中国科学技术大学

University of Science and Technology of China

本科毕业论文

A Dissertation for the Bachelor's Degree

SEM 模拟成像中 3D 模型的制作及在 Si 蚀刻线宽测量中的应用

The Method of 3D Model Making for the Simulation of SEM Imaging and its Application in the Measurement of Si Lines

姓　　名	汪 玉鹤
B.S. Candidate	Yuhe Wang
导　　师	丁泽军 教授
Supervisor	Prof. Zejun Ding

二〇一二年六月

June,2012

中国科学技术大学

学士学位论文

题　　目	SEM 模拟成像中 3D 模型的制作及在 Si 蚀刻线宽测量中的应用
院　　系	少年班学院
姓　　名	汪 玉鹤
学　　号	PB08000668
导　　师	丁泽军 教授

二〇一二年六月

University of Science and Technology of China

A Dissertation for the Bachelor's Degree

The Method of 3D Model Making for the Simulation of SEM Imaging and its Application in the Measurement of Si Lines

B.S. Candidate Yuhe Wang

Supervisor Prof. Zejun Ding

Hefei, Anhui 230026, China

June,2012

致　谢

谨在此向所有帮助过我的人致以我最真诚的谢意！

首先要感谢丁泽军教授。丁老师不仅上课幽默风趣，让我在计算物理课上愉快地学到了丰富的专业知识，而且平日里待人热情豪爽，让实验室里所有人都感觉轻松自在，工作充满动力。当然丁老师的严谨治学也让人称道，故近水楼台先得月，我也深受大家风范的熏陶。

其次要感谢张鹏师兄的细心指导和特别关照。张鹏师兄为人和蔼谦虚，在我研究陷入困境时总是耐心地加以指点，不仅教会我使用服务器进行大型计算的知识，而且屡次帮我查找程序错误，连论文答辩也要多次关心叮嘱。我的一点一滴的进步，都离不开张师兄的热情帮助。张鹏者，不显山露水之高人也！

再次，要感谢科大超算中心为我的理论计算提供平台支持，祝愿所有在超算中心做计算的同行都能有意外的发现。

最后，感谢给我最多无私关爱的父母和朋友们，你们的支持是我永远的动力！

目 录

中文内容摘要

针对 SEM 模拟成像中任意复杂 3D 模型的构建问题，本文提出了一个有效的解决方案，即利用现在市场上非常成熟的 3D 建模软件——3DS MAX 构建快速三维模型，然后转换成模拟程序所能理解的格式。为此本文分析了 3ds 文件的格式，编写了转换程序 ConvertModel，并且制作了一些 SEM 镜头下常见的模型如花粉、芯片线路进行模拟，得到的模拟图像和实验观察到的图像相差无几。然后本文将这种模拟研究法应用在了 Si 蚀刻线宽度测量中，并在转换程序中加入了粗糙度处理，使模型的表面更接近实物表面，计算得出了测量修正值和粗糙度的初步关系。实践证明，3DS MAX+ConvertModel+模拟程序这一套工具大大缩短了研究耗费时间，并将研究对象拓展到更广泛的领域，因此具有重大的意义。

关键字：SEM 成像模拟，3D 建模，3DS MAX，Si 蚀刻线

Abstract

Aimed at solving the problem of how to construct 3D models with complex geometric structure in the simulation of SEM imaging, this thesis brought forward a complete and effective solution, which is to use the very matured 3d model buiding software 3DS Max to construct the models and then convert its files into the format that our simulation software can understand. So we analyzed the 3ds format(3DS MAX' s file format), wrote a converting software ConvertModel, and made some models that are commonly seen under SEM such as pollen and chips lines to get the simulation imagine, which turned out to be quite close to those real ones. In addition, we applied this method into the research of the exact measurement of Si lines, and modified the model as rough as real Si line to obtain the relationship of the roughness and the correction. These applications prove that the toolkit of 3DS MAX+ ConvertModel+Simulation Software can reduce the workload vastly, and extend our research objects to a wider field. So this work is of great significance.

Key Words：SEM Imaging Simulation, 3D Model Making, 3DS MAX, Si Lines

第一章 绪论

第一节 SEM 简介

一、SEM 成像分析

利用扫描电子显微镜研究各种材料几乎已经成为所有材料学研究中必不可少的步骤。其最主要依赖于从样品表面出射的二次电子或背散射电子作为成像的信号来源。对于二次电子（能量<50 eV），由于其出射时从样品不同地方的产额不一样，比如同等实验条件下在平坦地方的产额小于倾斜的地方，倾斜的地方又小于凹凸的地方等。这些特点使二次电子信号提供了样品表面形貌的信息。再次，二次电子出射的区域在表面只有直径约纳米量级的范围，所以分辨率也较高。

背散射电子由于能量较高可以到达固体中较深的内部，以及经受大角度的弹性散射（与原子势相关），对于提供的样品表面的形貌信息并不能像二次电子那样细致。但是能够提供更多样品的信息比如成分等，并且分辨率没有二次电子的好，这是由于出射范围较大的缘故。

对于提供样品表面的形貌信息，虽然二次电子成像具有很多优点，但是也只局限于提供衬度信息，并不能够给出被测样品的几何结构信息，比如高度，横向维度等。纵然在图像上可以通过明暗的分界线看出样品的二维的边界，但这是不准确的，因为受电子束与固体相互作用机制的限制，直接看出的界限并不是真实的边界，比如 CD-SEM(测长扫描电子显微镜) 测量梯形半导体门的宽度得出的图像在边界出会出现 bloom 区域，这就难以判断真实的边界。所以怎样定被测样品的真实几何结构参数是一个较难以解决的问题。

二、CD-SEM

近年来，测长扫描电子显微镜(CD-SEM)对于纳米装置制造业中精确的度量评估起到重要作用，尤其在半导体工业中对纳米级刻蚀线宽测量有着很大的优越性。CD-SEM 除了具有常规的 SEM 的特点之外，且拥有很高的分辨率和大的吞吐量，可以实施在线监测和线下检测。但是，通常利用 CD-SEM 获得的二次电子图像确定构件的线宽会有很大的不确定性，比如，构件的 top-down 面扫描图像或线扫描图像的亮边或双峰的距离与实际构件的线宽有着差别。这些差别会随

着构件的几何参数（高度，边墙角等）以及入射电子束的束宽而变化，所以，相应线宽确定算法的提出至关重要。目前已有的线宽测定算法都有各自的利与弊。比如，对于较大体系（>100 nm）线宽的确定，常用几种经验的算法：最大导数法、递归基线算法和 S 形拟合算法等，但对小到几十纳米的线宽范围，就不再适用，且没有物理基础。Novikov 等人提出一种建立模型构型与 SEM 线扫描曲线的特殊点（峰与谷）间的简单对应关系的方法称为临界形状度量算法[1]。这种方法比较简单易行，但对于小边墙角情况误差比较大。Frase 等人给出了 e 指数拟合方法，对于大的线宽尺度比较适用，但对于小尺寸和小边墙角情况就不再适用[2,3]。此外还有 Tanaka 等人基于实验和 Monte Carlo 模拟结果提出的多参数侧面标定算法，但需要引入修正[4,5,6]。

第二节 SEM 成像的模拟

目前扫描电子成像的模拟同样在材料学的研究中起着不可替代的作用，它不仅有助于理解图像衬度形成机理以及电子束与固体相互作用的机制，同时也提供了一个建立扫描图像与样品结构，实验参数等联系的平台。如果可以构建一系列已知几何参数的样品模型，通过模拟方法得到相应的扫描电子显微镜(SEM)图像并与实验图像对比，且模拟的结果与实验测量有很好的吻合性，那么这项工作会具有很重要的意义。首先，由于事先构建的一系列样品的几何结构参数已知，最为接近实验图像的样品结构参数就可以近似认为是真实样品的结构参数。其次，对 SEM 分辨率的评估有一定的指导意义，因为理论上已知构件与构件之间的距离，再通过对模拟的 SEM 图像应用相关算法测量构件之间的距离，通过比较就可以对分辨率做一定程度的评估。

然而，进行这样的模拟是一件很困难的工作，特别是样品几何结构较为复杂的情况。过去的许多年，基于蒙特卡罗(MC)模拟的理论研究一直不断的改进模拟 SEM 图像，已经模拟出很多较为简单的几何结构的样品的图像。针对目前的研究都是集中在简单且规则的几何结构上，我们决定计算复杂的 SEM 模拟图像，这项研究一来国际同行都没有进行过，二来如果研究成功不仅可以如上所说测量真实结构体的参数，评估电镜分辨率，而且将有助于扫描电子显微学界建立标准，构建数据库等。

对于 CD-SEM 的模拟，目前被认为最精确的算法是由 Villarrubia 等人引入的

一种基于模型的数据库方法[7,8,9]，即利用 MC 方法建立各种相关参数与模拟结果间的对应数据库。但是这种方法需要非常大的计算量并且要有比较精确的理论模型。在此，我们运用精确的 MC 模型模拟电子与固体的相互作用与级联的二次电子产生过程，为了更加接近实际的测量条件引入粗糙表面来试图建立一种基于模拟的数据库算法。模拟程序是利用 Fortran 和 MPI 编写的并行程序，粗糙表面由三角形网格法生成并且运用空间分割的算法加速运算[10,11,12,13]。

第三节 问题的提出

一、3D 建模工具

基于三角形网格模型和空间分割加速算法的成像模拟程序已由本实验室构建完成，能够精准地模拟任意规则和不规则几何体的成像。图 1-3-1 和图 1-3-2 是一些模拟图像，能够较好地反映实验结果。其中，规则几何体的模拟适用于规则晶体的研究，而非规则模型的模拟成像具有更加广泛的用途。

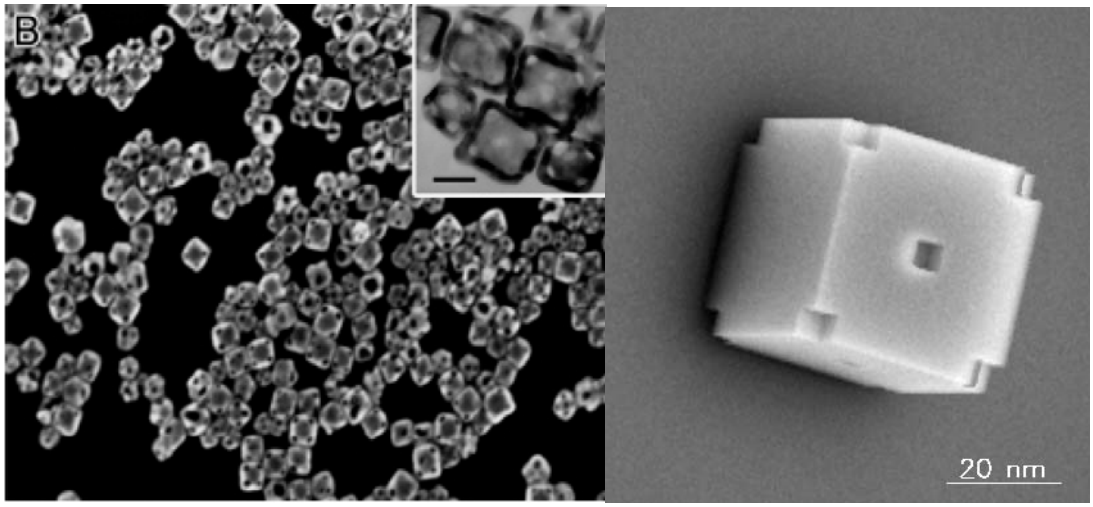

图 1-3-1 规则晶体的实验（左）和模拟图像（右）对照

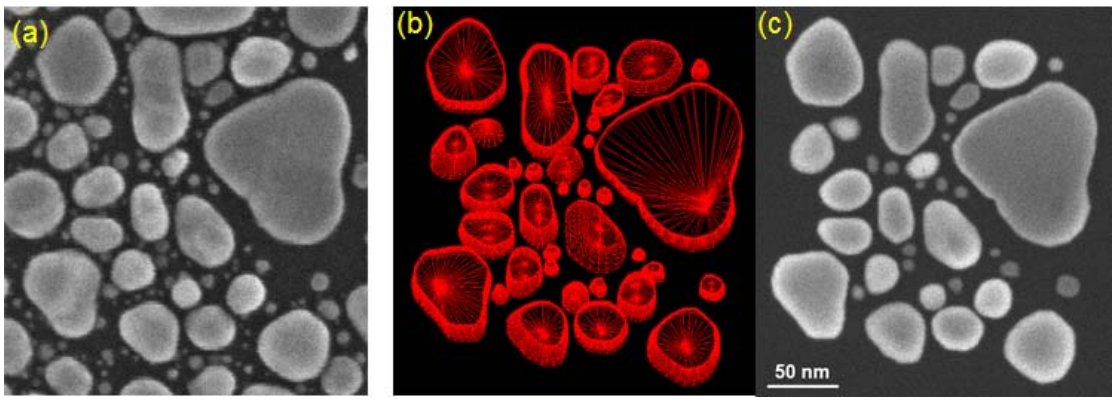

图 1-3-2 Au/C 系统非规则几何体的实验图(a)和模拟图像(b)(c)对照

然而，成像模拟在操作上并非十分简单。首先我们需要根据实验得出的 SEM 图像建立一个封闭的 3D 三角网络，尽量和实验图像一致；然后根据模型的大小、网格划分的程度设置参数，在大型计算机上运行模拟程序，得到模拟的图像；接着将模拟图像和实验图像对比，修改模型形状，反复进行模拟对照，直到两张图像吻合度最高为止。这样，模型的几何结构就精确地反映了实验样品颗粒的几何形状。

