrueType解释器对以下三种距离进行区别：黑色、白色和灰色。有些指令如MDRP、MIRP、ROUND等要求

指明是哪种距离。黑色距离是指经过黑色区域的距离，白色距离是指经过白色区域的距离，灰色距离是黑色与白色距离的总和。如下图5-3所示：[2, 1]之间的距离为黑色距离B，[3, 0]之间的为灰色距离G，[4, 6]之间为白色距离W。

距离类型用来确定ROUND (舍入)和ROUND STATE的指令在不同的输出设备上的工作方式。舍入不影响灰色距离，但黑色和白色距离需要在舍入发生之前进行补偿，所需的补偿量由设备驱动器来设置。比如，一台打印机使用的是大像素，解释器将通过收缩黑色距离增加白色距离进行补偿。由于灰色距离是白色距离与黑色距离之和，所以不会发生改变。当确定两点间的距离时，总是以投影矢量所定义的方向进行测量的。当一个点移动时，它移动的距离也沿着投影矢量进行测量。矢量可被设置为任意所需的方向。一般情况下，投影矢量被设置为以X方向进行测量。这样矢量与X轴平行。同样，若设置为Y方向，矢量与Y轴平行。当投影矢量指向X轴正方向来确定两点的距离时，只需要取这两点的X坐标值之差。值得注意的是，由于投影矢量有方向，所以距离是带符号的。正距离是指按投影矢量的方向进行测量的，负距离是按与投影矢量相反的方向进行测量的。

5.3.2.2 距离的管理

TrueType解释器区别两种距离：一种是以投影矢量的方向进行测量(正距离)，另一种是以与投影矢量相反的方向测量(负距离)。这样就需要变量AUTO FLIP。布尔变量AUTO FLIP的赋值决定CVT表中数值的符号是否重要。如果AUTO FLIP为真，CVT中的值将需要与实际测量的符号相匹配。这将控制距离以投影矢量或相同的方向或相反的方向进行测量。当用指令来改变一字符轮廓上的一些点的位置时，组成该字符的曲线会出现扭结或变形扭曲现象，此时要使该曲线变得光滑。其光滑过程实际上是将那些所有未被移动的点进行再分配，使得它们的位置与移动了的点的位置保持一致。为了助于管理轮廓形状，解释器用到触摸点这个概念。当指令可随时移动一个点时，这个点被标记为可触摸的。有的指令如IUP则只影响非触摸点。为了使增添指令有效工作，最好明确指出触摸的点。当一个文字的宽度缩小至一定大小时，舍入后的值可能为零。这样会导致某些文字的特征丢失。比如，一个字根可能会在小点模式下完全消失。通过对一个像素的MINIMUM DISTANCE赋值可使文字的特征即使在小模式下也不会丢失。

5.3.3 使用 CUTIN 控制

TrueType为字体的文字特征提供了几种协调坐标值。这些协调使得解释器能

在不同输出设备上输出统一的字形结果。对一个文字或特征来说其有效像素数目很少时，协调很有用。它防止在特征尺寸大小上的细小差别随文字轮廓中的像素中心位置的改变而变得非常明显。当特征尺寸大小或位置上的细微差别可由有效像素数表示时，协调成为其依赖。TrueType允许在小PPEM时调整特征，但当有足够的像素点数时，轮廓可转换为原始设计。有两种方法实现协调，这两种方法都用到了CUT IN值。其一是用CVT表和CONTROL VALUE CUT IN并允许用CVT表中的项来协调。此方法允许多种值被协调^[39]。第二种方法是进一步采用协调并使所有的值转换为一个单值。此方法依赖于SINGLE WIDTH CUT IN与SINGLE WIDTH VALUE。CONTROL VALUE CUT IN允许解释器在小模式下选择并使用CVT表中的值，但在其它模式下要转换为原始轮廓。当表中的值与直接从轮廓上所测量的值之差的绝对值大于CUT IN的值时，就要用轮廓测量值。SINGLE WIDTH CUT IN是解释器将忽略在CVT表中的值与轮廓中的SINGLE WIDTH值之间的距离差。它允许特征在只有小PPEM时转换为单个预定义的尺寸大小。当CVT表与SINGLE WIDTH VALUE值之差小于SINGLE WIDTH CUT IN时，就会用到SINGLE WIDTH VALUE。

5.3.4 设计 TrueType 指令解释器

TrueType字体在三个规格表数据中可能包含着指令：Font程序、CVT程序和glyph数据。Font程序仅在字体首次加载时运行，通常它由函数定义和指令定义组成，以供CVT程序和glyph数据中的指令调用。CVT程序在字体尺寸发生变化时运用，它的主要用途是修改CVT表中的数据。Glyph数据中的指令只影响当前glyph本身。该模块的内部结构入图5-4所示。

如果还是以前面提到的Arial字体的“1”以6的磅值，在150dpi的显示设备上显示为例，它经过FONTRASTER模块进行坐标变换后各点的坐标（单位为64倍像素）为：

(310, 0) (237, 0) (237, 466) (210, 441) (125, 390) (91, 378)
(91, 449) (152, 477) (244, 559) (263, 598) (310, 598)

那么再经过 GRIDFITTING 模块进行 Hinting 后各点的坐标（单位为 64 倍像素）将变为：

(256, 0) (192, 0) (192, 500) (170, 539) (96, 421) (64, 406)
(64, 482) (119, 513) (200, 600) (216, 640) (256, 640)

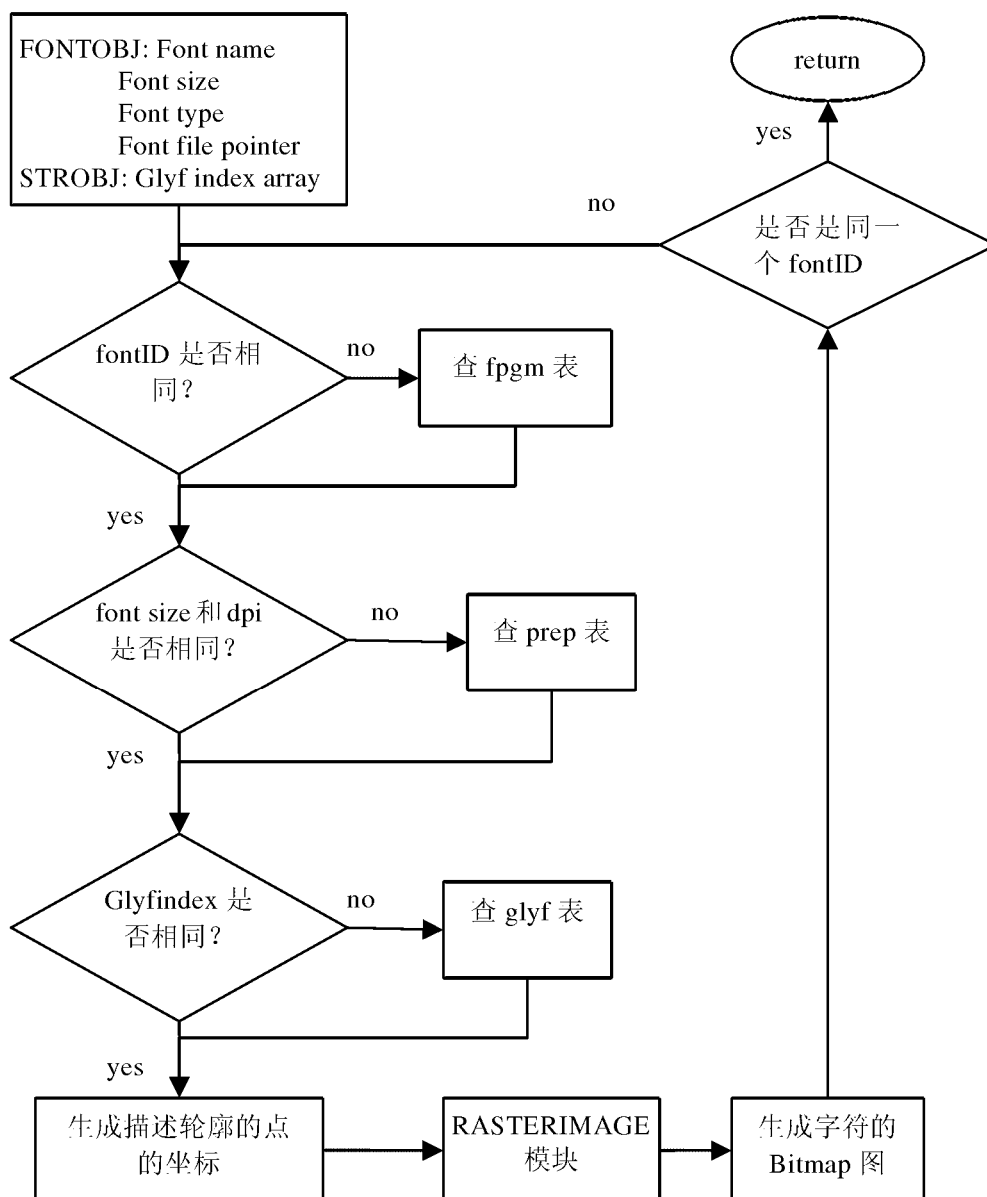


图5-4 Gridfitting模块内部结构

设计TrueType指令解释器的关键在于：

- 1，仔细分析和模拟每条TrueType指令，小心对待每条指令中的堆栈操作，否则将难免引起堆栈溢出。
- 2，分配解释器工作空间：数据堆栈和解释堆栈。数据堆栈供指令压入和弹出数据使用；解释堆栈供控制转向使用。
- 3，建立两个图形状态结构：缺省图形状态和当前状态。缺省状态初始化为TrueType规范中指定的默认值。CVT程序可以修改缺省图形状态中的变量，修改后

