浙江工业大学

# 计算机图形学期末大作业

**2020/2021(1)**

实验题目　三角网格模型数据绘制及轮廓提取

学生姓名　项建航

学生学号　201806010108

学生班级　软工 1801

任课教师　简琤峰

提交日期　2020-12-8

计算机科学与技术学院

目录

# 目录

# 一、实验要求

- ✓ 读取 data.xml 数据，绘制三角网格模型；
- ✓ 对三角网格模型数据的边界进行提取（边界点、边界边），并绘制出边界边；
- ✓ 根据获取的边界点，采用 B 样条曲线拟合出边界的轮廓；
- ✓ 源代码实现对三角面片的绘制；（20 分）
- ✓ 源代码实现对轮廓的边的绘制；（10 分）
- ✓ 边界点或边界边提取算法优化；（20 分）
- ✓ 在上述基础上，源代码实现 B 样条曲线拟合绘制；（20 分）
- ✓ 提交实验报告能够说明实现方法并且提供系统运行实例截图；（10 分）
- ✓ 对关键代码进行解释说明；（10 分）
- ✓ 在此基础上进行算法效率分析；（10 分）

# 二、实验环境

## 2.1 操作系统环境

**操作系统 : windows10 版本 x64**

**处理器：intel(R) Core(TM) i5-8250U CPU @1.60GHz**

## 2.2 C++环境以及 OpenGL 环境

**IDE 环境: Visual Studio 2019**

**OpenGL**：glutddls37beta

## 2.3 TinyXML 包

版本：TinyXML\ TinyXML2

用途：基于 C++环境下，读取并修改 XML 文件；

核心文件：tinyxmlerror.cpp、tinyxmlparser.cpp、tinyxml.cpp、tinystr.cpp 、tinyxml2.cpp；

# 三、实验数据格式

数据文件：data.xml、2.xml、3.xml 文件

- 顶点列表<verts> 所有的三角面片的顶点记录在 <verts>中，其中<v p="0 -15 75"/>表示点的三维坐标值；
- 面片列表 <facets color="959595">

  所有的三角面片的面记录在<f>中。

  <f v="43 28 1" fn="0 4.93054498004026e-014 1">

  <n d="0 0 1"/>

  <n d="0 0 1"/>

  <n d="0 0 1"/>

  </f>

- 表示：v="43 28 1" 代表组成该三角面片的三个顶点，其值为 <verts>中对应<v>的序列号；fn="0 4.93054498004026e-014 1"代表该面片的法向量；<n d="0 0 1"/>代表顶点的法向量。