此流程最困难的是如何构建和修改 3D 的三角网格模型。对于规则几何体，通过简单的计算即可得出三角形坐标；但是对于非规则的几何体（无法用简单的解析式表达），人工计算坐标的效率极其低下，形状的修改更是难以控制。虽然图 XX 较好地展示了非规则几何体的模拟成像，但也耗费了一个多月的时间才最终将模型完工。

所以寻找或编写一款构建 3D 模型的有力工具成为了当务之急。在暑期大学生研究计划中，我模仿 3ds max 编写过一款叫 ModelMaker 的小软件，实现了 3ds max 的一部分建模功能，界面如图 1-3-3 所示。只需简单地输入数值和拖放，就可以构建很多非规则的 3D 模型。相比起来，也有一些优点，如表 1-3-1。但毕竟时间精力有限，实现的功能模块很少，在构建复杂模型时效率就比较低，而且网格剖分也不够优化。由于编写 3D 的功能模块非常复杂，需要很多人合作，我便改换思路，他山之玉可以攻石，尝试编写一款软件，将 3ds max 的模型文件转换成我们所需的文件格式。这样就可以利用 3ds max 强大的建模功能，迅速构建出更多复杂模型进行计算模拟。具体工作参见第二章。

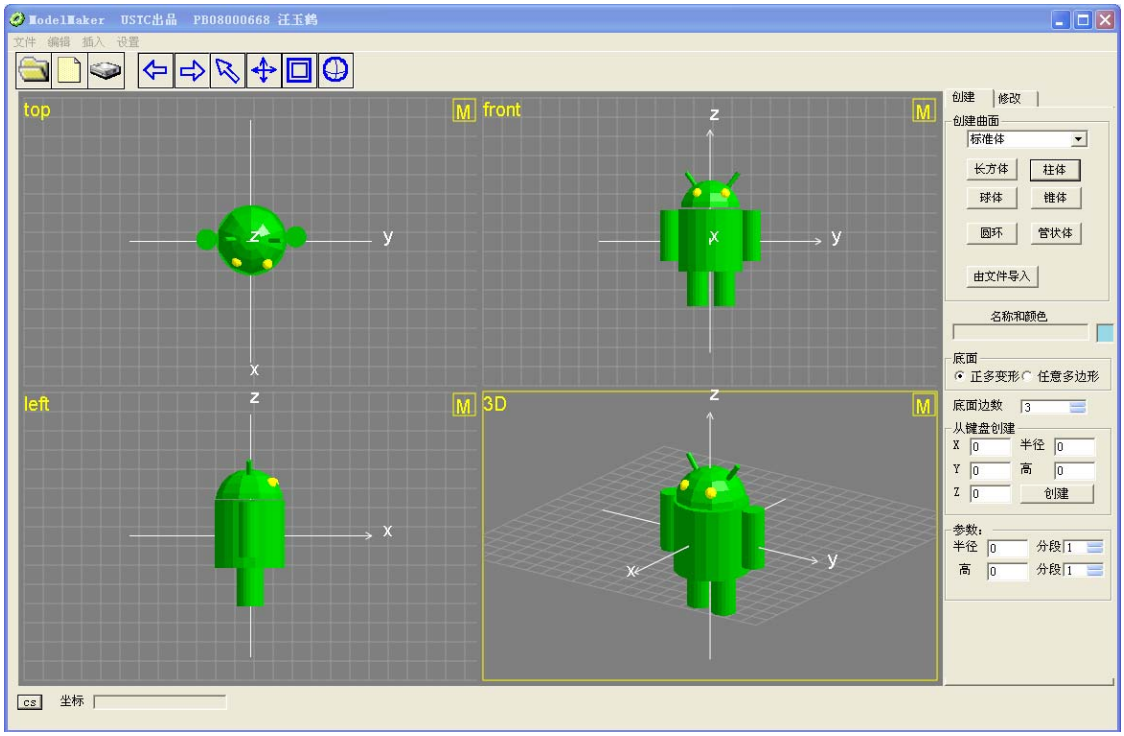

图 1-3-3 ModelMaker 界面

表 1-3-1 ModelMaker 和 3ds max 的一些比较

特点	3DS MAX 建模软件	针对实验室研发的 ModelMaker
功能	很强大	目前功能有限
价格	价格昂贵	免费
易用性	学习耗时	容易上手
建模方式	不仅仅用三角形描述	全部用三角形描述
封闭性	未必封闭	保证封闭
可定制性	难以定制所需的功能	容易扩展所需功能

二、线宽的精确测量

对于提供样品表面的形貌信息，虽然二次电子成像具有很多优点，但是也只局限于提供衬度信息，并不能够给出被测样品的几何结构信息，比如高度，横向维度等。注意，虽然在图像上可以通过明暗的分界线看出样品的二维的边界，但这是不准确的！受电子束与固体相互作用机制的限制，直接看出的界限并不是真实的边界，而是有一个微小的偏差，且此偏差会随着构件的几何参数以及入射电

子束的束宽而变化。

对于比较大的构件（100nm 以上），此偏差基本可以忽略。然而对于小尺寸构件，此偏差就逐渐变得不可忽略。随着半导体芯片行业的不断发展，晶体管的尺寸越做越小，最新的 Intel Ivy 架构线宽甚至小到 22nm。此时 CD-SEM 测得的尺寸和实际尺寸间的偏差将十分显著，必须进行修正。然而，真实的尺寸难以通过实验准确获知，研究这种偏差更是无从下手。

虽然目前实验研究比较困难，但理论模拟研究却是简便易行。我们可以构建精确尺寸的模型，然后模拟 CD-SEM 的成像过程，比较计算所得的线宽和真实的线宽。这样不仅可以建立起修正偏差的数据库供工业使用，而且可以为研究偏差的论理公式提供更多参考，具有十分重要的意义。

在模拟研究中，我们还需要考虑真实的构件表面粗糙度对线宽测量的影响。构造表面粗糙的 3D 模型并非难事，只需将每个格点沿着法线移动即可，移动的距离用高斯分布描述，比较符合实际情况。

鉴于半导体工业的重要性，我们将重点对 Si 蚀刻线宽的偏差修正展开研究。具体内容请参见第三章。

第二章 3ds 模型转换软件的研发

第一节 3ds 文件格式

一、3DS MAX

3DS MAX 原名：3D Studio Max)是由 AudoDesk 公司开发的全功能的三维计算机图形软件，界面图 2-1-1 所示。由于其强大的 3d 建模功能，在广告、影视、工业设计、建筑设计、多媒体制作、游戏、辅助教学以及工程可视化等领域有着极广泛的应用，屡屡在国际上获得大奖。

许多游戏开发者也常常利用 3DS MAX 建模，可是 3DS MAX 导出的文件并非游戏开发者想要的格式。他们是怎么将模型应用到游戏中呢？原来 3ds max 可以把数据导出为一种名叫 3ds 格式的文件，虽然 Autodesk 一直没有公布 3ds 文件的结构信息，但是也基本被破解出来了大半。只要掌握了 3ds 文件的结构，那么就可以随意读取需要的那部分数据。

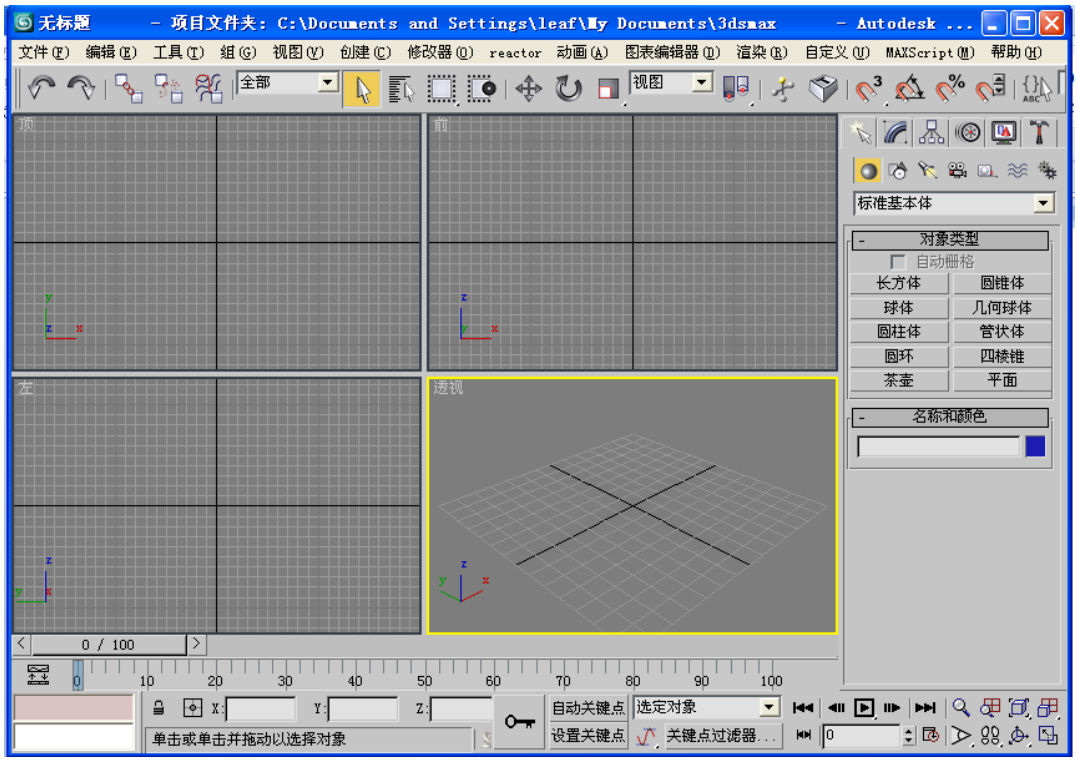

图 2-1-1 3DS MAX 建模程序界面

二、3ds 文件结构详解[14]

```
MAIN3DS  (0x4D4D)
  |
  +--EDIT3DS  (0x3D3D)
  |  |
  |  +--EDIT_MATERIAL (0xAFFF)
  |  |  |
  |  |  +--MAT_NAME01 (0xA000)
  |  |
  |  +--EDIT_CONFIG1  (0x0100)
  |  +--EDIT_CONFIG2  (0x3E3D)
  |  +--EDIT_VIEW_P1  (0x7012)
  |  |  |
  |  |  +--TOP           (0x0001)
  |  |  +--BOTTOM        (0x0002)
  |  |  +--LEFT          (0x0003)
  |  |  +--RIGHT         (0x0004)
  |  |  +--FRONT         (0x0005)
  |  |  +--BACK          (0x0006)
  |  |  +--USER          (0x0007)
  |  |  +--CAMERA        (0xFFFF)
  |  |  +--LIGHT         (0x0009)
  |  |  +--DISABLED      (0x0010)
  |  |  +--BOGUS         (0x0011)
  |  |
  |  +--EDIT_VIEW_P2  (0x7011)
  |  |  |
  |  |  +--TOP           (0x0001)
  |  |  +--BOTTOM        (0x0002)
  |  |  +--LEFT          (0x0003)
  |  |  +--RIGHT         (0x0004)
  |  |  +--FRONT         (0x0005)
  |  |  +--BACK          (0x0006)
  |  |  +--USER          (0x0007)
  |  |  +--CAMERA        (0xFFFF)
  |  |  +--LIGHT         (0x0009)
  |  |  +--DISABLED      (0x0010)
  |  |  +--BOGUS         (0x0011)
  |  |
  |  +--EDIT_VIEW_P3  (0x7020)
  |  +--EDIT_VIEW1    (0x7001)
  |  +--EDIT_BACKGR   (0x1200)
  |  +--EDIT_AMBIENT  (0x2100)
  |  +--EDIT_OBJECT   (0x4000)
  |  |  |
  |  |  +--OBJ_TRIMESH   (0x4100)
  |  |  |  |
  |  |  |  +--TRI_VERTEXL        (0x4110)
  |  |  |  +--TRI_VERTEXOPTIONS  (0x4111)
  |  |  |  +--TRI_MAPPINGCOORS   (0x4140)
  |  |  |  +--TRI_MAPPINGSTANDARD(0x4170)
  |  |  |  +--TRI_FACEL1         (0x4120)
  |  |  |  |  |
  |  |  |  |  +--TRI_SMOOTH     (0x4150)
  |  |  |  |  +--TRI_MATERIAL   (0x4130)
  |  |  |  |
  |  |  |  +--TRI_LOCAL       (0x4160)
  |  |  |  +--TRI_VISIBLE     (0x4165)
  |  |  |
  |  |  +--OBJ_LIGHT    (0x4600)
  |  |  |  |
  |  |  |  +--LIT_OFF         (0x4620)
  |  |  |  +--LIT_SPOT        (0x4610)
  |  |  |  +--LIT_UNKNWN01    (0x465A)
  |  |  |
  |  |  +--OBJ_CAMERA   (0x4700)
  |  |  |  |
  |  |  |  +--CAM_UNKNWN01    (0x4710)
  |  |  |  +--CAM_UNKNWN02    (0x4720)
  |  |  |
  |  |  +--OBJ_UNKNWN01 (0x4710)
  |  |  +--OBJ_UNKNWN02 (0x4720)
  |  |
  |  +--EDIT_UNKNW01  (0x1100)
  |  +--EDIT_UNKNW02  (0x1201)
  |  +--EDIT_UNKNW03  (0x1300)
  |  +--EDIT_UNKNW04  (0x1400)
  |  +--EDIT_UNKNW05  (0x1420)
  |  +--EDIT_UNKNW06  (0x1450)
  |  +--EDIT_UNKNW07  (0x1500)
  |  +--EDIT_UNKNW08  (0x2200)
  |  +--EDIT_UNKNW09  (0x2201)
  |  +--EDIT_UNKNW10  (0x2210)
  |  +--EDIT_UNKNW11  (0x2300)
  |  +--EDIT_UNKNW12  (0x2302)
  |  +--EDIT_UNKNW13  (0x2000)
  |  +--EDIT_UNKNW14  (0xAFFF)
  |
  +--KEYF3DS (0xB000)
     |
     +--KEYF_UNKNWN01 (0xB00A)
     +--............. (0x7001)
     +--KEYF_FRAMES   (0xB008)
     +--KEYF_UNKNWN02 (0xB009)
     +--KEYF_OBJDES   (0xB002)
        |
        +--KEYF_OBJHIERARCH  (0xB010)
        +--KEYF_OBJDUMMYNAME (0xB011)
        +--KEYF_OBJUNKNWN01  (0xB013)
        +--KEYF_OBJUNKNWN02  (0xB014)
        +--KEYF_OBJUNKNWN03  (0xB015)
        +--KEYF_OBJPIVOT     (0xB020)
        +--KEYF_OBJUNKNWN04  (0xB021)
        +--KEYF_OBJUNKNWN05  (0xB022)
```