的值将成为字体当时尺寸下的默认值。当前图形状态是解释器解释某个glyph指令时的图形状态。每个glyph指令在被解释之前，当前图形状态必须与缺省图形状态一致；这样，某个glyph的指令执行决不会影响到其他glyph指令的执行。

4，将glyph指令和放缩后的轮廓点数据交给解释器解释。解释器的实质性工作是根据指令调整轮廓点所处的平面位置。Glyph轮廓数据的点的顺序必须与glyph指令设计时所隐含的完全一致，否则glyph轮廓在指令执行后将被调整得面目全非^[40]。

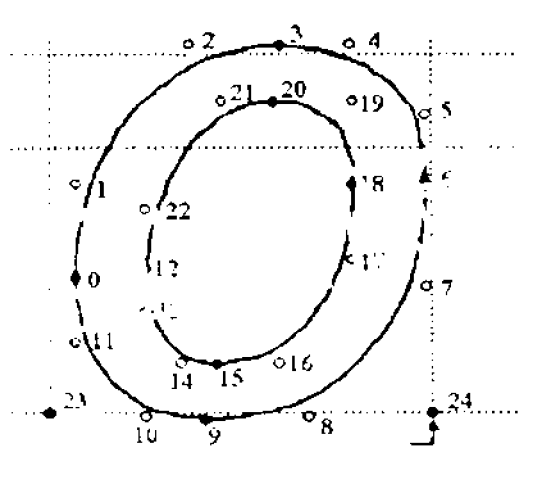


图 5-5 斜体大写字母“O”的轮廓

Windows函数GetGlyphOutline所返回得就是经过指令解释后的轮廓点数据。

在具体实施 Hinting 时可以分为高级 Hinting 和低级 Hinting 两个步骤。下面我们以一个例子来说明文字的 Hinting 过程。图 5-5 是斜体大写字母“O”的轮廓。

首先，对图中的大写字母“O”进行高级 Hinting，其策略为：

- (2) 控制该字符的顶部与底部，使它们与其它大写字母的顶部和底部一致。
- (3) 控制该字符顶部与底部的曲线弧度，使之与其它大写字母的曲线弧度一致。
- (4) 控制该字符左边和右边的曲线弧度，使之与其它大写字母的曲线弧度一致。
- (5) 固定所有点，使它们不被移动。

然后，对图中的大写字母“O”进行低级 Hinting，也就是具体编写 Hinting 代码。在编写 Hinting 代码时，比较有效的办法是先沿着同一个方向进行 Hinting，之后再切换到另一方向。这在 TrueType 中并未作要求。下面我们先控制 Y（竖直）方向及其高度，然后再处理 X（水平）方向，具体步骤如下：

- (1) 将 Hinting 方向设置为 Y 方向
- (2) 将 pt3（即 point3，点 3）赋值给控制值，使之与大写字母的顶部舍入高度

- 一致，并将它舍入到网格线上。
- (3) 将 pt9 赋值给控制值，使之与大写字符的顶部舍入高度一致，并将它舍入到网格线上。
 - (4) 控制 pt3 和 pt20 之间的距离，使之为大写字符 Y 方向的舍入控制值，或将 pt20 舍入到网格边界上。
 - (5) 控制 pt9 和 pt15 之间的距离，使之为大写字符 Y 方向的舍入控制值，或将 pt15 舍入到网格边界上。
 - (6) 将方向设置为 X 方向。
 - (7) 将 pt0 放在网格边界上。
 - (8) 控制 pt0 和 pt12 之间的距离，使之为大写字符 X 方向的舍入控制值，或将 pt12 舍入到网格边界上。
 - (9) 将 pt6 放在网格边界上。
 - (10) 控制 pt6 和 pt18 之间的距离，使之为大写字符 X 方向的舍入控制值，或将 pt18 舍入到网格边界上。
 - (11) 将尚未涉及到的点分别以 X、Y 方向添加到上面已处理了的点之间。

将上述的 Hinting 过程指令化，为如下代码：

```
SVTCA[Y]      /*将 Hinting 方向设置为 Y 方向*/
MIAP[R],3,3    /*将 pt3 赋给 cvt3，并进行舍入*/
MIAP[R],9,9    /*将 pt9 赋给 cvt9，并进行舍入*/
SRP0[],3       /*将当前参考点设为 pt3*/
MIRP[m>RB1],20,73 /*控制顶部舍入特征*/
SRP0[],9       /*将当前参考点设为 pt9*/
MIRP[m>RB1],15,73 /*控制低部舍入特征*/
SVTCA[X]      /*将 Hinting 方向设置为 X 方向*/
SRP0[],23      /*从左边界点开始 Hinting*/
MDRP[M<RWh], 0 /*对左边界点进行移动、舍入*/
MIRP[m>RB1],12,69 /*控制左边点的设入距离*/
SRP0[],24      /*从右边界点开始 Hinting*/
MDRP[M<RWh], 6 /*对右边的点进行移动、设入*/
MIRP[m>RB1],18,69 /*控制右边点的设入距离*/
IUP[X]         /*在 X 方向上添加未使用过的点*/
IUP[Y]         /*在 Y 方向上添加未使用过的点*/
```


5.4 字符光栅化

本节主要论述字符光栅化，即 Raster 化相关的内容。作为准备，先介绍了基本的多边形扫描线填充算法，贝塞尔曲线的基本知识和轮廓内外点判断的方法；在这些基础上，结合 TrueType Font 字体轮廓的特点详细论述了整个模块的实现。

Raster 化模块要实现的功能是根据 GRIDFITTING 模块得到的轮廓坐标和填充模式，填充字符轮廓，生成一个描述字符的数组，该数组和生成该字符的 Bitmap 位图有一一对应的关系。

5.4.1 多边形扫描线填充算法基础

对于每条横跨多边形的扫描线，填充算法确定扫描线与多边形一边的交点，然后将这些点从左到右的存储起来，并把每对交点间对应位置的像素点亮。

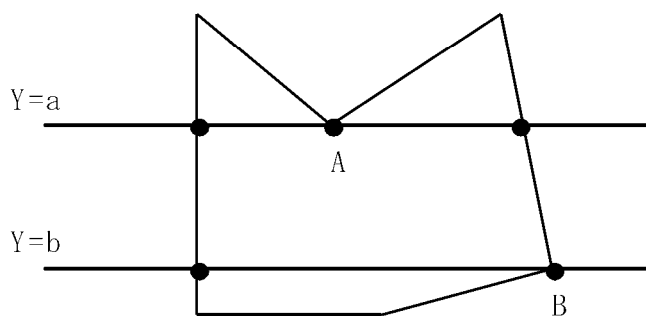


图 5-6 轮廓与扫描线的交点

5.4.1.1 顶点处理

需要注意的是，有些扫描线与多边形顶点相交，这种情况需要特殊处理。如图 5-6 所示，在有些位置上如 A 点，扫描线通过一个顶点与多边形的两条边相交，在这条扫描线的交点列表上要记入两个交点，但在有些位置上如 B 点，只需记入一个交点。根据多边形的拓扑结构，我们可以通过标识相交边相对于扫描线的位置来确定那些地方需要这样的额外处理。按顺时针或逆时针方向搜索多边形边界，并观察从一条边移动到另一条边时，顶点纵坐标的相对变化来判断需要什么样的额外处理。假如两条相邻边的端点纵坐标值单调递增或递减如 B 点，那么对于穿过该顶点的扫描线，则只能将该顶点记为一个交点；否则，共享顶点表示多边形边界上的一个局部极值如 A 点，这两条边与穿过该顶点的扫描线的交点应作为两个交点添加到交点列表中。

解决将顶点计为一个或两个交点的问题的一种方法是，将多边形的某些边缩

短，从而分离那些应计为一个交点的顶点。我们可以按照指定的顺时针或逆时针方向处理整个多边形边界上的非水平边。在处理每条边时进行检测，确定该边与下一条非水平边是否有单调递增或单调递减的端点 y 值。假如有，可以将较低的一条边缩短，从而保证对通过公共顶点（两条边的交点）的扫描线仅有一个交点产生。图 5-7 示例了一条边的缩短过程。当两条边的顶点 y 值增加时，将当前边的较高端点 y 值减去 1，如图 5-7 (a) 所示；当端点 y 值单调递减时，如图 5-7 (b) 所示，就减去紧随当前边的一条边的较高端点 y 值。