# 四、实验内容及主要步骤

## 4.1 数据的读取

**Step1：**观察数据结构以及格式，数据文件 data.xml 文件为 XML 类型文件，需要调用 TinyXML 包进行 XML 文件的读取与修改，数据内容如图 4-1-1 所示.

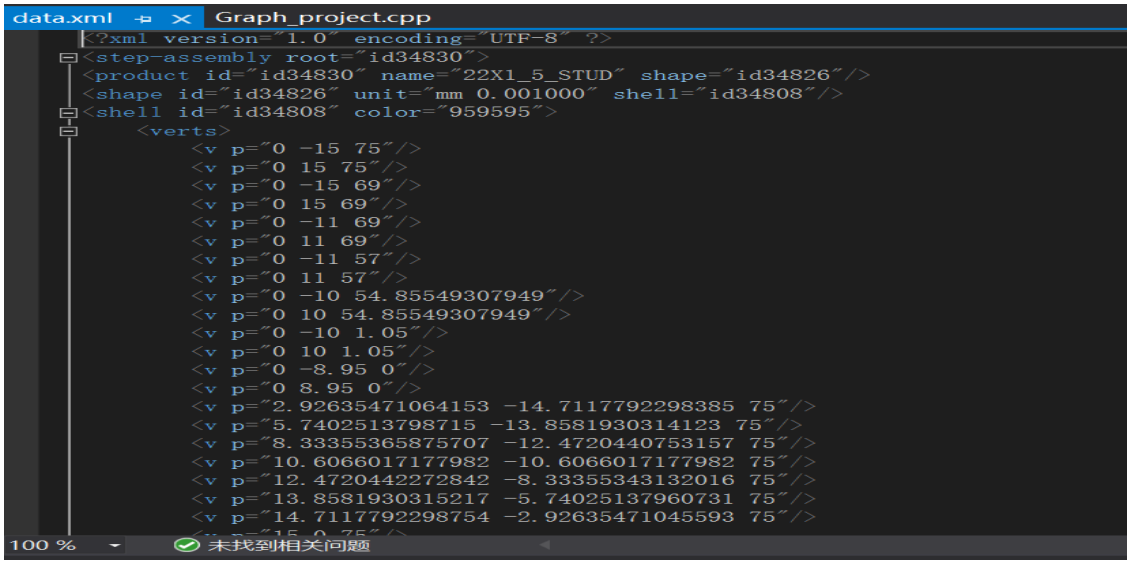



图 4-1-1 data.xml 文件的数据内容

**Step2：**深入了解数据文件内容后，发现顶点数据都储存于<verts>之内共计 238 个顶点。

三角面片数据都存于<facets>之内，共计 13 个曲面，472 个三角面片。

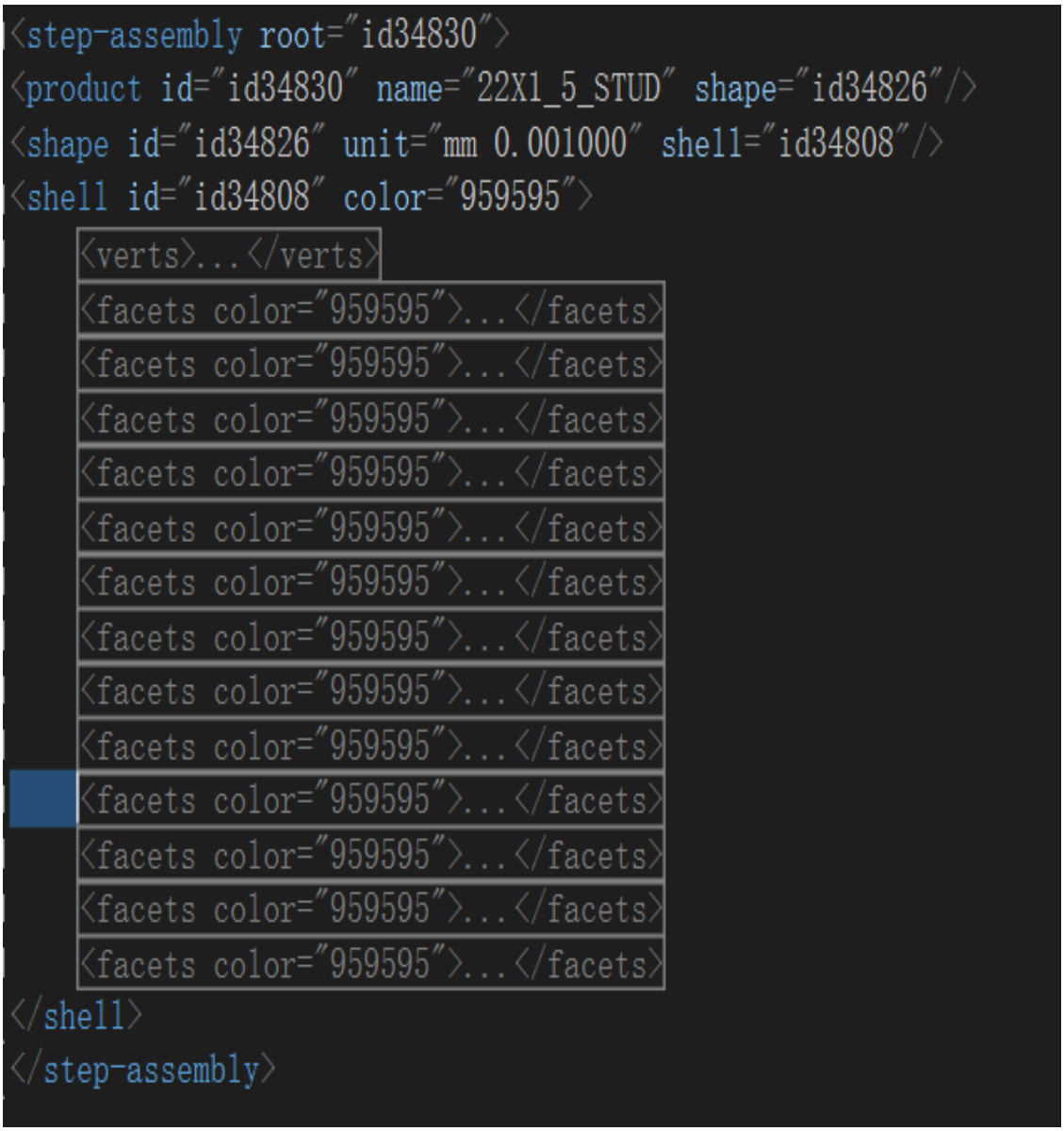



图 4-1-2 数据文件中顶点以及三角面片数据

**Step3：**依据上一章中对数据格式的说明，编写 C++程序，对 data.xml 文件进行读取，相应的读取函数 loadXML()代码如下。

关键代码解释

1、 关于字符串里“40 30 38”之内的数字，采用 stringstream 方法进行读取；

2、 doc.FirstChildElement()代表指向当前文件根目录

3、 NextSiblingElement()代表指向下一个同级标签根目录

4、 const char *attr2 = elem4->Attribute("fn")代表提取 elem4 元素下标签 fn 的值

## 代码片段展示

### 函数 loadXML()：用于读取 data.xml

```
void loadXML() {
    /*
    * 加载 XML 文件
    */
    TiXmlDocument doc;//申明一个文档类型变量，用来存储读取的 xml 文档
    //检测 xml 文档是否存在
    if(!doc.LoadFile("E:\\data\\data.xml"))
    {
        cerr << doc.ErrorDesc() << endl;
    }
    TiXmlElement* root = doc.FirstChildElement();//指向 xml 文档的根元素
    if (root == NULL)//检测根元素存在
    {
        cerr << "Failed to load file: No root element." << endl;
        doc.Clear();
    }
    //elem 指向根的第一个孩子元素
    // root is <step-assembly root="id34830">
    for (TiXmlElement* elem = root->FirstChildElement(); elem != NULL; elem =
        elem->NextSiblingElement()) {
        const char* arr = elem->Attribute("id"); //通过元素属性寻找元素
        // elem is <shell id="id34808" color="959595">
        if (strcmp(arr, "id34808") == 0) {
            TiXmlElement* elem1 = elem->FirstChildElement();
            // Step1：提取顶点坐标，并顺序标记
            // elem1 is <verts>
            for (TiXmlElement* elem2 = elem1->FirstChildElement(); elem2 != NULL;
                elem2 = elem2->NextSiblingElement()) {
                // elem2 is <v p="0 -15 75"/>
                const char* attr0 = elem2->Attribute("p");
                stringstream ss0(attr0);
                numV = numV + 1;
                vertex[numV].SN = numV; // 记录顶点的序号
                ss0 >> vertex[numV].x;
                ss0 >> vertex[numV].y;
                ss0 >> vertex[numV].z;
            }
            // Step2：提取面片
            for (TiXmlElement* elem3 = elem1->NextSiblingElement();