图 2-1-2-1　3ds 文件结构树

3ds 文件结构是由“块”(chunk)组成的。它们描述了接在它们后面的数据的信息，即这些数据是如何组成的。“块”是由两部分组成的：

(1)ID，表明这个块包含的是什么类型的数据

(2)下一个数据块的位置。也就是说，如果你不明白这个块的用处，你可以迅速地跳过它。因为下一个数据区的相对位置（字节数）已经得到了。

其数据结构如下：

```
unsigned short ID;        //2 个字节，无符号的，块的 ID
unsigned long Length;     //4 个字节，无符号的，描述了下一个数据区对于当前数据区的位置，实际上就是本数据区的大小。
```

每个块是一个层次结构，由 ID 表示。3ds 文件有一个主块，ID 是 0x4D4D。这个块永远是 3ds 文件的开始，可以用它来鉴定本文件是否为一个 3ds 文件。在开始块里面就是主要块了，块与块之间是嵌套分布的。图 XXX 显示了不同 ID 的块以及它们在文件中的位置。这些 ID 都被#define 命名，便于看出其代表的意义。从图 2-1-2-1 中可看出，3ds 文件包含了很多的信息，包括材质、视角、三角网格、灯光、摄像机、动画帧等，而在模拟 SEM 成像时，只需要三角网格的信息即可。3ds 文件的块结构允许我们迅速跳过不关心的块，只读取其中的三角网格块信息，效率很高。这样我们需要处理的块就精简为图 2-1-2-2，相当简洁。

```
MAIN3DS   (0x4D4D)
  |
  +--EDIT3DS   (0x3D3D)
  |  |
  |  +--EDIT_OBJECT    (0x4000)
  |  |  |
  |  |  +--OBJ_TRIMESH    (0x4100)
  |  |  |  |
  |  |  |  +--TRI_VERTEXL           (0x4110)
  |  |  |  +--TRI_FACEL1            (0x4120)
```

图 2-1-2-2 精简的 3ds 文件块

第二节 C++类 CLoad3DS

对于编程实现，最好采用 C++类封装的方式，将 3ds 数据读取功能封装成一个模块，这样便于移植到其他程序中。首先，考虑对外的接口函数，需要有一个导入函数 bool Import3DS(CString strFileName)和一个清除函数 void CleanUp()，另外构造函数也需要设计成CLoad3DS(CString strFileName)便于直接用文件名构建此类。CLoad3DS 用到了 vector 类模板，对外的数据成员 vector<t3DObject> vObject 提供网格数据的访问。同时，我还加入了纹理数据的读取，使得 CLoad3D

适用性更广。类设计如下：

```
class CLoad3DS
{
public:
    CLoad3DS();                                 // 空的构造函数
    CLoad3DS(CString strFileName);              // 构造时已载入模型
    ~CLoad3DS();                                // 析构函数
    bool Import3DS(CString strFileName);// 装入 3ds 文件到模型结构中
    void CleanUp();                             // 清除模型，以便下次载入

    vector<tMaterialInfo> vMaterials;           // 材质信息向量容器
    vector<t3DObject> vObject;                  // 模型信息向量容器

private:
    //去除重复冗余的网格数据，便于粗糙处理
    void OptimizeVertex();
    // 读一个字符串
    int GetString(char *);
    // 读下一个块
    void ReadChunk(tChunk *);
    // 读下一个块
    void ProcessNextChunk(tChunk *);
    // 读下一个对象块
    void ProcessNextObjectChunk(t3DObject *pObject, tChunk *);
    // 读下一个材质块
    void ProcessNextMaterialChunk(tChunk *);
    // 读对象颜色的 RGB 值
    void ReadColorChunk(tMaterialInfo *pMaterial, tChunk *pChunk);
    // 读对象的顶点
    void ReadVertices(t3DObject *pObject, tChunk *);
    // 读对象的面信息
    void ReadVertexIndices(t3DObject *pObject, tChunk *);
    // 读对象的纹理坐标
    void ReadUVCoordinates(t3DObject *pObject, tChunk *);
    // 读赋予对象的材质名称
    void ReadObjectMaterial(t3DObject *pObject, tChunk *pPreviousChunk);
    // 计算对象顶点的法向量
    void ComputeNormals();

    // 文件指针
    FILE *m_FilePointer;
    tChunk *m_CurrentChunk;
    tChunk *m_TempChunk;
};
```

3ds 文件的层次结构使得数据读取显得较为简单，流程如图 2-2-1 所示。

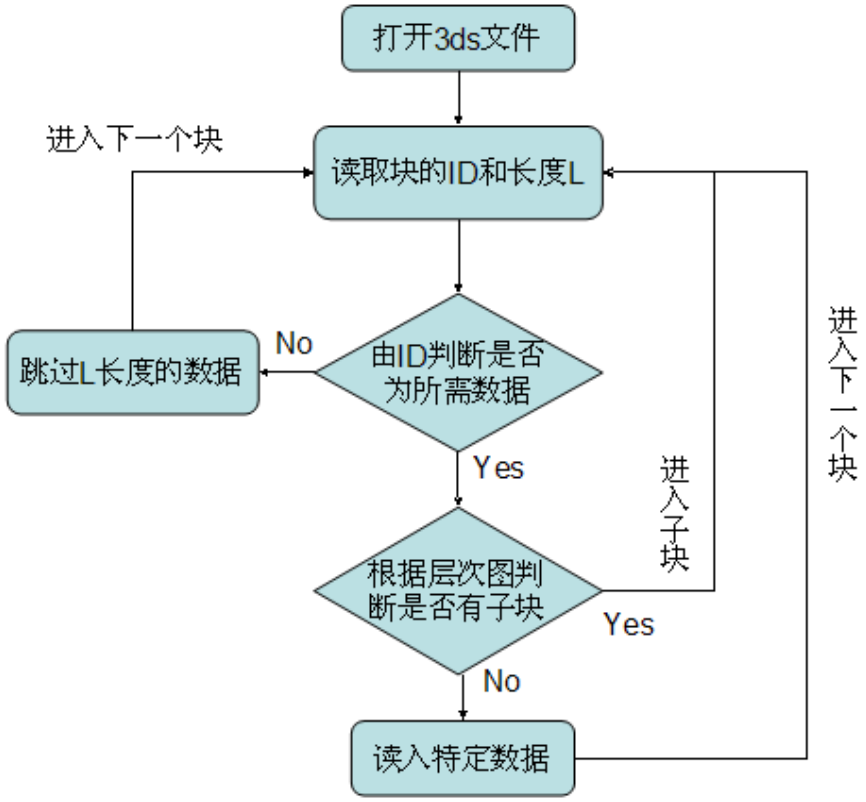

图 2-2-1　3ds 文件读取流程图

对于这样的流程图，由于要不断进入子块，子块和父块的逻辑又相似，用递归函数实现最为简单。类 CLoad3DS 的私有函数 void ProcessNextChunk(tChunk *pPreviousChunk)是主递归函数，不断调用其他函数完成全部数据的读取。其中网格信息是这样组织的：t3DObject.pVerts 是一个 numOfVerts 维 CVector3 类型的数组，里面是三角网络的所有顶点坐标，不分先后顺序；t3DObject.pFaces 是一个 numOfFaces 维 tFace 类型的数组，记录了每个面所属的 3 个顶点的坐标索引和纹理坐标索引，由索引可直接访问坐标值。

数据读取到此介绍，然而还要进行一些预加工。

首先，实验发现，由 3DS MAX 生成的 3ds 文件存在顶点坐标冗余的问题，即 t3DObject.pVerts 中记录的顶点坐标有重复，导致某些面虽共顶点，但顶点索引却指向内存中不同位置，尽管这些位置中的数据完全相同。在网格粗糙处理时，要改变某一顶点的坐标，则要查找所有相同点，既占内存又效率低下。对此，函数 void OptimizeVertex()被用于去掉冗余顶点坐标，更新面顶点索引。

其次，每个顶点的法线方向需进行计算，以便产生光照效果和粗糙效果。一

个顶点可以属于多个面，其法线方向的计算可以有两种方案。一种是取这些面的法线的矢量平均值，但有时候会出现一些三角形很大而另一些很小的情况，简单平均值于理想的法线位置偏差较大。另一种方案则进行改进，把三角形面积作为权重，取各个面法线的加权平均值，这样便克服了上述缺点，计算也非常快，因为两个边向量叉乘既得出法向量，又同时得三角形面积，直接将叉乘结果取平均再单位化即可。

第三节 ConvertModel 程序设计

核心类 CLoad3DS 设计好了之后，剩下的就是建立程序外壳进行包装。首先考虑界面。由于要进行粗糙处理，必须有效果预览窗口，当然，能从任意角度预览模型最好。除了一些控制按钮，还要有模型信息显示窗口。在 3DS MAX 程序中建立的模型大小不定，位置不定，不能直接导入到 SEM 成像模拟程序中，还需放缩和移动位置，对应也有参数控制框。按照此思路，设计出的程序 ConvertModel 界面如图 XXX 所示。

编程语言：visual C++	函数库：MFC，openGL
主框架：MFC 对话框程序	IDE：visual studio 6.0

名词解释：

> MFC(Microsoft Foundation Classes)

微软基础类库，用于在 C++环境下编写应用程序的一个框架，对绝大多数 WindowsAPI 进行了封装，显著得提高了 Windows 程序的开发效率。

> OpenGL

是一个开放的、独立于操作系统的三维图形函数库，通过简单的调用就可以实现各种 3D 场景的高速渲染，绝大部分显卡都支持。

> ActiveX 控件

控件就是具有用户界面的独立组件，例如按钮、列表框、编辑框。可以非常容易地通过拖拽“画”在对话框模板中，从而改变对话框程序外观和功能

框架生成步骤：

首先利用 VC++6.0 的应用程序生成向导（MFC appWizard）创建一个 MFC 对话框程序框架，然后设置链接 glut32.dll、opengl32.dll、glu32.dll 以便调用 openGL 的函数库，接着在资源编辑器里设计程序界面，调整控件位置，然后

通过宏映射生成控件响应函数，最后编写功能函数，由控件响应函数来调用。这样编译好程序后，程序的执行流程为：空闲等待---点击控件---执行控件响应函数---执行功能函数----函数返回后进入空闲态，继续等待....