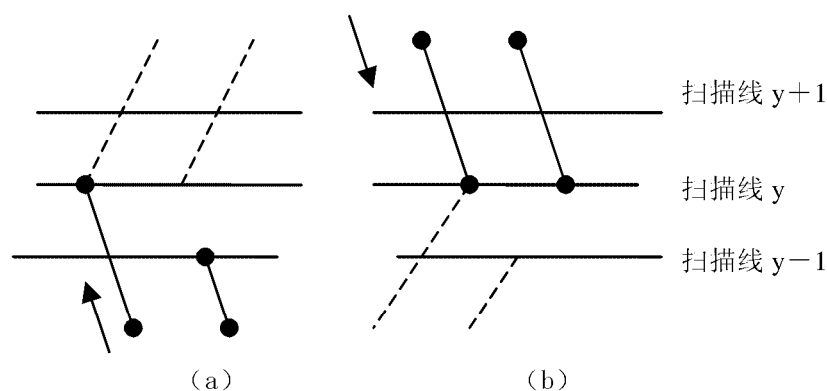


图 5-7 顶点在扫描线上的处理

5.4.1.2 连贯性的运用

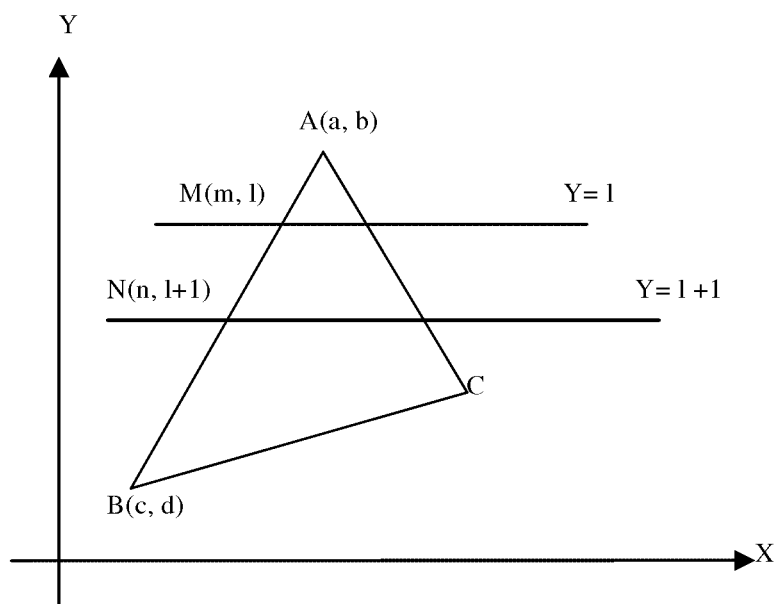


图 5-8 连贯性运用

在扫描转换及其他图形算法中完成的计算，经常利用待显示场景的各种连贯

性。这里所指的连贯性，可以简单地看做是场景中地一部分地特性以某种方式与场景中的其他部分相关，这种关系可以用来减少处理。连贯方法经常包括沿一条扫描线或在连续的扫描线间应用的增量计算。利用沿一条边从一条扫描线到下一条扫描线移动时，斜率是常数这一事实，我们在确定两边交点时可以采用增量坐标计算。图 5-8 示例了多边形右面一条边与两条扫描线连续相交的情况。

这条边的斜率 K 可以通过 $A(a, b)$ 、 $B(c, d)$ 点坐标计算出来：

$$K = \frac{b-d}{a-c} \quad (1)$$

设直线方程为：

$$y = Kx + B \quad (2)$$

则将交点 M 、 N 两点坐标分别带入 (2) 式得：

$$l = Km + B \quad (3)$$

$$l+1 = Kn + B \quad (4)$$

(4) 式减 (3) 式得：

$$1 = K(n - m)$$

即：

$$\Delta x = \frac{1}{K} = \frac{a-c}{b-d} = n - m \quad (5)$$

因此每个后继交点的 x 值都可以通过加 (5) 式所计算出来的值得到，当然这个值对于同一条边是不会该变的，所以简化了处理。

5.4.2 贝塞尔曲线

法国工程师 Pierre Bezier 使用这个样条逼近方法为雷诺设计汽车。贝塞尔样条有很多性质，可以更好地作用于曲线和曲面设计，并且更加方便，也更加容易实现。在 TrueType Font 字体中轮廓地曲线部分就是由二阶贝塞尔曲线描述的。二阶贝塞尔曲线由三个点确定，曲线起点 P_0 、终点 P_2 和一个控制点 P_1 ；控制点不在曲线上，起点终点在曲线上，其参数方程为：

$$P(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2 \quad 0 \leq t \leq 1$$

当 $t=1$ 时曲线为终点 P_2 ，当 $t=0$ 时曲线为起点 P_0 。

5.4.2.1 中间点坐标的计算

由于在曲线轮廓中有可能出现两个或两个以上的点都不在轮廓上的情况，所以在处理时我们要计算一些中间点的坐标。如图 5-9 所示的是连续两个点不在轮廓上的情况，点 P 就是我们要额外计算的中间点。

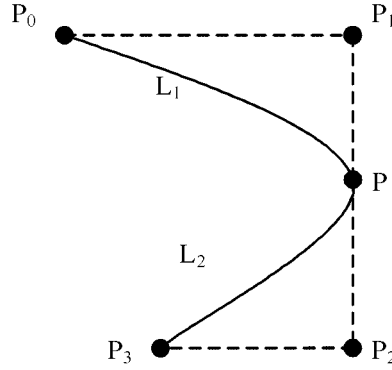


图 5-9 中间点

假设贝塞尔曲线 L_1 的参数方程是：

$$L_1(t_1) = (1-t_1)^2 P_0 + 2t_1(1-t_1)P_1 + t_1^2 P \quad 0 \leq t_1 \leq 1 \quad (1)$$

假设贝塞尔曲线 L_2 的参数方程是：

$$L_2(t_2) = (1-t_2)^2 P + 2t_2(1-t_2)P_2 + t_2^2 P_3 \quad 0 \leq t_2 \leq 1 \quad (2)$$

将 (1) 式对 t_1 求导，(2) 式对 t_2 求导，分别得 (3) 式和 (4) 式：

$$L'_1(t_1) = 2[(P_0 - 2P_1 + P)t_1 + P_1 - P_0] \quad 0 \leq t_1 \leq 1 \quad (3)$$

$$L'_2(t_2) = 2[(P - 2P_2 + P_3)t_2 + P_2 - P] \quad 0 \leq t_2 \leq 1 \quad (4)$$

由于 L_1 和 L_2 是组成同一轮廓的贝塞尔曲线，所以它们应该具有几何连续性，所以 L_1 和 L_2 在连接处，也就是中间点 P 处的一阶导数是相等的，既：

$$\begin{aligned} L'_1(1) &= L'_2(0) \\ \Rightarrow P_0 - 2P_1 + P + P_1 - P_0 &= P_2 - P \\ \Rightarrow 2P &= P_2 + P_1 \\ \Rightarrow P &= \frac{P_2 + P_1}{2} \end{aligned}$$

所以中间点 P 是 P_2 和 P_1 的中点，且 P 点在曲线上。对于连续两个以上的控制点不在轮廓上的情况，都可以转换为上面论述的这种情况，这样中间点的坐标就能够很容易的计算出来了。

5.4.2.2 贝塞尔曲线细分算法

由于光栅设备都是由离散的点组成的，所以对于任何曲线都要用直线进行逼近表示。细分算法对于显示逼近样条十分有用，因为它可以重复细分过程直到控制图形逼近曲线路径。然后将控制点坐标作为曲线位置，描画出整个图形^[41]。

图 5-10 表示了二阶贝塞尔曲线段中递归细分的第一步。

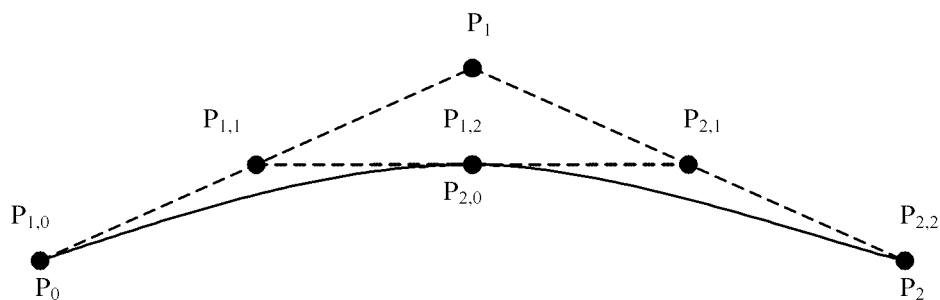


图 5-10 细分算法

贝塞尔曲线用参数点函数

$$P(u) = (1-u)^2 P_0 + 2u(1-u)P_1 + u^2 P_2 \quad 0 \leq u \leq 1$$

进行描述。在第一步细分中，利用中点 $P(0.5)$ 将原曲线分成两部分。

第一段函数记为：

$$P_1(s), \text{ 其中 } s = 2u \quad 0 \leq u \leq 0.5$$

第二段函数记为：

$$P_2(t), \text{ 其中 } t = 2u - 1 \quad 0.5 \leq u \leq 1$$

两条新曲线段与原始曲线段具有相同数目的控制点。每个新曲线段中，两个端点处的边界条件（位置和斜率）也必须与原始曲线段的位置和斜率相同。这为我们确定每个曲线段的控制点的位置给出了四个条件。对于第一部分曲线的三个新控制点是：

$$P_{1,0} = P_0$$

$$P_{1,1} = \frac{1}{2}(P_0 + P_1)$$

$$P_{1,2} = \frac{1}{4}(P_0 + 2P_1 + P_2)$$

对于后半条曲线，三个新控制点是：

$$P_{2,2} = P_2$$

$$P_{2,1} = \frac{1}{2}(P_2 + P_1)$$

$$P_{2,0} = \frac{1}{4}(P_0 + 2P_1 + P_2)$$

计算新控制点仅需要加法和移位（除 2）操作：

$$P_{1,0} = P_0$$

$$P_{1,1} = \frac{1}{2}(P_0 + P_1)$$

$$P_{2,2} = P_2$$

$$P_{2,1} = \frac{1}{2}(P_2 + P_1)$$

$$P_{2,0} = \frac{1}{2}(P_{2,1} + P_{1,1})$$

$$P_{1,2} = P_{2,0}$$

这些步骤可以重复递归多次，推出条件取决于是否需要进一步细分曲线以得到更多的控制点，或取决于是否要定位逼近曲线。一旦通过细分得到了一组显示点，则当曲线段很小时，可以结束细分过程。决定结束细分过程的一种方法是，检查每段中相邻控制点的距离。若这些距离“充分”小，我们可以停止细分；或者当每段的控制点几乎共线时，可以停止细分。