                elem3 != NULL; elem3 = elem3->NextSiblingElement()) {
                // elem3 is <facets color="959595">
                numM = numM + 1;
                meshes[numM].SN = numM;
                for (TiXmlElement* elem4 = elem3->FirstChildElement();
                    elem4 != NULL; elem4 = elem4->NextSiblingElement()) {
                    // elem4 is <f v="43 28 1" fn="0 4.93054498004026e-014 1">
                    // v is 顶点序号
                    const char* attr1 = elem4->Attribute("v");
                    stringstream ss1(attr1);
                    numF = numF + 1;
                    facet[numF].SN = numF;
                    ss1 >> facet[numF].v1;
                    ss1 >> facet[numF].v2;
                    ss1 >> facet[numF].v3;
                    // xml 文档里的顶点序号从 0 开始，需要更新
                    facet[numF].v1 = facet[numF].v1 + 1;
                    facet[numF].v2 = facet[numF].v2 + 1;
                    facet[numF].v3 = facet[numF].v3 + 1;
                    // fn is 面片的法向量
                    const char* attr2 = elem4->Attribute("fn");
                    stringstream ss2(attr1);
```

```
                    ss2 >> facet[numF].fn.x;
                    ss2 >> facet[numF].fn.y;
                    ss2 >> facet[numF].fn.z;
                    // n is 顶点的法向量
                    int i = 0;
                    for (TiXmlElement* elem5 = elem4->FirstChildElement();
                        elem5 != NULL; elem5 = elem5->NextSiblingElement()) {
                        // elem5 is <n d="0 0 1"/>
                        const char* attr3 = elem5->Attribute("d");
                        stringstream ss3(attr3);
                        ss3 >> facet[numF].vn[i].x;
                        ss3 >> facet[numF].vn[i].y;
                        ss3 >> facet[numF].vn[i].z;
                        i++;
                    }
                    // 将三角面片加入当前曲面
                    meshes[numM].mesh.push_back(facet[numF].SN);
                }
            }
        }
    }
    cout << "--------------加载 data.xml 文件，完成！--------------" << endl;
    cout << "顶点的数量：" << numV << endl;
    cout << "面片的数量：" << numF << endl;
    cout << "曲面的数量：" << numM << endl;

}
```