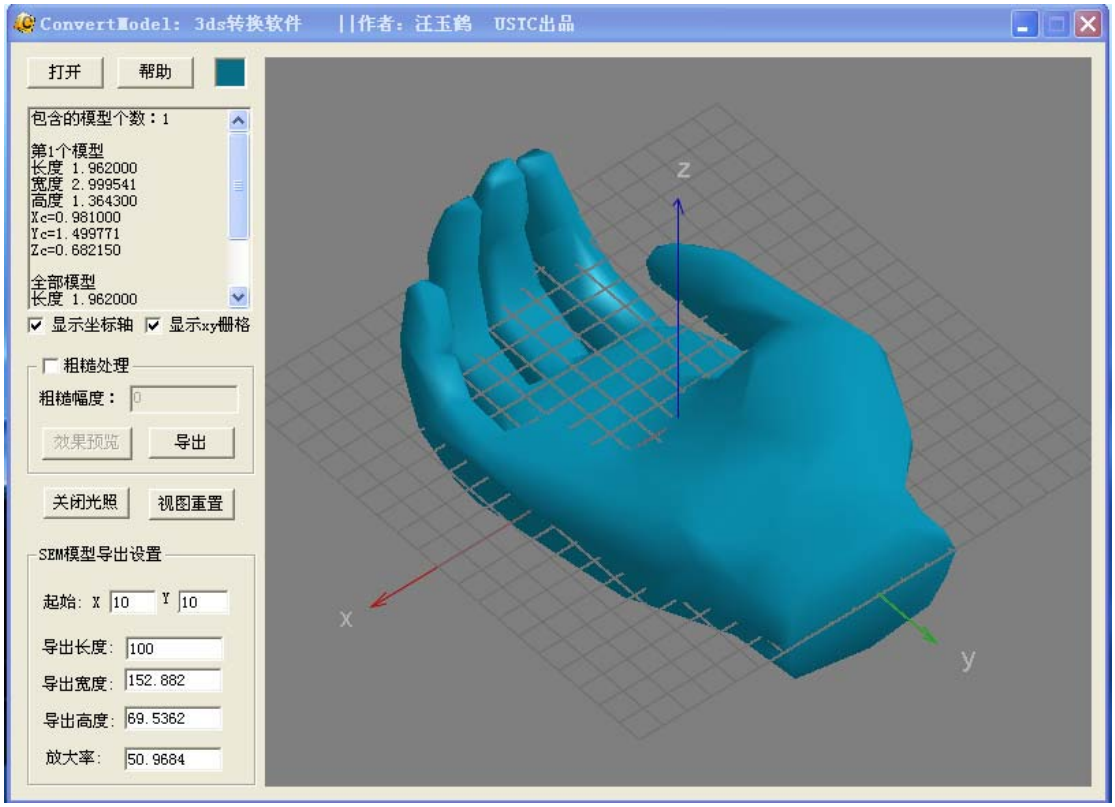

图 2-3-1 ConvertModel 程序界面

图 2-3-1 展示了手模型的预览界面。此模型是由 3DS MAX 创建的，形象逼真，这正是 3DS MAX 优势所在。如果不进行粗糙处理，那么就可以按“SEM 模型导出设置”把 CLoad3DS 中的数据输出到文件中。目标文件的格式较简单，首先是两行固定的浮点数以示间隔，然后就是一个三角形的三个顶点的坐标，每一个面 11 行浮点值，接着不断重复，直到所有面的坐标都包含进去。

第四节 表面粗糙处理

ConvertModel 程序在转换时可以对模型表面进行粗糙处理，使之更接近真实的物体。根据 YYY 的研究，当模型表面凸凹幅度满足高斯分布时，可较准确地模拟真实物体表面形貌。模型每个顶点的法线计算已在第二节说明，现在只需将顶点坐标沿法线移动即可，移动距离满足 $N(0,\sigma)$ 的高斯分布，其中 σ 是可变参量。

高斯分布的产生相对简单。首先，由 16807 随机数生成器得到[0,1]间的均匀分布，C 语言函数如下：

```
double random(bool func)   //16807 随机数产生器
{
    static unsigned int In;
     if(func==1)
    {
      In=16807*(In%127773)-2836*(In/127773);
      if(In<0) In=In+2147483647;
      return (double)In/2147483647;
    }
    else //用系统时间初始化迭代数
    {
        CTime tm=CTime::GetCurrentTime();
         short y,m,d,h,n,s;
        y=tm.GetYear();
        m=tm.GetMonth();
        d=tm.GetDay();
        h=tm.GetHour();
         n=tm.GetMinute();
        s=tm.GetSecond();
        In=y+70*(m+12*(d+31*(h+23*(n+59*s))));
    }
}
```

然后，利用 Box-Muller 法产生高斯分布，其中还利用舍选的方法替换耗时的三角函数，并剔除了超过 3σ 的样本，毕竟模型表面不会局域过于凸凹。C 函数实现如下：

```
double StandGuass() //标准高斯分布
{
    static double u,v,r,sample;
    do
    {
        do
        {
            u=Random(TRUE);
            v=Random(TRUE);
            r=sqrt(u*u+v*v);
        }while(r>1);
        if(!r) continue; //防止被 0 除
        sample=(u/r)*sqrt(-2*log(r*r));
    } while (fabs(sample)>3);//舍去 3σ 之外的值
    return sample;
```

}

第五节 任意 3D 模型 SEM 成像模拟

前面已经阐述，在此之前本实验室的程序已经可以模拟生成任意形状模型的 SEM 图像，但是建模耗时太长，无法展开更多研究。现在，ConvertModel 程序可以将 3DS MAX 构建的模型直接转换成目标文件，而利用 3DS MAX 建模则是一件十分惬意的事，只要熟悉操作，几分钟便可以建立看起来十分复杂的模型。

图 2-5-1 展示的是可编辑多边形法构建的人手模型，导入模拟程序，得到的模拟 SEM 图像与模型截图几乎完全一致。

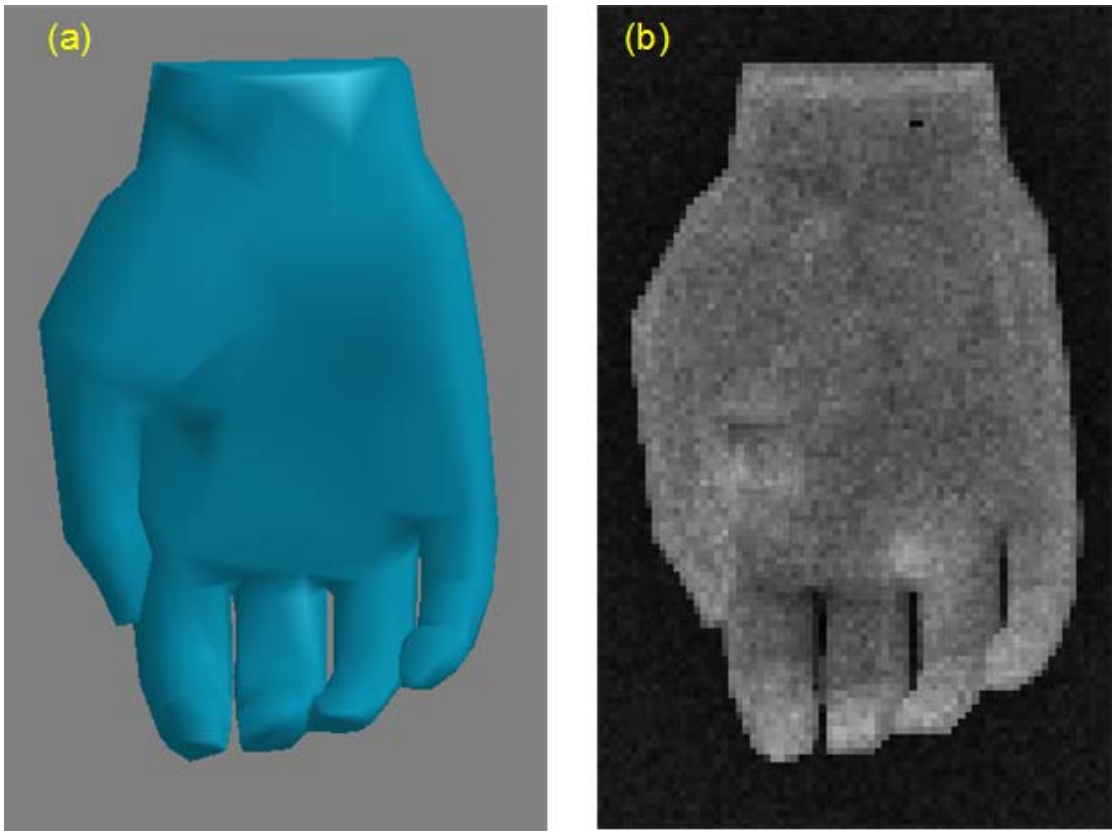

图 2-5-1 人手模型截图(a)和 SEM 模拟图片(b)

图 2-5-2 则展示了样条曲线挤出法建立的 USTC 字样的 3D 模型，用时不超过一分钟。从网格图上看，三角形的分布还是十分复杂的，手工计算显然很慢。

图 2-5-2 立体文字截图(a)和 SEM 模拟图片(b)：USTC

图 2-5-1 和图 2-5-2 中的模型一般不会出现在 SEM 镜头下，只是用来说明这种方法模拟复杂模型的强大能力。我们当然更希望模拟 SEM 镜头下常见的模型。图 2-5-3 和图 2-5-5 分别展示了芯片电路和孢子的 SEM 模拟图像，与原图具有较高的相似度。

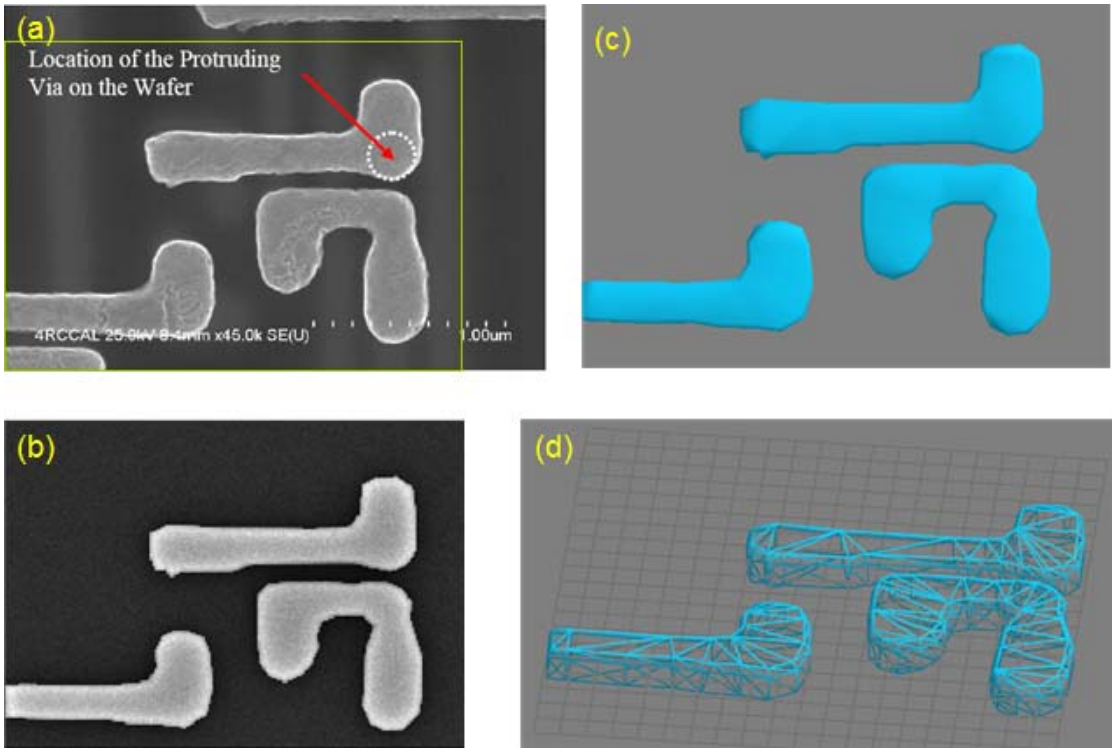

图 2-5-3 芯片电路的实物图(a)、模拟图(b)、3D 模型俯视图(c)和网格侧视图(d)

芯片电路由于比较简单，因此模拟效果更好一点。孢子模型的结构则比较复杂，构成的三角形较多，由于模拟程序还有缺陷，在某些角度会出现亮点或空洞情况，越是表面形貌复杂的模型，越容易出现这种情况。因此模拟程序仍有待改进。

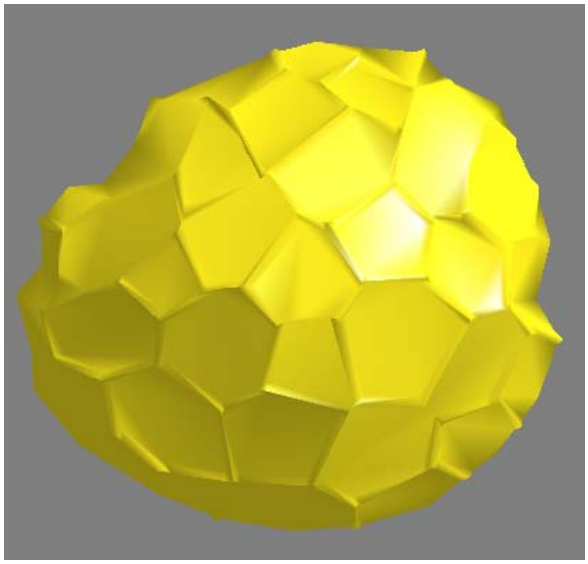

图 2-5-4 花粉模型(ConvertModel 中观察)