5.4.3 轮廓点内外点的判断

判断一个像素点是否在轮廓内部对于扫描转化是非常重要的。内部点的判断方法有两种，一种是较为简单的“奇偶判断法”，一种是对任何轮廓都适用的“线圈数（winding number）法”。

5.4.3.1 奇偶判断法

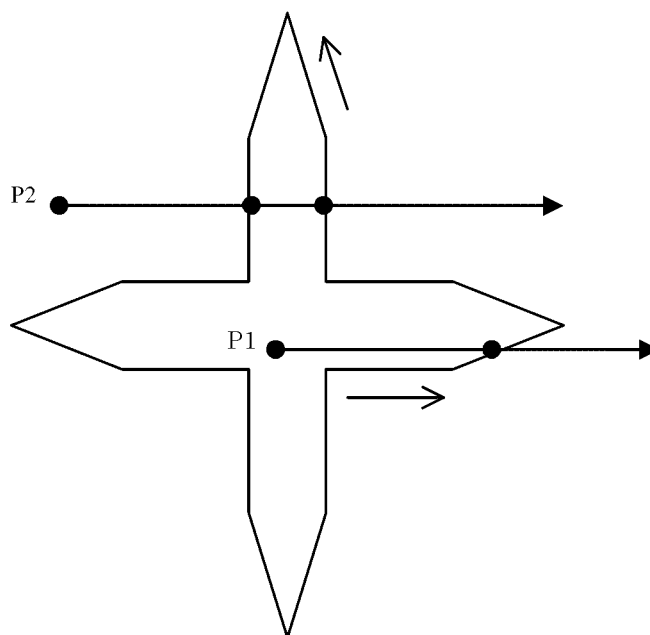


图 5-11 奇偶判断法

奇偶判断法只适用于轮廓没有交叉的 Font，对于轮廓有交叉的 Font 奇偶判断法不能正确判断内部点。判断方法如下：以判断点为端点，以任意方向，向无穷远处画一条射线；如果这条射线与轮廓的交点数为奇数，那么这个点在轮廓内部；

如果这条射线与轮廓的交点数为偶数，那么这个点不在轮廓内部。如图 5-11 是一个应用奇偶判断法来确定一个点是否在轮廓内的例子。图中轮廓与 P2 点引出的射线所得到的交点数为 2，所以 P2 点不在轮廓内；而轮廓与 P1 点引出的射线所得到的交点数为 1，所以 P1 点在轮廓内。

5.4.3.2 线圈数 (winding number) 法

线圈数 (winding number) 法适用于任何 Font 轮廓，包括轮廓有交叉的 Font。判断方法如下：一个点被认为是 glyph 的内部点，那么它线圈数 (winding number) 必为非零。winding number 是指：以该点为端点，以任意方向，向无穷远处画一条射线；winding number 从 0 开始计数，当 glyph 的轮廓从右向左或从下到上穿过该射线时 winding number 减 1，当 glyph 的轮廓从左向右或从上到下穿过该射线时 winding number 加 1，最后如果 winding number 不为零，则该点为 glyph 内部的点。轮廓的方向可以通过描述轮廓的点的序号来确定，轮廓的方向总是从低序号的点到高序号的点的。图 5-12 是一个应用 winding number 来确定一个点是否在轮廓内的例子。图中轮廓与 P2 点引出的射线所得到的 winding number 为 0，所以 P2 点不在轮廓内；而轮廓与 P1 点引出的射线所得到的 winding number 为 -1，所以 P1 点在轮廓内。

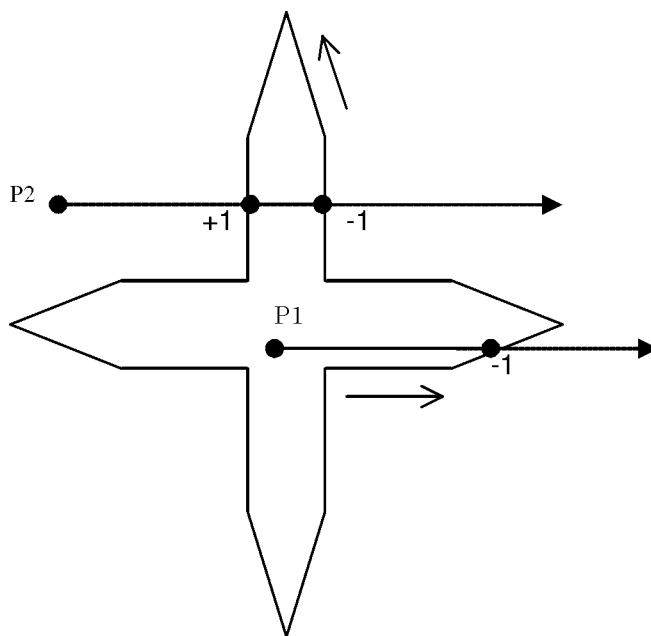


图 5-12 线圈数法

5.4.3.3 两种判断方法的比较

两种判断方法区别在于：“奇偶判断法”无法利用轮廓的方向信息，所以就不

能区别轮廓是从上还是从下，从左还是从右与判断点引出的射线相交。如图 5-13 所示，现在要判断 P 点是否在轮廓内。用“奇偶判断法”P 点引出的射线与轮廓的交点数为 2，所以 P 点应该不在轮廓内；用“线圈数法”轮廓与 P 点引出的射线的两次相交都是从下向上穿过射线，所以 winding number 都要减 1，最后得到的 winding number 为-2，即 P 点在轮廓内。显然正确的结果是“线圈数法”得到的 P 点在轮廓内。

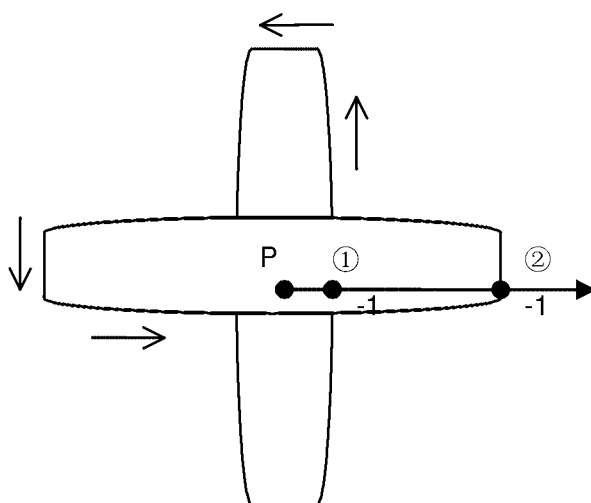


图 5-13 两种判定方法的比较

5.4.4 Raster 化模块的实现

由于轮廓中的曲线是由二阶贝塞尔曲线描述的，所以在填充时，我们要先以直带曲，将曲线转化为短的线段来近似的代替。这个转换的过程其实就是一次用描述字符轮廓的点将字符包含的各个轮廓“虚拟描画”的过程，之所以叫做虚拟描画是因为，并没有真正的将轮廓显示在显示设备上，而只是将描述轮廓的点按顺序“走”了一遍。通过第一次“虚拟描画”过程我们把轮廓转化为仅由长短线段描述，而且这些线段的顶点也按顺序来存储在一个数组中的。我们在填充轮廓时，就依次成对的从这个数组中取出线段顶点坐标，第二次“虚拟描画”轮廓一遍。描画的目的在于求轮廓与扫描线的交点坐标和交点属性。为了提高填充精度，且简化对水平和竖直线段的处理，我们在第二次“虚拟描画”时分两步来“虚拟描画”轮廓，一步求与水平扫描线的交点，一步求与竖直扫描线的交点；再分别根据这两组坐标来填充轮廓；最后以这两次填充的并集作为最后的结果。整个过程如图 5-14 所示。

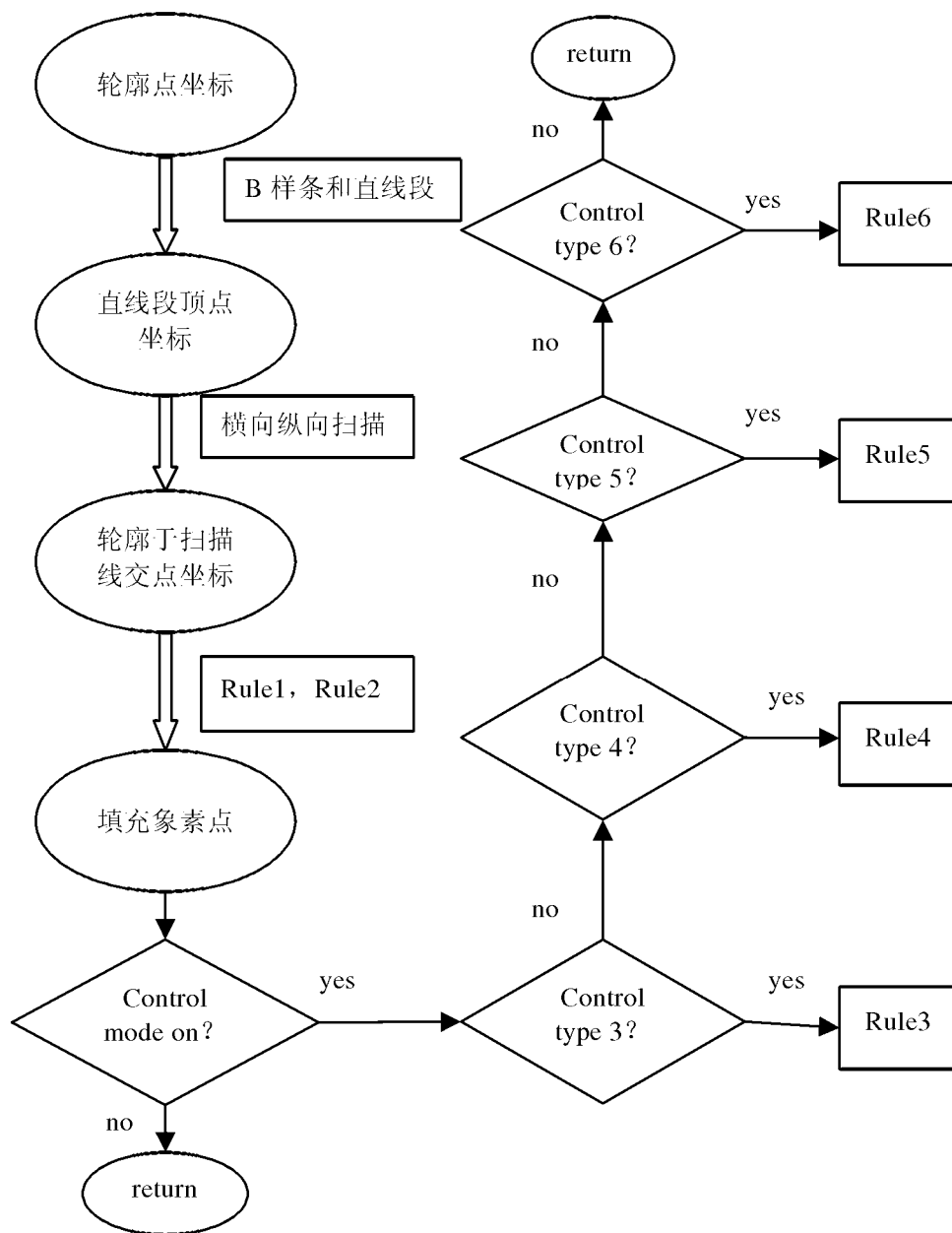


图 5-14 RasterImage 模块内部结构

代码如下：

```

BOOL TTF_RasterImage(VOID *p)
{
    CTTFFontFileInfo* pTTFontFileInfo = (CTTFFontFileInfo *)p;
    CGLYFDATA *pGlyfData = (CGLYFDATA*)(pTTFontFileInfo->m_pGlyfData);
    CGridfitting *pGirdfitting = (CGridfitting*)(pTTFontFileInfo->m_pGirdfitting);

```

```

USHORT    usNumOfContours = pGlyphData->m_iNumContours;
USHORT    usNumOfPoints = pGlyphData->m_iPoints;
int* piEndPoints = pGlyphData->m_pEndPoints;
g_nOffset.x = 0;
g_nOffset.y = 0;
g_usScanLineNum = pTTFFontFileInfo->m_PixelRow;
int N = pTTFFontFileInfo->m_PixelRow * pTTFFontFileInfo->m_PixelCol;
memset(pTTFFontFileInfo->m_pPixelArray,0x0,N);
PBYTE pbBitmap = pTTFFontFileInfo->m_pPixelArray;
BOOL bConMode = pTTFFontFileInfo->m_bConvertMode;
LONG nConType = pTTFFontFileInfo->m_byConvertType;
// base on scanline: X = n
if( g_xScanline)
{
    delete []g_xScanline;
    g_xScanline = NULL;
}
g_xScanline = new SCANLINE[g_usScanLineNum];
for(int i = 0; i <= g_usScanLineNum - 1; i++)
{
    g_xScanline[i].nCur = 0;
    for(int j = 0; j < MAX_CROSSNUM; j++)
    {
        g_xScanline[i].Coor[j].FDCoordinate = -1.0;
        g_xScanline[i].Coor[j].usOutlineNum = -1;
        g_xScanline[i].Coor[j].usCrossPtNum = -1;
        g_xScanline[i].Coor[j].usMaxCroPtNum = 0;
        g_xScanline[i].Coor[j].sDirection = 0;
    }
}
if( g_yScanline)
{

```

```

        delete []g_yScanline;
        g_yScanline = NULL;
    }
    // base on scanline: Y = n
    g_yScanline = new SCANLINE[g_usScanLineNum];
    for(int i = 0; i <= g_usScanLineNum - 1; i++)
    {
        g_yScanline[i].nCur = 0;
        for(int j = 0; j < MAX_CROSSNUM; j++)
        {
            g_yScanline[i].Coor[j].FDCoordinate = -1.0;
            g_yScanline[i].Coor[j].usOutlineNum = -1;
            g_yScanline[i].Coor[j].usCrossPtNum = -1;
            g_yScanline[i].Coor[j].usMaxCroPtNum = 0;
            g_yScanline[i].Coor[j].sDirection = 0;
        }
    }

    /* store all the points which are on the outline ,including the points which are
    produced when draw Bezier */
    g_nMaxSize = 500;
    g_ProcessedCoorNum = 0;
    if( g_ProcessedRasterCoor)
    {
        delete []g_ProcessedRasterCoor;
        g_ProcessedRasterCoor = NULL;
    }
    g_ProcessedRasterCoor = new ProcessedPoint[g_nMaxSize];
    for(int i = 0; i < g_nMaxSize; i++)
    {
        g_ProcessedRasterCoor[i].F26D6Point.x = -2.0;
        g_ProcessedRasterCoor[i].F26D6Point.y = -2.0;
        g_ProcessedRasterCoor[i].bBegPoint = FALSE;
    }

```

```

//record the points which are on the outline (including line and Bezier)
GetPointsOnOutline(pGlyphData);
/* record the cross points between outline and scanline X=n, the cross points are
stored in g_xScanline */
RecordXCrossPoints(pGlyphData);
/* record the cross points between outline and scanline Y=n, the cross points are
stored in g_yScanline */
RecordYCrossPoints(pGlyphData);
/* base on the g_xScanline and g_yScanline fill the array pByte which is used by
bitmap*/
FillPixelArray(bConMode, nConType, pbBitmap);
pTTFFontFileInfo->m_nProcessedCoorNum = g_ProcessedCoorNum;
pTTFFontFileInfo->m_ProcessedOutlineCoor = g_ProcessedRasterCoor;
/* used by EXE for displaying the information of crosspoints between outlines and
scanlines */
pTTFFontFileInfo->m_xScanline = g_xScanline;
pTTFFontFileInfo->m_yScanline = g_yScanline;
pTTFFontFileInfo->m_usScanLineNum = g_usScanLineNum;
return TRUE;
}

```

5.4.4.1 第一次“虚拟描画”

如前面章节所提到的，glyph 表包含了 TrueType Font 文件中所有字符的数据。glyph 表由简单 glyph 子表和复合 glyph 子表组成，Glyph 子表的个数可以从 maxp 表中读取。复合 glyph 子表描述的是几个简单 glyph 如何组成复合 glyph，所以归根结底描述字符轮廓的点的坐标信息是存放在简单 glyph 子表中的。在简单 glyph 子表中与描画轮廓直接相关的有三个数组：flags 数组、以及 xCoor、yCoor 数组。在 TrueType Font 中组成轮廓线条的是直线和曲线的连接，其中曲线是由二阶贝塞尔曲线描述的。二阶贝塞尔曲线由三个点描述，起点，终点和一个控制点；显然，起点和终点都在曲线上，而控制点不在曲线上。flags 数组中的每一项的第 0bit 就是对某点在不在曲线上的描述；若第 0bit 为 1 则该点在轮廓上，若第 0bit 为 0 则该点不在轮廓上。描述轮廓的点的坐标在 xCoor、yCoor 数组中的排列是绕轮廓依次有序排列的，序号由小到大，同时也就描述了轮廓的方向，所以将这些点按规则依次连接起来就能描画出该轮廓。还是以前面提到的 Arial 字体的“1”以 6pt

的磅值,在 150dpi 的显示设备上显示为例,它在经 GRIDFITTING 模块进行 Hinting 后各点的坐标(单位为 64 倍像素)将变为:

(256, 0) (192, 0) (192, 500) (170, 539) (96, 421) (64, 406)
(64, 482) (119, 513) (200, 600) (216, 640) (256, 640)

将这些点遵循下面的规则连接起来就是如图 5-15 的样子:

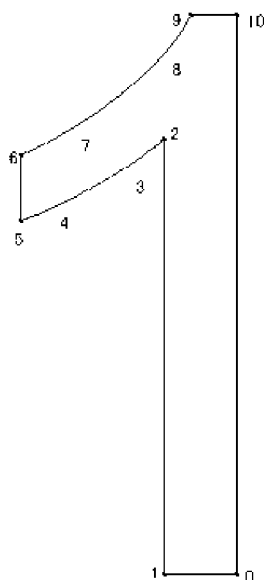


图 5-15 arial 字体的 l

(1) 从起点开始逐点连接;也就是从点 0 开始。

(2) 若点 i 在轮廓上,并且点 $i+1$ 也在轮廓上,那么用直线将它们连接起来。点 0 到点 1, 点 1 到点 2 都是这样。

(3) 若点 i 在轮廓上,点 $i+1$ 在轮廓外,那么这时候采用贝塞尔曲线连接。点 i 被当作贝塞尔曲线的起点,点 $i+1$ 被当作贝塞尔曲线的控制点,此时再读点 $i+2$ 的信息,若

1) 点 $i+2$ 在轮廓上:此时点 $i+2$ 被当作贝塞尔曲线的终点,和点 i , $i+1$ 一起描画出整个贝塞尔曲线;

2) 点 $i+2$ 在轮廓外:此时隐含的取点 $i+1$ 和点 $i+2$ 的中点作为附加点 j ,并以点 j 作为贝塞尔曲线的终点,和 i , $i+1$ 一起描画出一条贝塞尔曲线(此时曲线到点 j 为止)。同时,以点 j 为起点,点 $i+2$ 为控制点,再读入点 $i+3$ 的信息……如此循环,直到发生(1)的情况。点 2 到点 5 就是这样,实际上点 3 和点 4 中间有一个隐含的贝塞尔曲线的终(起)点。这是 2 段贝塞尔曲线的结合。

(4) 若起点 i 就在轮廓外,那么若点 $i+1$ 在轮廓上,则以 $i+1$ 为起点:若点 $i+1$ 也

在轮廓外，则以点 i 和 $i+1$ 的中点为起点，也是贝塞尔曲线的起点，同时以 $i+1$ 为控制点，成为 (3) 的情形。不管哪种情况，点 i 都会作为这个轮廓的最后一点来处理。（相当于绕了一个圈）

我们事先将存储顶点的数组 `g_ProcessedRasterCoor[]` 的每一项都初始化为 $(-2, -2)$ ；用直线直接连接两个描述点时就将这两个点的坐标直接存入顶点数组 `g_ProcessedRasterCoor[]`，用贝塞尔曲线连接描述点时就按照上面介绍的贝塞尔曲线细分算法，将分割后的线段顶点存入顶点数组 `g_ProcessedRasterCoor[]`；当描画完该字符的一个轮廓后在存储顶点的数组中加一个标志点 $(-1, -1)$ 。这样将该字符的所有轮廓的描述点都“走”完一次后，第一次“虚拟描画”也就完成了，该过程由函数 `GetPointsOnOutline(pGlyphData)` 实现，产生顶点数组 `g_ProcessedRasterCoor[]`；接下来就是用存储在 `g_ProcessedRasterCoor[]` 中的顶点来进行第二次“虚拟描画”了。

5.4.4.2 第二次“虚拟描化”

这一步的目的就是求字符各个轮廓与纵横扫描线之间的交点，亦即一些在下一步填充过程中需要的交点的信息。由于我们是虚拟描画，所以所描画的直线不用考虑反走样等在光栅设备上显示直线时所要注意的问题。另一方面我们是分两步来分别求字符各个轮廓与纵横扫描线之间的交点，所以对于水平扫描线 $Y=n$ ，不用处理和水平线段相交的问题，而且每个交点的纵坐标都是 n ；同理对于竖直扫描线 $X=n$ ，也不用处理和竖直线段相交的问题，而且每个交点的横坐标都是 n 。这样一来，每一次只需依此从顶点数组中取出两个顶点来和扫描线求交点就可以了，而且这个求交点的过程和数学上求交点的过程是一样的，是不会产生误差的。下面以竖直扫描线 $X=n$ 为主线进行论述。

以扫描线填充算法为基础，建立一个表。此表为一个以 `SCANLINE` 结构为类型的数组，数组下标为扫描线的编号，例如：

```
SCANLINE g_xScanline[xSize];
```

`SCANLINE` 结构有两个域，最重要的是一个 `XYCOORDINATE` 类型的数组 `Coor`，还有一个域为 `nCur`，指示数组 `Coor` 实际存放了多少个元素，即此扫描线与该字符各个轮廓总共的交点数。

```
typedef struct tagSCANLINE
{
    SHORT      nCur;
```

```

XYCOORDINATE  Coor[MAX_CROSSNUM];
}SCANLINE,*PSCANLINE;

```

XYCOORDINATE 类型又有五个域，FDCoordinate 存放的是竖直扫描线与轮廓交点的纵坐标；由于一个字符可能由多个轮廓组成，所以用 usOutlineNum 来表示轮廓号，标识该交点是那个轮廓与扫描线相交产生的；sDirection 记录产生该交点的轮廓与扫描线相交时的方向信息；usCrossPtNum 表示该交点是该轮廓和扫描线的第几个交点；usMaxCroPtNum 表示该轮廓和扫描线一共有多少个交点。

```

typedef struct tagXYCOORDINATE
{
    F26D6      FDCoordinate;
    SHORT      sDirection;
    SHORT      usOutlineNum;
    SHORT      usCrossPtNum;
    USHORT     usMaxCroPtNum;
}XYCOORDINATE,*PXYCOORDINATE;

```

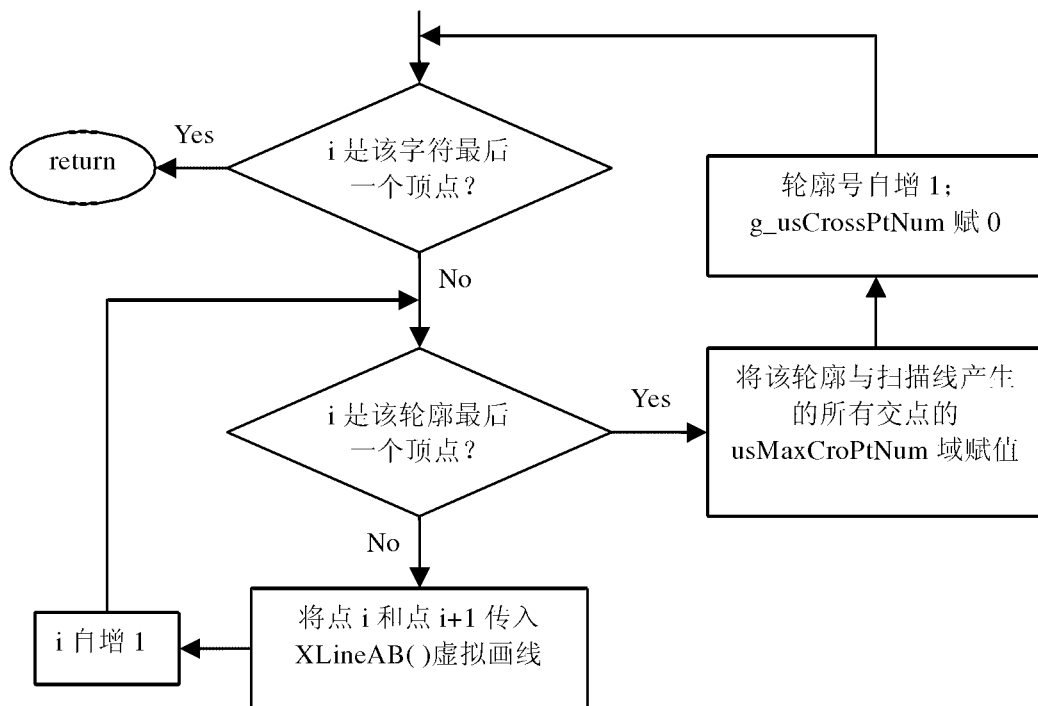


图 5-16 第二次虚拟描画流程图

sDirection 将用于轮廓内外点的判断，它仅有-1 和+1 两个值，对于竖直扫描线，

在虚拟描画轮廓线段时，如果轮廓由左往右穿过扫描线，则将 sDirection 的值设为 +1，反之为 -1。UsOutlineNum、usOutlineNum 和 usMaxCroPtNum 三个值将用于对像素丢失的控制。主要流程如图 5-16 所示：

现在假设我已经从顶点数组 g_ProcessedRasterCoor[] 中取得了两个顶点的坐标 A(A.x, A.y)、B(B.x, B.y)，方向是 A 到 B。那么就调用虚拟画线函数 XLineAB 完成接下来的工作，虚拟画线函数 XLineAB() 的算法如下：