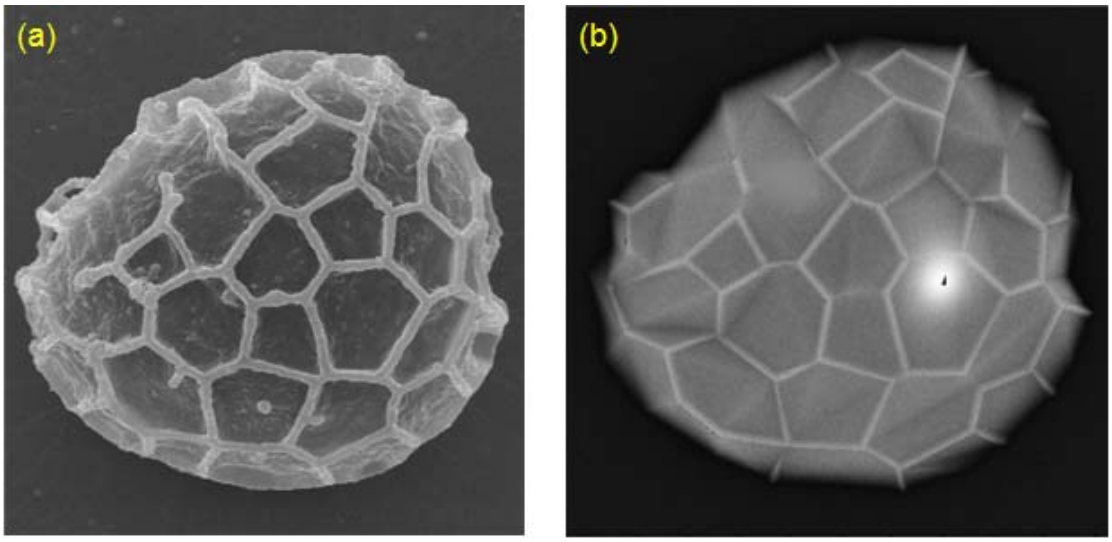

图 2-5-5 花粉在电镜下的实物图(a)和模拟图(b)对比

有了这种建模的途径，研究任何微观物质 SEM 图像便更加便捷，故此项工作具有十分重要的意义。

第三章 Si 蚀刻线宽测量

第一节 实验的原理

半导体集成电路中刻蚀线宽的精确性直接决定着电路的性能。门电路线宽已经降到 30 nm 以下，相应测量精度应低于 2.6 nm。因此，纳米尺度的精确测量已成为半导体行业未来发展的关键和极具挑战性的工作。 国际上一些研究组织和科研机构（ITRS，NIST 等）对此付出了大量的研究精力。纳米度量学中通常运用的仪器设备有 SEM,TEM 和 AFM。其中 SEM 被广泛应用于临界维度的测量，在 IC 工业中，多运用测长扫描电镜（CD-SEM）来测量线宽因为 CD-SEM 有着高分辨，高效率等优点。但是，由于它是 SE 信号成像，需要有线宽确定的精确方法。

测量的误差包含四个方面：一是 CD-SEM 生成的图像在此精度下，边界不再那么明晰，不同的判定标准会造成结果的差异。二是 SEM 扫描图像并非像光学测量那样跟样品完全相似，考虑到电子和样品间复杂的相互作用，扫描图像和实际样品间存在一定偏差。三是样品表面不是十分平滑，存在粗糙度，这样边界的判定更加困难。四是 Si 蚀刻线可能是小角度梯形，垂直扫描上下边界重影严重，影响测量精度。

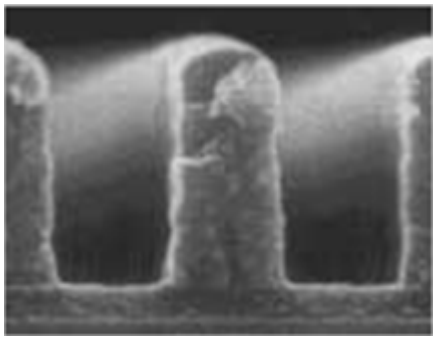

图 3-1-1 Si 蚀刻形状图

从实验上研究这四个偏差比较困难，主要是由于难以制作出精确尺寸的样品供对照研究。然而模拟研究不存在这个问题，3D 模型可以用 3DS MAX 快速制作出来，其尺寸、角度、粗糙度都是精确可控的，我们可以通过模拟 SEM 扫描过程，按照一定的判断原则处理扫描信号，将测量的线宽和模型的准确线宽进行对照，得出修正量和尺寸、角度、粗糙度之间的关系，试图建立经验公式，给精确理论的建立提供第一手资料。

那么判定边界位置的原则怎样确定呢？图 3-1-2(b)显示的是横向扫描到边界

时 CD-SEM 检测到的电子信号。由于一般边界处是较陡的梯形，所以图像会先近似沿下降，然后在到达底面时变平缓，下降直线和平缓近似线的交点位置可认为是边界点，即图(b)中的蓝色方块处。

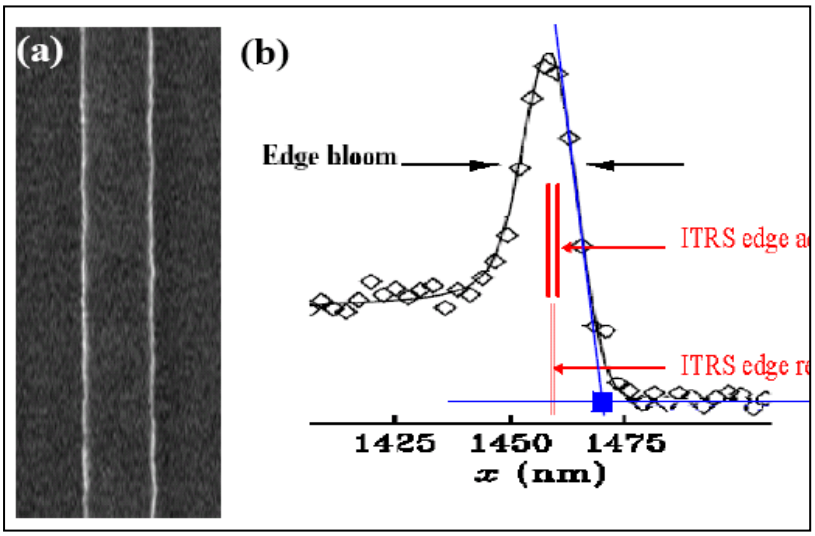

图 3-1-2 线宽扫描图像和信号图

模拟实验的过程是：按照不同集合尺寸构建模型，模型的边要合理地分段，以便模拟粗糙表面。通过 ConvertModel 添加粗糙度并转换格式，导入模拟程序进行计算。按照上述方法得到两个边界点，测得模拟长度，与模型的实际长度比较，绘制图表。

第二节 Si 蚀刻线模型的建立

第二章已经详细讨论了快速构造模型的 3DS MAX+ConvertModel 万能工具，现在就可以应用在 Si 蚀刻线宽研究上。根据图 XXX 中 Si 蚀刻线的形貌，我们可以建立图 3-2-1 所示的简化模型。

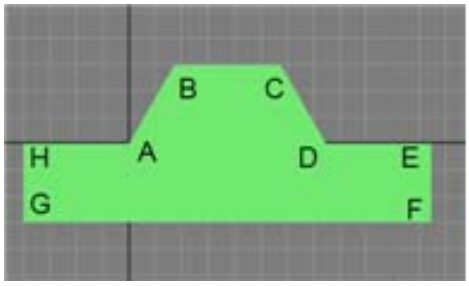

图 3-2-1 Si 蚀刻线模型

在 3DS MAX 中构建该模型最简单的方法是利用样条线挤出 3D 模型。首先，用鼠标拖动或键盘输入建立上述封闭面，然后对每一个边进行分段，然后切换到修改器，选择“挤出”效果，调整分段，就可以得到如图 3-2-2 左侧所示的网格模型。

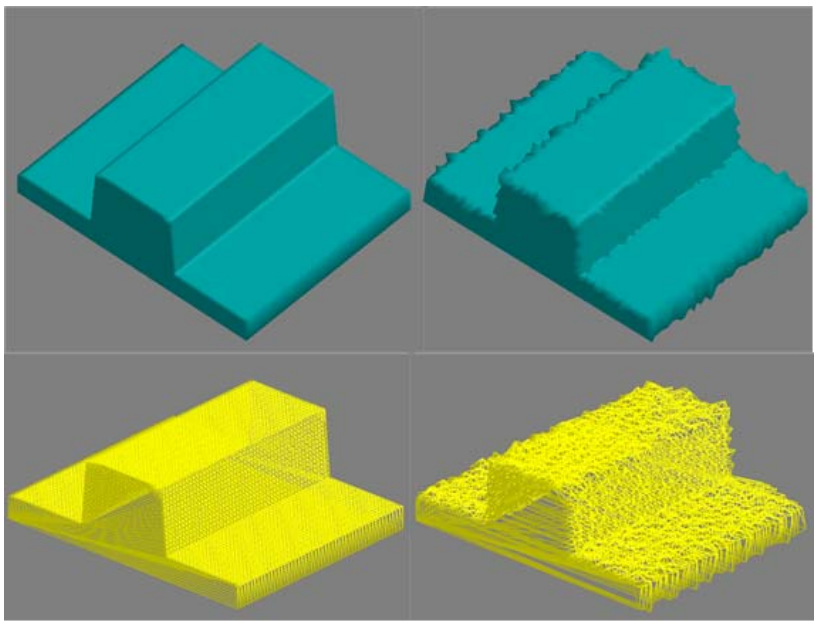

图 3-2-2 光滑（左）和粗糙（右）蚀刻模型

接着导入 ConvertModel，加入粗糙处理，效果如图 3-2-2 右侧所示。此问题中由于模型比较对称，尺寸要求十分精确，最好不要用鼠标拖动形成样条线，而是通过键盘输入。为简单起见，我特意写了一个很小的程序来计算 A-H 八个格点的坐标，界面如图 3-2-3 所示：

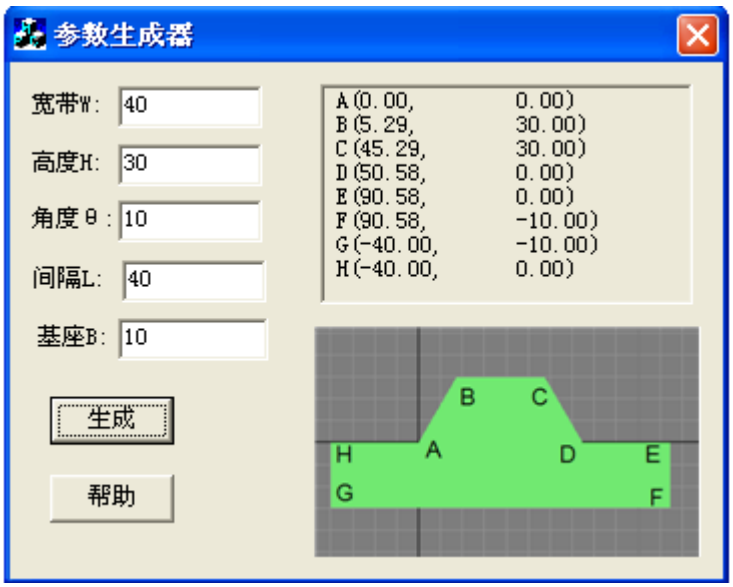

图 3-2-3 边角坐标计算小程序

需要的参数是上表面宽度 W、蚀刻线高度 H、边墙角 θ 、相邻线间距、及基座厚度。其中影响线宽测量的参数是前三个。将右上角的坐标输入到 3DS MAX 即可迅速建立起尺寸精确的蚀刻线模型。

第三节 模拟计算测长

一、与边墙角 θ 的关系

先不考虑表面粗糙度，只改变边墙角度θ的取值，分别为 5，10，15，20 度，设置墙高 H=20nm，上顶宽度 W=10nm，快速建立模型，然后输入实验室模拟程序进行线扫描，扫描图像如图 3-3-1-1 所示，可见这样的模型模拟出的扫描信号边界比较清晰，直接在 origin 中读取末端最低点坐标即可的出底边的长度。测量值和理论值对比如下：

表 3-3-1 角度和测量值

角度 θ /°	理论/nm	测量/nm
5	13.4995	13.62
10	17.0531	17.03
15	20.718	20.72
20	24.5588	24.65

模拟测量最大误差不过 0.1nm，在实际误差容许的范围内。可见边墙角对测量误差的贡献很小。可以猜想，测量的主要误差来自于构件的表面粗糙度。

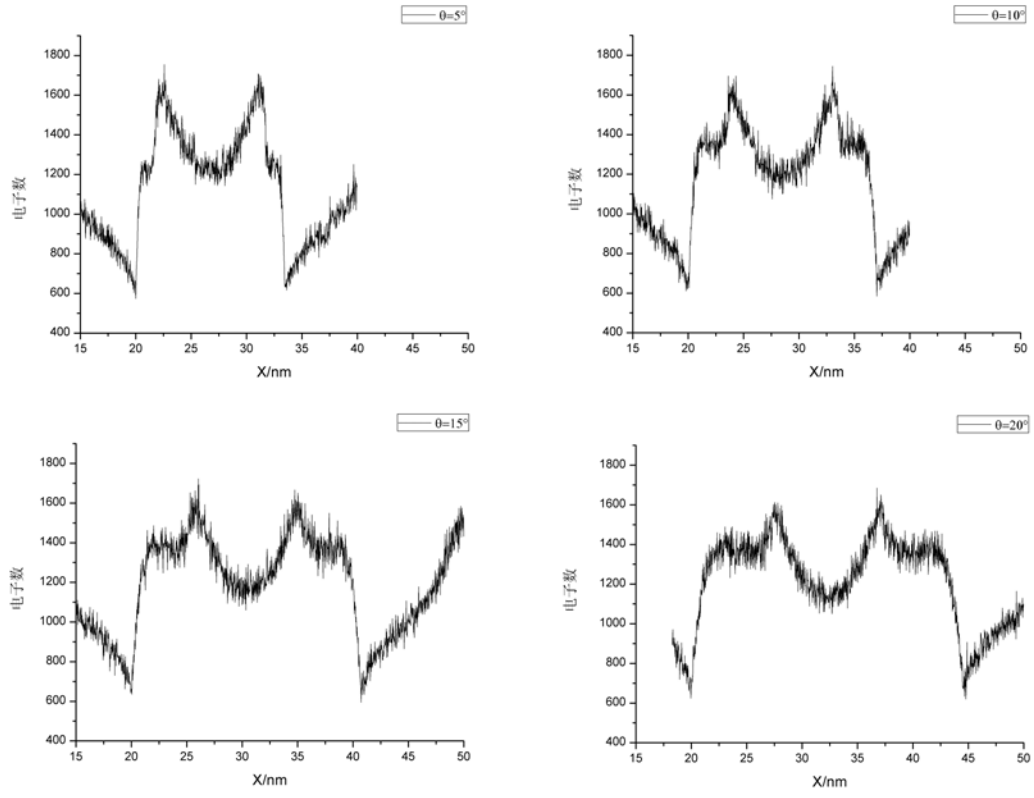

图 3-3-1-1 不同边墙角线扫描图像

二、与表面粗糙度的关系

一般情况下 Si 蚀刻线的边墙角比较小，故选定 θ=5°，与上述第一个模型相同，但是添加了表面粗糙度。取粗糙度 σ=0.2~0.5nm 四个较小值，用 ConvertModel 程序对模型进行粗糙处理，然后导入模拟程序进行线扫描，得到的图像如图 3-3-2-1 所示。由于边界有一个较小的坡面，边界扫描信号处近似直线，当信号在附近出现最大的跳跃时可判定为边界。在这些图像中，边界位置十分明显，因此可以直接利用 origin 中的坐标读取器读取边界点的坐标，得出的数据如表 3-3-2 所示。绘制出修正值和粗糙度的关系图，如图 3-3-2-2 所示。

表 3-3-2 粗糙度和边界点的关系

粗糙度/nm	左侧拐点/nm	电子数	右侧拐点/nm	电子数	修正值/nm
0.2	19.9	555	33.5	477	1.8502
0.3	19.6	491	33.85	632	2.5002
0.4	19.6	579	33.65	398	2.3002
0.5	19.65	577	33.78	388	2.3802

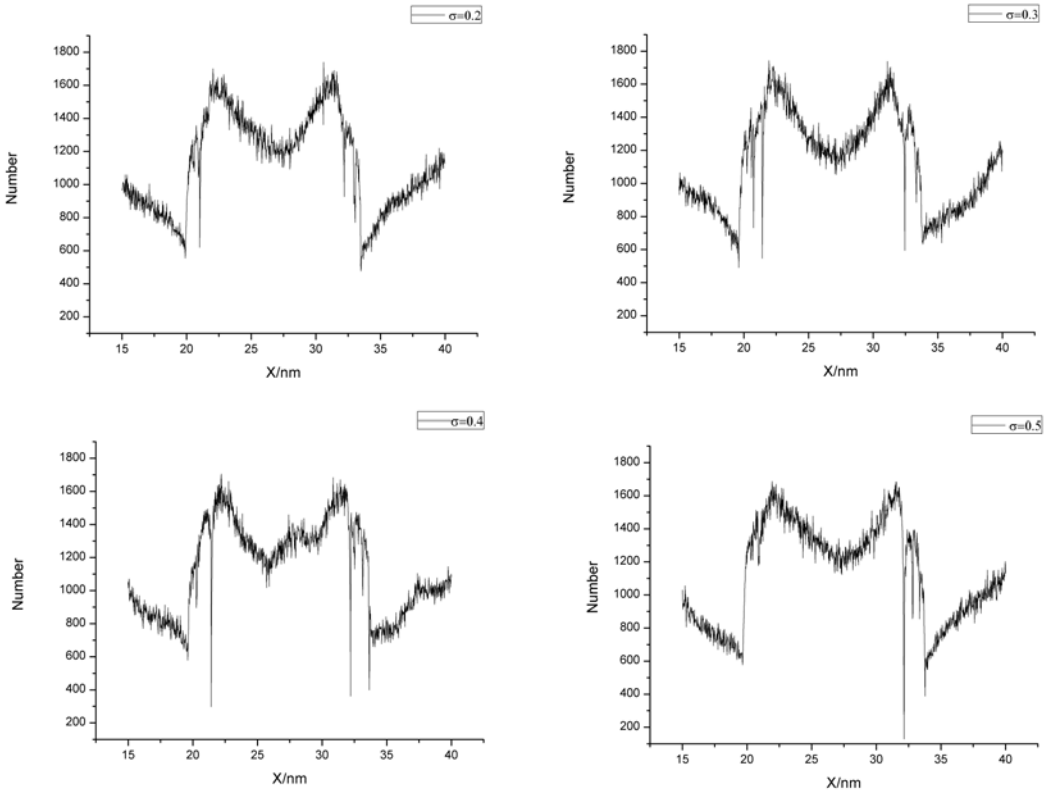

图 3-3-2-1 不同粗糙度下扫描信号图像

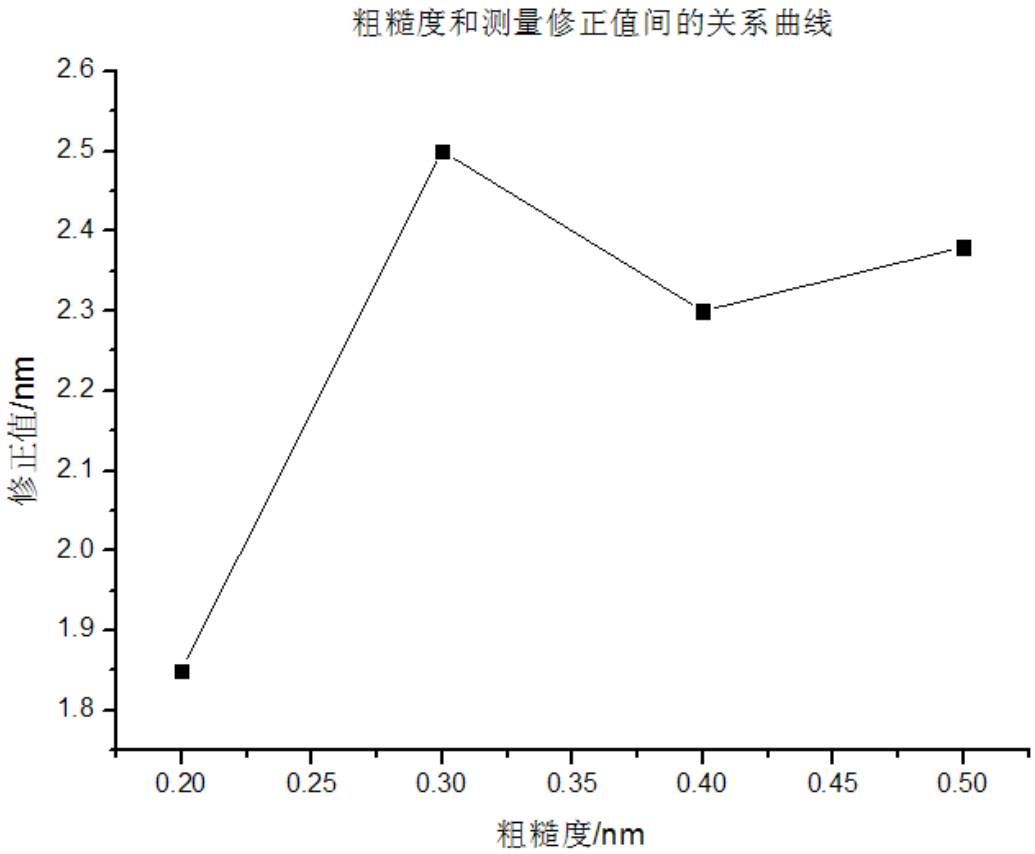

图 3-3-2-2 粗糙度和测量修正值间的关系

可见修正和粗糙度并非呈递增关系，但由于时间仓促，尚未获得更多的点来研究二者的联系。不过这些工作仅仅是简单地重复，后续工作将进一步完善。此项工作利用到了 3DS MAX 快速构建模型的能力，大大加快了理论研究的进度，是第二章内容的良好佐证。

参 考 文 献

[1] Yu. A. Novikov, Yu. V. Ozerin, A. V. Rakov and P. A. Todua, *Meas. Sci. Technol.* **18**, 367 (2007)

[2] C. G. Frase and W. Hαβler-Grohne, *Surf. Interface Anal.* **37**, 942 (2005)

[3] C. G. Frase, W. Hαβler-Grohne, G.. Dai, H. Bosse, Yu. A. Novikov and A.V. Rakov, *Meas. Sci. Technol.* **18**, 439 (2007)

[4] C. Shishido, Y. Takagi, M. Tanaka, O. Komuro, H. Morokuma and K. Sasada, *Proc. of SPIE* **4689**, 653 (2002)

[5] M. Tanaka, C. Shishido, Y. Takagi, H. Morokuma, O. Komuro and H. Mori, *Proc. of SPIE* **5038**, 624 (2003)

[6] H. Morokuma, A. Miyamoto, M. Tanaka, M. Kazui and A. Takane, *Proc. of SPIE* **5375**, 727 (2004)

[7] J. S. Villarrubia, A. E. Vladar, B. D. Bunday and M. Bishop, *Proc. of SPIE* **5375**, 199 (2004)

[8] J. S. Villarrubia, A. E. Vladar and M. T. Postek, J. Microlithogr., *Microfabr., Microsyst.* **4**, 033002 (2005)

[9] J. S. Villarrubia, A. E. Vladar and M. T. Postek, *Surf. Interface Anal.* **37**, 951 (2005)

[10] H.M. Li and Z.J. Ding, *Scanning.* **27**, 254(2005)

[11] Z.J. Ding and H.M. Li, *Surf. Interface Anal.* **37**, 912 (2005)

[12] Y.G. Li, S.F. Mao, H.M. Li, S.M. Xiao and Z.J. Ding, *J. Appl. Phys.* **104**, 064901(2008).