- (1) 若线段 AB 平行于扫描线，则返回依次取下面的点，否则进入下一步；
- (2) 若线段 AB 和扫描线相交则进入下一步，否则返回依次取下面的点；
- (3) 若 B 点 B.x 为整数，则 B 点在扫描线上，为避免顶点被重复计算为交点，将 B.x 减 1；
- (4) 判断线段 AB 和扫描线相交的方向，给 sDirection 赋值；若 B.x > A.x 则 sDirection 被赋为 +1，若 B.x < A.x 则 sDirection 被赋为 -1，且交换 A、B 两点的坐标；
- (5) 计算交点坐标，给 Coor[i] 中各数据项赋值；

值得说明的是，数组 Coor 的实际大小即 nCur 值绝对应该是一个偶数，因为扫描线和轮廓的交点总是成对出现的，nCur 值为 0 则表明该扫描线与轮廓无交点。

5.4.4.3 填充与像素丢失控制模式

在第二次虚拟描画后，得到了轮廓和扫描线的交点。接下来就是利用这些交点根据相关规则填充轮廓生成 Bitmap 位图。

以竖直扫描线的编号为横坐标，Coor[i].Coordinate 为纵坐标的点为可能需要填充的区域起点；以竖直扫描线的编号为横坐标，Coor[i+1].Coordinate 为纵坐标的点为可能需要填充的区域终点。具体该区域是否应该填充还需要利用 sDirection 的值根据线圈数法则进行判断。假设可能需要填充的区域为 Coor[i].Coordinate 到 Coor[i+1].Coordinate。那么累加 Coor[i].sDirection (i 从 0 到 i)，如果最后得到的值为非零则该区域需要填充，如果为零则该区域不能填充。最后建立一个与 bitmap 位图坐标点一一对应的数组 BYTE bFontArray [sArraySize]；数组中的值要么为 0 要么为 1。若 bFontArray[(Y-1) * xSize + X] (或 bFontArray[(X-1) * xSize + Y]) 为 1 则 bitmap 位图中的点(X,Y)被点亮。

在填充过程中我们采用更加细致的规则来控制像素丢失。规则如下：

Rule1：如果像素中心在轮廓内，点亮该像素。

Rule2：如果像素中心在轮廓上，点亮该像素。

Rule3: 如果同一轮廓的两条边和连接两个相邻像素中心点的水平或竖直扫描线都相交，且两个像素都没有被 Rule1 和 Rule2 点亮，那么将最左边（对于水平扫描线）或最下面（对于竖直扫描线）的像素点亮。

Rule4: 在 Rule3 的基础上，为了避免“stubs”，如果同一轮廓的两条边和连接两个相邻像素中心点的水平或竖直扫描线都相交，且继续和其它的水平扫描线或竖直扫描线相交，且两个像素都没有被 Rule1 和 Rule2 点亮，那么将最左边（对于水平扫描线）或最下面（对于竖直扫描线）的像素点亮。

Rule5: 如果同一轮廓的两条边和连接两个相邻像素中心点的水平或竖直扫描线都相交，且两个像素都没有被 Rule1 和 Rule2 点亮，那么像素中心靠近轮廓两条边中点的像素点亮。

Rule6: 在 Rule5 的基础上，为了避免“stubs”，如果同一轮廓的两条边和连接两个相邻像素中心点的水平或竖直扫描线都相交，且继续和其它的水平扫描线或竖直扫描线相交，且两个像素都没有被 Rule1 和 Rule2 点亮，那么像素中心靠近轮廓两条边中点的像素点亮。

Rule1 和 Rule2 完成基本的填充功能；Rule3 和 Rule5 实现不带“stubs”判断的像素丢失控制；Rule4 和 Rule6 实现带“stubs”判断的像素丢失控制；Rule4、Rule6 比 Rule3、Rule5 对于填充左边还是右边（或上面还是下面）的像素有了更加科学的判断。如图 5-14，程序具体实现时先根据 Rule1 和 Rule2 进行基本判断，完成基本填充之后再根据控制模式选择 Rule3 到 Rule6 中的一条规则来进行像素丢失控制。如图 5-17(a-d)是利用 Rule3 到 Rule6 填充同一轮廓时产生的不同效果。

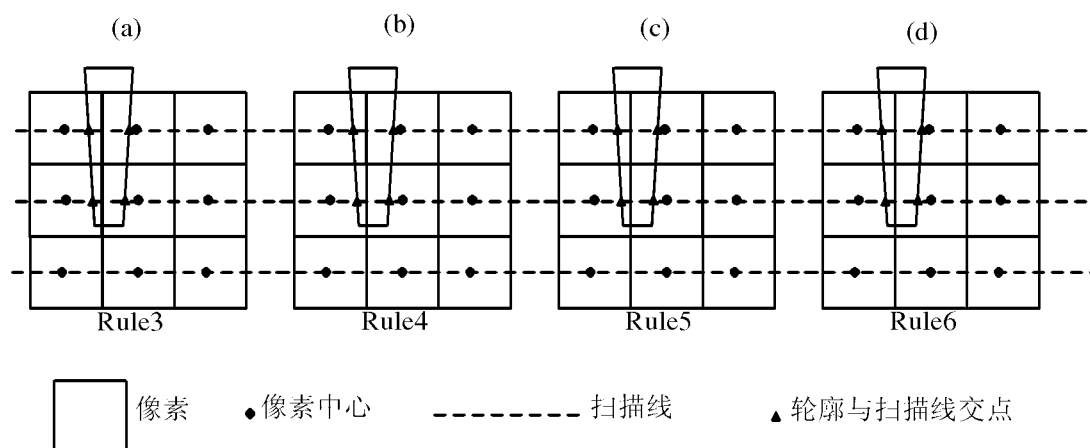


图 5-17 Rule3 到 Rule6 的填充效果

在根据 Rule4 和 Rule6 判断“stubs”的情况时用到了 usOutlineNum、usCrossPtNum 和 usMaxCroPtNum 这三值。对于 Rule6 整个代码实现具体如下：

```

// base on xScanline X=n
for(int x = 0; x < g_usScanLineNum; x++)
{
    if(g_xScanline[x].nCur != 0)
    {
        if((g_xScanline[x].nCur % 2) != 0)
            return;
        else
        {
            for(int i = 0; i+1 < g_xScanline[x].nCur; i++)
            {
                int nWindingNum = 0;
                for(int j = i; j >= 0; j--)
                    nWindingNum += g_xScanline[x].Coor[j].sDirection;
                if(0 != nWindingNum)
                {
                    ptBeg.y=(LONG)ceil(g_xScanline[x].Coor[i].FDCoordinate);
                    ptEnd.y=(LONG)floor(g_xScanline[x].Coor[i+1].FDCoordinate);
                    if((ptBeg.y > ptEnd.y) && (pbBitmap[nLine * ptBeg.y + x] != 1) &&
                        (pbBitmap[nLine * ptEnd.y + x] != 1))
                        // "ptBeg.y > ptEnd.y" means the distance between BeginPoint and EndPoint is
                        // shorter than 1 pixel. so there is a possible dropout point.
                    {
                        if(g_xScanline[x].Coor[i].usOutlineNum!=
                            g_xScanline[x].Coor[i+1].usOutlineNum)
                        {
                            int y = int(g_xScanline[x].Coor[i].FDCoordinate);
                            if(((g_xScanline[x].Coor[i].FDCoordinate*64+g_xScanline[x].Coor[i+1].FDCoordinate*64)/2-int(g_xScanline[x].Coor[i].FDCoordinate)*64)>32)
                                y = y + 1;
                            pbBitmap[nLine * y + x] = 1;          //SetPixel(x , y)
                        }
                    }
                }
                else
                {
                    int temp=abs(g_xScanline[x].Coor[i].usCrossPtNum-g_xScanline[x].Coor[i+1].usCrossPtNum);
                    if((temp != 1) && ((temp + 1) != g_xScanline[x].Coor[i].usMaxCroPtNum))
                    {
                        int y = int(g_xScanline[x].Coor[i].FDCoordinate);
                        if(((g_xScanline[x].Coor[i].FDCoordinate*64+g_xScanline[x].Coor[i+1].FDCoordinate*64)/2-int(g_xScanline[x].Coor[i].FDCoordinate)*64)>32)
                            y = y + 1;
                        pbBitmap[nLine * y + x] = 1;          //SetPixel(x , y)
                    }
                }
            }
        }
    }
}

```

```

    }
  }
}
}

```

根据 Rule6 要实现像素丢失控制，我们要判断几个事实。

首先，由于相邻的两个像素点的中心连线要和同一轮廓的两条边相交，所以这两个交点的距离肯定比 1 小，而且这两个像素要都没有被点亮，这是我们要首先判断的既：

```

ptBeg.y=(LONG)ceil(g_xScanline[x].Coor[i].FDCoordinate);
ptEnd.y=(LONG)floor(g_xScanline[x].Coor[i+1].FDCoordinate);
if((ptBeg.y > ptEnd.y) && (pbBitmap[nLine * ptBeg.y + x] != 1) &&
    (pbBitmap[nLine * ptEnd.y + x] != 1))

```

其次，要判断这两个交点是否是同一个轮廓产生的交点。若不是同一个轮廓，因为已经通过线圈数法则的判定：

```

for(int j = i; j >= 0; j--)
    nWindingNum += g_xScanline[x].Coor[j].sDirection;
if(0 != nWindingNum)