[13] 李会民，扫描电子显微镜成像及电子能谱的模拟研究，USTC博士学位论文，200505

[14] Jim Pitts, 3D Studio File Format, 199412

附录：C++类 CLoad3DS 源码

由于 CLoad3DS 封装了读取 3ds 文件的大部分方法，很容易移植到其他程序中，故附上其源码，以便他人借鉴。

一、CLoad3DS 的头文件：

```
#include <vector>
#include <stdio.h>
using namespace std;
/*****************3ds 文件相关宏和数据结构*****************/
//  基本块(Primary Chunk)，位于文件的开始
#define PRIMARY      0x4D4D

//  主块(Main Chunks)
#define OBJECTINFO    0x3D3D                // 网格对象的版本号
#define VERSION      0x0002            // .3ds 文件的版本
#define EDITKEYFRAME  0xB000                // 所有关键帧信息的头部

//  对象的次级定义(包括对象的材质和对象）
#define MATERIAL     0xAFFF            // 保存纹理信息
#define OBJECT       0x4000            // 保存对象的面、顶点等信息

//  材质的次级定义
#define MATNAME      0xA000            // 保存材质名称
#define MATDIFFUSE    0xA020                // 对象/材质的颜色
#define MATMAP       0xA200            // 新材质的头部
#define MATMAPFILE    0xA300                // 保存纹理的文件名

#define OBJECT_MESH   0x4100                // 新的网格对象

//  OBJECT MESH 的次级定义
#define OBJECT_VERTICES    0x4110          // 对象顶点
#define OBJECT_FACES       0x4120          // 对象的面
#define OBJECT MATERIAL    0x4130          // 对象的材质
#define OBJECT_UV          0x4140          // 对象的 UV 纹理坐标

// 保存块信息的结构
struct tChunk
{
    unsigned short int ID;                  // 块的 ID
    unsigned int length;                    // 块的长度
    unsigned int bytesRead;                 // 需要读的块数据的字节数
};

/***********************模型相关数据结构*************************/
// 定义 3D 点的类，用于保存模型中的顶点
class CVector3
{
public:
    float x, y, z;
};

// 定义 2D 点类，用于保存模型的 UV 纹理坐标
class CVector2
{
public:
    float x, y;
};

//  面的结构定义
struct tFace
{
    int vertIndex[3];           // 顶点索引
    int coordIndex[3];          // 纹理坐标索引
};
```

```
//  材质信息结构体
struct tMaterialInfo
{
    char  strName[255];         // 纹理名称
    char  strFile[255];         // 如果存在纹理映射，则表示纹理文件名称
    BYTE  color[3];             // 对象的 RGB 颜色
    int   texureId;             // 纹理 ID
    float uTile;                // u 重复
    float vTile;                // v 重复
    float uOffset;             // u 纹理偏移
    float vOffset;              // v 纹理偏移
} ;

//  对象信息结构体
struct t3DObject
{
    int  numOfVerts;            // 模型中顶点的数目
    int  numOfFaces;            // 模型中面的数目
    int  numTexVertex;          // 模型中纹理坐标的数目
    int  materialID;            // 纹理 ID
    bool bHasTexture;           // 是否具有纹理映射
    char strName[255];          // 对象的名称
    float range[3][2];          // 对象顶点坐标的最大最小值
    CVector3  *pVerts;          // 对象的顶点
    CVector3  *pNormals;        // 对象的法向量
    CVector2  *pTexVerts;       // 纹理 UV 坐标
    float *pRoughness;          // 粗糙度
    tFace *pFaces;              // 对象的面信息
};

/***************3ds 模型类，负责从文件载入模型数据***************/

// CLoad3DS 类处理所有的装入代码
class CLoad3DS
{
public:
    CLoad3DS();                          // 空的构造函数
    CLoad3DS(CString strFileName);          // 构造时已载入模型
    ~CLoad3DS();
    bool Import3DS(CString strFileName);   // 装入 3ds 文件到模型结构中
    void CleanUp();                      // 清除模型，以便下次载入

    vector<tMaterialInfo> vMaterials;  // 材质信息向量容器
    vector<t3DObject> vObject;         // 模型信息向量容器

private:
    void OptimizeVertex();
    // 读一个字符串
    int GetString(char *);
    // 读下一个块
    void ReadChunk(tChunk *);
    // 读下一个块
    void ProcessNextChunk(tChunk *);
    // 读下一个对象块
    void ProcessNextObjectChunk(t3DObject *pObject, tChunk *);
    // 读下一个材质块
    void ProcessNextMaterialChunk(tChunk *);
    // 读对象颜色的 RGB 值
    void ReadColorChunk(tMaterialInfo *pMaterial, tChunk *pChunk);
    // 读对象的顶点
    void ReadVertices(t3DObject *pObject, tChunk *);
    // 读对象的面信息
    void ReadVertexIndices(t3DObject *pObject, tChunk *);
    // 读对象的纹理坐标
    void ReadUVCoordinates(t3DObject *pObject, tChunk *);
    // 读赋予对象的材质名称
    void ReadObjectMaterial(t3DObject *pObject, tChunk *pPreviousChunk);
    // 计算对象顶点的法向量
```

```
    void ComputeNormals();

    // 文件指针
    FILE *m_FilePointer;
    tChunk *m_CurrentChunk;
    tChunk *m TempChunk;
};
```

二、CLoad3DS 的源文件：

```
#include "stdafx.h"
#include "3dsModel.h"
#include "math.h"

//  构造函数的功能是初始化 tChunk 数据
CLoad3DS::CLoad3DS()
{
}

CLoad3DS::CLoad3DS(CString strFileName)
{
    Import3DS(strFileName);
}

CLoad3DS::~CLoad3DS()
{
    CleanUp();
}

//  打开一个 3ds 文件，读出其中的内容，并释放内存
bool CLoad3DS::Import3DS(CString strFileName)
{
    if(vObject.size())
    {
        MessageBox(NULL, "模型文件已经载入，请不要重复载入", "Error", MB OK);
        return false;
    }
    // 打开一个 3ds 文件
    m_FilePointer = fopen(strFileName, "rb");
    // 确保所获得的文件指针合法
    if(!m FilePointer)
    {
        MessageBox(NULL, "Unable to find the file:"+strFileName, "Error", MB_OK);
        return false;
    }

    //申请当前块和临时块
    m_CurrentChunk = new tChunk;
    m_TempChunk = new tChunk;
    // 将文件的第一块读出并判断是否是 3ds 文件
    ReadChunk(m_CurrentChunk);
    // 确保是 3ds 文件
    if (m CurrentChunk->ID != PRIMARY)
    {
        MessageBox(NULL, "Unable to load PRIMARY chuck from file:"+strFileName,
"Error", MB OK);
        return false;
    }

    // 现在开始读入数据，ProcessNextChunk()是一个递归函数
    ProcessNextChunk(m_CurrentChunk);
    //清除重复点
    OptimizeVertex();
    // 在读完整个 3ds 文件之后，计算顶点的法线
    ComputeNormals();

    //为每个顶点分配粗糙度存储空间
    for (int i=0;i<vObject.size();i++)
```

```
    {
        vObject[i].pRoughness=new float[vObject[i].numOfVerts];
        memset(vObject[i].pRoughness,0,vObject[i].numOfVerts*sizeof(float));
    }

    fclose(m FilePointer);                          // 关闭当前的文件指针
    delete m_CurrentChunk;                          // 释放当前块
    delete m_TempChunk;                             // 释放临时块

    return true;
}


//  下面的函数释放清除模型对象，便于下次载入
void CLoad3DS::CleanUp()
{
    if(vObject.size())
    {
        // 删除每个对象申请的内存
        for(int i = 0; i < vObject.size(); i++)
        {
            // 删除所有的变量
            delete [] vObject[i].pFaces;
            delete [] vObject[i].pNormals;
            delete [] vObject[i].pVerts;
            delete [] vObject[i].pTexVerts;
            delete [] vObject[i].pRoughness;
        }
        //清空容器
        vObject.clear();
        vMaterials.clear();
    }
}



//  下面的函数读出 3ds 文件的主要部分
void CLoad3DS::ProcessNextChunk(tChunk *pPreviousChunk)
{
    t3DObject newObject = {0};                  // 用来添加到对象链表
    tMaterialInfo newTexture = {0};             // 用来添加到材质链表
    unsigned int version = 0;                   // 保存文件版本
    int buffer[50000] = {0};                    // 用来跳过不需要的数据
    m_CurrentChunk = new tChunk;                // 为新的块分配空间

    //  下面每读一个新块，都要判断一下块的 ID，如果该块是需要的读入的，则继续进行
    //  如果是不需要读入的块，则略过

    // 继续读入子块，直到达到预定的长度
    while (pPreviousChunk->bytesRead < pPreviousChunk->length)
    {
        // 读入下一个块
        ReadChunk(m_CurrentChunk);
        // 判断块的 ID 号
        switch (m_CurrentChunk->ID)
        {
        case VERSION:                           // 文件版本号
            // 读入文件的版本号，并将字节数添加到 bytesRead 变量中
            m_CurrentChunk->bytesRead       +=       fread(&version,       1,
m CurrentChunk->length - m CurrentChunk->bytesRead, m FilePointer);
            // 如果文件版本号大于 3，给出一个警告信息
            if (version > 0x03)
                MessageBox(NULL, "This 3DS file is over version 3 so it may load
incorrectly", "Warning", MB_OK);
            break;

        case OBJECTINFO:                        // 网格版本信息
            // 读入下一个块
            ReadChunk(m TempChunk);
```

```
            // 获得网格的版本号
            m_TempChunk->bytesRead += fread(&version, 1, m_TempChunk->length -
m TempChunk->bytesRead, m FilePointer);
            // 增加读入的字节数
            m_CurrentChunk->bytesRead += m_TempChunk->bytesRead;
            // 进入下一个块
            ProcessNextChunk(m_CurrentChunk);
            break;

        case MATERIAL:                          // 材质信息
            // 在纹理链表中添加一个空白纹理结构
            vMaterials.push back(newTexture);
            // 进入材质装入函数
            ProcessNextMaterialChunk(m_CurrentChunk);
            break;

        case OBJECT:                            // 对象的名称
            // 该块是对象信息块的头部，保存了对象了名称
            // 添加一个新的 tObject 节点到对象链表中
            vObject.push_back(newObject);
            // 初始化对象和它的所有数据成员
            memset(&vObject.back(), 0, sizeof(t3DObject));
            // 获得并保存对象的名称，然后增加读入的字节数
            m CurrentChunk->bytesRead  +=  GetString(vObject.back().strName);

            // 进入其余的对象信息的读入
            ProcessNextObjectChunk(&vObject.back(), m CurrentChunk);
            break;

        case EDITKEYFRAME:
            // 跳过关键帧块的读入，增加需要读入的字节数
            m_CurrentChunk->bytesRead        +=        fread(buffer,        1,
m CurrentChunk->length - m CurrentChunk->bytesRead, m FilePointer);
            break;

        default:
            //  跳过所有忽略的块的内容的读入，增加需要读入的字节数
            m_CurrentChunk->bytesRead        +=        fread(buffer,        1,
m CurrentChunk->length - m CurrentChunk->bytesRead, m FilePointer);
            break;
        }

        // 增加从最后块读入的字节数
        pPreviousChunk->bytesRead += m_CurrentChunk->bytesRead;
    }

    // 释放当前块的内存空间
    delete m CurrentChunk;
    m_CurrentChunk = pPreviousChunk;
}


//  下面的函数处理所有的文件中对象的信息
void     CLoad3DS::ProcessNextObjectChunk(t3DObject     *pObject,     tChunk
*pPreviousChunk)
{
    int buffer[50000] = {0};                    // 用于读入不需要的数据
    // 对新的块分配存储空间
    m_CurrentChunk = new tChunk;

    // 继续读入块的内容直至本子块结束
    while (pPreviousChunk->bytesRead < pPreviousChunk->length)
    {
        // 读入下一个块
        ReadChunk(m_CurrentChunk);
        // 区别读入是哪种块
        switch (m_CurrentChunk->ID)
        {
        case OBJECT MESH:                   // 正读入的是一个新块
```

```
            // 使用递归函数调用，处理该新块
            ProcessNextObjectChunk(pObject, m_CurrentChunk);
            break;

        case OBJECT_VERTICES:              // 读入是对象顶点
            ReadVertices(pObject, m CurrentChunk);
            break;

        case OBJECT FACES:                 // 读入的是对象的面
            ReadVertexIndices(pObject, m_CurrentChunk);
            break;

        case OBJECT_MATERIAL:              // 读入的是对象的材质名称

            // 该块保存了对象材质的名称，可能是一个颜色，也可能是一个纹理映射。同时在该块
中也保存了
            // 纹理对象所赋予的面
            // 下面读入对象的材质名称
            ReadObjectMaterial(pObject, m_CurrentChunk);
            break;

        case OBJECT_UV:                    // 读入对象的 UV 纹理坐标
            // 读入对象的 UV 纹理坐标
            ReadUVCoordinates(pObject, m CurrentChunk);
            break;

        default:
            // 略过不需要读入的块
            m_CurrentChunk->bytesRead        +=        fread(buffer,        1,
m CurrentChunk->length - m CurrentChunk->bytesRead, m FilePointer);
            break;
        }
        // 添加从最后块中读入的字节数到前面的读入的字节中
        pPreviousChunk->bytesRead += m_CurrentChunk->bytesRead;
    }

    // 释放当前块的内存空间，并把当前块设置为前面块
    delete m_CurrentChunk;
    m CurrentChunk = pPreviousChunk;
}

//  下面的函数处理所有的材质信息
void CLoad3DS::ProcessNextMaterialChunk(tChunk *pPreviousChunk)
{
    int buffer[50000] = {0};                    // 用于读入不需要的数据
    // 给当前块分配存储空间
    m_CurrentChunk = new tChunk;
    // 继续读入这些块，知道该子块结束
    while (pPreviousChunk->bytesRead < pPreviousChunk->length)
    {
        // 读入下一块
        ReadChunk(m_CurrentChunk);
        // 判断读入的是什么块
        switch (m CurrentChunk->ID)
        {
        case MATNAME:                           // 材质的名称
            // 读入材质的名称
            m_CurrentChunk->bytesRead  +=  fread(vMaterials.back().strName,  1,
m_CurrentChunk->length - m_CurrentChunk->bytesRead, m_FilePointer);
            break;

        case MATDIFFUSE:                        // 对象的 R G B 颜色
            ReadColorChunk(&vMaterials.back(), m CurrentChunk);
            break;

        case MATMAP:                            // 纹理信息的头部
            // 进入下一个材质块信息
            ProcessNextMaterialChunk(m_CurrentChunk);
            break;
```

```
        case MATMAPFILE:                          // 材质文件的名称
            // 读入材质的文件名称
            m_CurrentChunk->bytesRead += fread(vMaterials.back().strFile, 1,
m_CurrentChunk->length - m_CurrentChunk->bytesRead, m_FilePointer);
            break;

        default:
            // 掠过不需要读入的块
            m_CurrentChunk->bytesRead       +=       fread(buffer,       1,
m_CurrentChunk->length - m_CurrentChunk->bytesRead, m_FilePointer);
            break;
        }
        // 添加从最后块中读入的字节数
        pPreviousChunk->bytesRead += m CurrentChunk->bytesRead;
    }

    // 删除当前块，并将当前块设置为前面的块
    delete m_CurrentChunk;
    m_CurrentChunk = pPreviousChunk;
}

//  下面函数读入块的 ID 号和它的字节长度
void CLoad3DS::ReadChunk(tChunk *pChunk)
{
    // 读入块的 ID 号，占用了 2 个字节。块的 ID 号象 OBJECT 或 MATERIAL 一样，说明了在块中所
包含的内容
    pChunk->bytesRead = fread(&pChunk->ID, 1, 2, m_FilePointer);
    // 然后读入块占用的长度，包含了四个字节
    pChunk->bytesRead += fread(&pChunk->length, 1, 4, m FilePointer);
}

//  下面的函数读入一个字符串
int CLoad3DS::GetString(char *pBuffer)
{
    int index = 0;
    // 读入一个字节的数据
    fread(pBuffer, 1, 1, m_FilePointer);
    // 直到结束
    while (*(pBuffer + index++) != 0) fread(pBuffer + index, 1, 1, m_FilePointer);
// 读入一个字符直到 NULL
    // 返回字符串的长度
    return strlen(pBuffer) + 1;
}


//  下面的函数读入 RGB 颜色
void CLoad3DS::ReadColorChunk(tMaterialInfo *pMaterial, tChunk *pChunk)
{
    // 读入颜色块信息
    ReadChunk(m TempChunk);
    // 读入 RGB 颜色
    m_TempChunk->bytesRead += fread(pMaterial->color, 1, m_TempChunk->length -
m TempChunk->bytesRead, m FilePointer);
    // 增加读入的字节数
    pChunk->bytesRead += m_TempChunk->bytesRead;
}

//  下面的函数读入顶点索引
void CLoad3DS::ReadVertexIndices(t3DObject *pObject, tChunk *pPreviousChunk)
{
    unsigned short index = 0;                   // 用于读入当前面的索引
    // 读入该对象中面的数目
    pPreviousChunk->bytesRead    +=    fread(&pObject->numOfFaces,    1,    2,
m_FilePointer);
    // 分配所有面的存储空间，并初始化结构
    pObject->pFaces = new tFace [pObject->numOfFaces];
    memset(pObject->pFaces, 0, sizeof(tFace) * pObject->numOfFaces);
    // 依次读入所有的面索引
```

```
    for(int i = 0; i < pObject->numOfFaces; i++)
    {
        for(int j = 0; j < 4; j++)
        {
            // 读入当前面的第一个点
            pPreviousChunk->bytesRead  +=  fread(&index,  1,  sizeof(index),
m_FilePointer);
            if(j < 3)
            {
                // 将索引保存在面的结构中
                pObject->pFaces[i].vertIndex[j] = index;
            }
        }
    }
}

//  下面的函数读入对象的 UV 坐标
void CLoad3DS::ReadUVCoordinates(t3DObject *pObject, tChunk *pPreviousChunk)
{
    // 为了读入对象的 UV 坐标，首先需要读入 UV 坐标的数量，然后才读入具体的数据
    // 读入 UV 坐标的数量
    pPreviousChunk->bytesRead   +=   fread(&pObject->numTexVertex,   1,   2,
m_FilePointer);
    // 分配保存 UV 坐标的内存空间
    pObject->pTexVerts = new CVector2 [pObject->numTexVertex];
    // 读入纹理坐标
    pPreviousChunk->bytesRead      +=      fread(pObject->pTexVerts,      1,
pPreviousChunk->length - pPreviousChunk->bytesRead, m_FilePointer);
}

//  读入对象的顶点
void CLoad3DS::ReadVertices(t3DObject *pObject, tChunk *pPreviousChunk)
{
    // 读入顶点的数目
    pPreviousChunk->bytesRead   +=   fread(&(pObject->numOfVerts),   1,   2,
m FilePointer);
    // 分配顶点的存储空间，然后初始化结构体
    pObject->pVerts = new CVector3 [pObject->numOfVerts];
    memset(pObject->pVerts, 0, sizeof(CVector3) * pObject->numOfVerts);
    // 读入顶点序列
    pPreviousChunk->bytesRead       +=       fread(pObject->pVerts,       1,
pPreviousChunk->length - pPreviousChunk->bytesRead, m FilePointer);
    // 现在已经读入了所有的顶点。
    // 因为 3D Studio Max 的模型的 Z 轴是指向上的，因此需要将 y 轴和 z 轴翻转过来。
    // 具体的做法是将 Y 轴和 Z 轴交换，然后将 Z 轴反向。
    pObject->range[0][0]=pObject->range[0][1]=pObject->pVerts[0].x;
    pObject->range[1][0]=pObject->range[1][1]=pObject->pVerts[0].y;
    pObject->range[2][0]=pObject->range[2][1]=pObject->pVerts[0].z;
    // 遍历所有的顶点,求出模型范围
    for(int i = 1; i < pObject->numOfVerts; i++)
    {
        if(pObject->pVerts[i].x            <            pObject->range[0][0])
pObject->range[0][0]=pObject->pVerts[i].x;
        if(pObject->pVerts[i].x            >            pObject->range[0][1])
pObject->range[0][1]=pObject->pVerts[i].x;
        if(pObject->pVerts[i].y            <            pObject->range[1][0])
pObject->range[1][0]=pObject->pVerts[i].y;
        if(pObject->pVerts[i].y            >            pObject->range[1][1])
pObject->range[1][1]=pObject->pVerts[i].y;
        if(pObject->pVerts[i].z            <            pObject->range[2][0])
pObject->range[2][0]=pObject->pVerts[i].z;
        if(pObject->pVerts[i].z            >            pObject->range[2][1])
pObject->range[2][1]=pObject->pVerts[i].z;
    }
}


//  下面的函数读入对象的材质名称
void CLoad3DS::ReadObjectMaterial(t3DObject *pObject, tChunk *pPreviousChunk)
```

```
{
    char strMaterial[255] = {0};            // 用来保存对象的材质名称
    int buffer[50000] = {0};                // 用来读入不需要的数据
    // 材质或者是颜色，或者是对象的纹理，也可能保存了象明亮度、发光度等信息。
    // 下面读入赋予当前对象的材质名称
    pPreviousChunk->bytesRead += GetString(strMaterial);
    // 遍历所有的纹理
    for(int i = 0; i < vMaterials.size(); i++)
    {
        //如果读入的纹理与当前的纹理名称匹配
        if(strcmp(strMaterial, vMaterials[i].strName) == 0)
        {
            // 设置材质 ID
            pObject->materialID = i;
            // 判断是否是纹理映射，如果 strFile 是一个长度大于 1 的字符串，则是纹理
            if(strlen(vMaterials[i].strFile) > 0) {
                // 设置对象的纹理映射标志
                pObject->bHasTexture = true;
            }
            break;
        }
        else
        {
            // 如果该对象没有材质，则设置 ID 为-1
            pObject->materialID = -1;
        }
    }
    pPreviousChunk->bytesRead += fread(buffer, 1, pPreviousChunk->length -
pPreviousChunk->bytesRead, m_FilePointer);
}


//  下面的这些函数主要用来计算顶点的法向量，顶点的法向量主要用来计算光照
// 下面的宏定义计算一个矢量的长度
#define   Mag(Normal)   (sqrt(Normal.x*Normal.x   +   Normal.y*Normal.y   +
Normal.z*Normal.z))

// 下面的函数求两点决定的矢量
CVector3 Vector(CVector3 vPoint1, CVector3 vPoint2)
{
    CVector3 vVector;
    vVector.x = vPoint1.x - vPoint2.x;
    vVector.y = vPoint1.y - vPoint2.y;
    vVector.z = vPoint1.z - vPoint2.z;
    return vVector;
}

// 下面的函数两个矢量相加
CVector3 AddVector(CVector3 vVector1, CVector3 vVector2)
{
    CVector3 vResult;
    vResult.x = vVector2.x + vVector1.x;
    vResult.y = vVector2.y + vVector1.y;
    vResult.z = vVector2.z + vVector1.z;
    return vResult;
}

// 下面的函数返回两个矢量的叉积
CVector3 Cross(CVector3 vVector1, CVector3 vVector2)
{
    CVector3 vCross;

    vCross.x = (vVector1.y * vVector2.z) - (vVector1.z * vVector2.y);

    vCross.y = (vVector1.z * vVector2.x) - (vVector1.x * vVector2.z);

    vCross.z = (vVector1.x * vVector2.y) - (vVector1.y * vVector2.x);
    return vCross;
}
```

```
// 下面的函数规范化矢量
CVector3 Normalize(CVector3 vNormal)
{
    double Magnitude= Mag(vNormal);                // 获得矢量的长度
    vNormal.x /= (float)Magnitude;
    vNormal.y /= (float)Magnitude;
    vNormal.z /= (float)Magnitude;
    return vNormal;
}

//  下面的函数用于计算对象的法向量
void CLoad3DS::ComputeNormals()
{
    CVector3 vVector1, vVector2, vPoly[3];
    // 遍历模型中所有的对象
    for(int index = 0; index < vObject.size(); index++)
    {
        // 获得当前的对象
        t3DObject *pObject = &vObject[index];
        // 分配需要的存储空间
        CVector3 *pTempNormals  = new CVector3 [pObject->numOfFaces];
        pObject->pNormals       = new CVector3 [pObject->numOfVerts];
        // 遍历对象的所有面,求出每个面的法向量
        for(int i=0; i < pObject->numOfFaces; i++)
        {
            //找到该面的三个顶点
            vPoly[0] = pObject->pVerts[pObject->pFaces[i].vertIndex[0]];
            vPoly[1] = pObject->pVerts[pObject->pFaces[i].vertIndex[1]];
            vPoly[2] = pObject->pVerts[pObject->pFaces[i].vertIndex[2]];
            // 计算该面的法向量
            vVector1 = Vector(vPoly[0], vPoly[2]);     // 获得多边形的矢量
            vVector2 = Vector(vPoly[1], vPoly[2]);     // 获得多边形的第二个矢量
            pTempNormals[i] = Cross(vVector1, vVector2);// 获得两个矢量的叉积

        }
        //  下面求顶点法向量
        CVector3 vSum = {0.0, 0.0, 0.0};
        CVector3 vZero = vSum;
        // 遍历所有的顶点
        for (i = 0; i < pObject->numOfVerts; i++)
        {
            for (int j = 0; j < pObject->numOfFaces; j++)  // 遍历所有的三角形面
            {                                              // 判断该点是否与其它
的面共享
                if (pObject->pFaces[j].vertIndex[0] == i ||
                    pObject->pFaces[j].vertIndex[1] == i ||
                    pObject->pFaces[j].vertIndex[2] == i)
                vSum = AddVector(vSum, pTempNormals[j]); //将共顶点的所有面的%非
规范向量%求和
            }
            // 规范化最后的顶点法向
            pObject->pNormals[i] = Normalize(vSum);
            vSum = vZero;   //清空 vSum，便于计算下一个点

        }
        // 释放存储空间，开始下一个对象
        delete [] pTempNormals;
    }
}

void CLoad3DS::OptimizeVertex()
{

    vector<CVector3> vert;
    int obj,v,f,index,cur;
    for (obj=0;obj<vObject.size();obj++)
    {
        //压入第一个面的三个点
```

```
        for (v=0;v<3;v++)
        {
            //index=vObject[obj].pFaces[0].vertIndex[v];
            index=vObject[obj].pFaces[0].vertIndex[v];
            vert.push_back(vObject[obj].pVerts[index]);
        }
        //依次测试各个面的所有点
        for (f=0;f<vObject[obj].numOfFaces;f++)
        {
            for (v=0;v<3;v++)
            {
                index=vObject[obj].pFaces[f].vertIndex[v];
                for (cur=0;cur<vert.size();cur++)//遍历当前 vector<CVector3>，看
有无重复元素
                {
                    if (vObject[obj].pVerts[index].x==vert[cur].x&& \
                        vObject[obj].pVerts[index].y==vert[cur].y&& \
                        vObject[obj].pVerts[index].z==vert[cur].z)
                    break;
                }
                if(cur==vert.size())
vert.push_back(vObject[obj].pVerts[index]);
                vObject[obj].pFaces[f].vertIndex[v]=cur;
            }
        }
        //内存整理
        vObject[obj].numOfVerts=vert.size();
        delete  vObject[obj].pVerts;
        vObject[obj].pVerts=new CVector3[vert.size()];
        for (v=0;v<vert.size();v++)
        {
            vObject[obj].pVerts[v].x=vert[v].x;
            vObject[obj].pVerts[v].y=vert[v].y;
            vObject[obj].pVerts[v].z=vert[v].z;
        }
        //释放 vert，便于计算下一个 Object
        vert.clear();
    }
}
```