```

所以填充离轮廓中心近的那个像素；若是同一个轮廓则要进一步判断是 dropout 而不是 stub。

根据 Rule4 或 Rule6: dropout 满足轮廓的两条边和连续的扫描线相交。而 stub 是轮廓的两条边穿过相邻像素中心点的连接线段（既扫描线）以后马上相交所产生的，所以这个轮廓的这两条边和该扫描线产生的这两个交点一定是该轮廓上两个相邻的交点。也就是说，这两个交点的 usCrossPtNum 值分别与该轮廓的 usMaxCroPtNum 值取模后的差为 1。所以对于 dropout 应满足的条件是：

```

int temp = abs(g_xScanline[x].Coor[i].usCrossPtNum - g_xScanline[x].
    Coor[i+1].usCrossPtNum);
if((temp != 1) && ((temp + 1) != g_xScanline[x].Coor[i].usMaxCroPtNum))

```

若是 dropout 则填充离轮廓中心近的那个像素。

第六章 结论

一个字符要打印在纸面上或者显示在显示器上，大概要经过三个过程。首先是 Scale，即要把存储在 TrueTypeFont 文件中的描绘该字符 glyph 的轮廓放缩成需要的尺寸。实际上就是将描述轮廓的点进行坐标变换。变换后的点都坐标不再是以 FUnit 为用户空间中的坐标，而是以像素为单位的设备空间中的坐标。然后是 Gridfit，即通过指令将描述轮廓的点的坐标进行微调。解释程序要执行关于该字符 glyph 指令，这些指令是用来使字符更好的在光栅设备中显示。最后是 Raster，即将 GridFit 的轮廓填充后生成 Bitmap 显示在光栅设备上。

为达到以上目的，在详细研究了 TrueTypeFont 技术，TrueType Font 文件的基本构成和一些背景知识的基础上，分析和研究了 TrueType Font Engine 的基本结构；并以此入手，提出了一种独立于微软字体引擎的，可用于打印机驱动程序的 TrueTypeFont Engine 设计模型；并最终实现了以上设计。

作为项目小组成员，作者参与了项目的前期设计和资料分析；然后具体负责设计和实现了光栅化模块。

在光栅化模块的设计实现中，由于 TrueType Font 字体文件结构特殊，它是以控制点来描述字符轮廓，而控制点是存放在 TrueType Font 字体文件的 glyf 表中的，所以作者在研究了轮廓内外点判断方法的基础上，将传统的多边形扫描线填充算法进行了合理地改进，最后提出了一种适用于 TrueType Font 字符轮廓的可填充交叉轮廓的扫描线填充算法。它的特点是根据 TrueTypeFont 字体轮廓的特殊描述方式——用控制点来描述字体轮廓——提出了纵横双向扫描的扫描方式。纵横双向扫描不仅避免了对水平线段和垂直线段的特殊处理；而且还可以提高字体轮廓的填充精度。该填充算法还有一个特点就是在“虚拟描画”时记录交点坐标，更重要的是它还同时记录了字体轮廓与扫描线相交的方向信息，这使它能很有效的通过线圈数法实现轮廓内外点判断，具有填充交叉轮廓的功能。

致 谢

首先要衷心的感谢我的导师张建中副教授。作为张老师的硕士研究生，我得到了张老师在学习和工作等各方面的指导，使我受益匪浅。他严谨的治学态度，一丝不苟的工作作风，都是我学习的榜样。

此外，我还要感谢我所在的公司，为我的毕业设计和其他的项目的实践提供了一个良好的学习工作环境，使我有机会接触更多的专业知识。

最后，还要感谢和我一起工作的同事、同学和朋友们，他们的帮助和配合使我顺利地完成了毕业设计。

参考文献

- [1] 徐福培等. 页面描述语言及其程序设计. 南京大学出版社, 1994
- [2] Adobe System Inc. PostScript Language Program Design. 1988
- [3] 孙家广, 杨长贵. 计算机图形学(新版). 北京: 清华大学出版社, 1995
- [4] 郭平欣, 张淞芝编. 汉字信息处理技术. 国防工业出版社, 1984
- [5] Adobe System Inc. Adobe Type 1 Font Format. Addison-Wesley, 1990
- [6] Adobe System Inc. CID_ Keyed Font Technology Overview. 1994
- [7] Adobe System Inc. The OpenType Font File
- [8] Microsoft Crop. TrueType Font Files. Microsoft Crop, 1995
- [9] 胡长原, 张福炎. 基于 Type 1 格式的曲线轮廓汉字的 hinting 技术, 1996
- [10] Adobe System Inc. Type 1 Font Format Supplement. 1994
- [11] Ph. Coueignoux. Character Generation by Computer. Computer Graphite and Image Processing, Vol. 16, 1981
- [12] Adobe System Inc. PostScript Language Reference Manual. 1. Addison-Wesley, 1985
- [13] 柳朝阳, 李叔梁. 压入区段端点的区域填充扫描线算法. 计算机辅助设计与图形学学报, 1996, 1(2): 15-25
- [14] 马辉, 陆国栋, 谭建荣, 等. 基于顶点与邻边相关性的多边形填充算法. 计算机图象图形学报, 2004, 11(9): 1336-1342
- [15] Microsoft corporation. TrueType Font Files. 学苑现版社, 1990, 79-211
- [16] 陈西垚, 陈伟, 佟丽芳. 通用扫描线填充算法存在的问题及其解决方法. 东北电力大学学报, 2006, 26(6): 52-54
- [17] 杨长强, 彭延军, 郑永果. 一种封闭 B 样条曲线的扫描线填充算法. 系统仿真学报, 2006, 18: 12-13, 17
- [18] 吴海辉, 樊庆林, 王虎. TrueType 字体技术的研究分析与应用. 电脑知识与技术(学术交流), 2007, 3: 97-112
- [19] 刘勇奎, 石教英. 曲线的整数型生成算法. 计算机报, 1998, 21(3): 270-280
- [20] 杨建红, 刘蓉, 余泽太. TrueType 字体在图形图像处理软件中的应用. 武汉大学学报(工学版), 2004, 6
- [21] 马自勤, 孔宪庶. CAPP 系统中特殊字符串处理技术的研究. 机械设计与制造, 2003, (2)

- [22] 吕强, 史磊. TrueType 字体文件格式初探. 计算机研究与发展, 1995, 32 (11)
- [23] TrueType 1.0 Font Files Technical Specification (Revision 1.66).
<http://www.microsoft.com>, 1995-08
- [24] 晏志军, 黄翔. 基于 iMAN 的 CAPP 系统的研究与实现. 机械制造与自动化, 2002, (3)
- [25] 段峰, 段伟, 王耀南, 等. 单片机巧用 Windows 矢量字库. 现代电子技术, 2001(3):55~56
- [26] 张正华, 杜宇人. 在 VB 下智能仪器与 PC 机之间的数值通信. 电子工程师, 2001, 27(1):3-4
- [27] 何立民. MCS-51 系列单片机应用系统设计系统配置与接口技术. 北京: 北京航空航天大学出版社, 1990. 157~160
- [28] 舒忠梅, 胡金柱, 左亚尧. TrueType 字体中文字指令化技术剖析. 微计算机信息, 1998, 14(5):57-59
- [29] Microsoft. TrueType Open Font Specification. 1995-07
- [30] Mulder S. Embedding Fonts Tutorial. <http://hotwired.lycos.com>
- [31] 叶以民, 孙玉方. Chinese TrueType Font Support in X Window. Journal of Computer Science and Technology, 1999, 14(1):27-33
- [32] 黄源, 王瑜. TrueType 汉字字形编辑器的设计与实现. 计算机研究与发展, 1998, 35(5):423-425
- [33] 舒忠梅, 左亚尧. 浅析 TrueType 中的 Hinting 原理及相关技术. 计算机应用研究, 1999, (7):25-30
- [34] 王瑜, 黄源. Windows 中 TrueType 字形数据的存取技术. 小型微型计算机系统, 1997, 18(11):75-81
- [35] 肖明, 胡金柱, 赵慧. 字形技术及 OpenType 字体文件格式研究. 中文信息学报, 1999, 13(6):53-60
- [36] TURNER D. FreeType project [EB/OL]. <http://www.freetype.org>. 2005-03-14
- [37] 章仁江, 王国瑾. 有理 Bézier 曲线离散终判准则的改进. 软件学报, 2003, 14(10):501-504
- [38] GROLEAU T. Proof of Casteljau method. 2002
- [39] 陈效群, 陈发来, 陈长松. 有理曲线的多项式逼近. 高校应用数学学报, 1998, 13(A):23-29
- [40] 黄宜华, 袁春风. 曲线轮廓汉字字形缩放与还原中的几个问题的研究. 中文信息学报, 1995, 9(2):28-36
- [41] 周天祥, 杨勋年, 汪国昭. 快速绘制 Bézier 曲线. 计算机辅助设计与图形学学报, 2002, 14(6):501-504
- [42] SHAMIR A, RAPPOPORT A. Quality Enhancements of Digital Outline Fonts. Computers & Graphics, 1997, 21(6):713-725

攻硕期间取得的研究成果

攻硕期间发表的论文：

[1] 刘翔, 张建中. 打印机驱动程序中 Truetype Font Engine 的研究与设计. 电子科技大学研究生学报计算机科学与技术增刊. 30 期: 38-42

电子科技大学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

硕士学位论文

MASTER DISSERTATION