实验结果

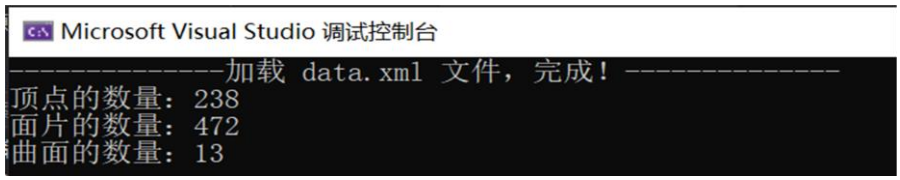



图 4-1-3 data.xml 数据里顶点、三角面片、曲面的数量

# 4.2 数据结构的设计

## 4.2.1 顶点的数据结构

关键代码解释

1、 Double x、y、z；//记录三维坐标下的顶点位置信息

2、 Int SN； // 记录顶点的编号（从 1 开始，注意数据中是从 0 开始）

3、 vector<int> facetSN; // 邻接三角面片的序号集合

4、 vector<int> edgeSN; // 邻接边的序号集合

代码片段展示

顶点的数据结构（Vertex）

```
struct Vertex { // 顶点
    double x, y, z;
    int SN; // Serial number
    Vertex() {};
    /*
     * 每一个顶点都有所属面片的序号集合
     * 每一个顶点都有所属边的序号集合
     * 每一个顶点都有所属曲面的序号集合
     */
    vector<int> facetSN; // 邻接三角面片的序号集合
    vector<int> edgeSN; // 邻接边的序号集合

    Vertex(double tx, double ty, double tz, int tSN) {
        x = tx;
        y = ty;
        z = tz;
        SN = tSN;
    }
} vertex[NUM];
```

### 4.2.2 边的数据结构

关键代码解释

1、 int v1、v2；//记录边的两个顶点的序号，规定 v1 始终小于 v2

2、 Int SN； // 记录边的编号（从 1 开始）

3、 Int MeshSN;// 所属曲面的序号

代码片段展示

边的数据结构（Edge）

```
struct Edge { // 边
    int v1, v2; // 为了区分无向边，规定v1始终小于v2
    int SN; // Serial number
    int MeshSN; // 所属曲面序号
    Edge() {};

    Edge(int tv1, int tv2) {
        v1 = tv1;
        v2 = tv2;
        SN = 0;
        MeshSN = 0;
    };
    }
edge[NUM
]
```

### 4.2.3 三角面片的数据结构

关键代码解释

1、 Int v1、v2、v3；//记录三角面片的三个端点序号

2、 Int SN； // 记录三角面片的编号（从 1 开始）

3、 Vertex fn； // 面的法向量

4、 Vertex vn[3]；//三个顶点的法向量

代码片段展示

三角面片的数据结构（Facet）

```
struct Facet { // 三角面片
    int v1, v2, v3; // 顶点序号
    int SN; // Serial number
    Vertex fn; // 面法向量
    Vertex vn[3]; // 三个顶点的法向量
    Facet() {}
    }
facet[NUM];
```

### 4.2.4 三维曲面的数据结构

关键代码解释

1、 vector<int> mesh; // 记录曲面内的三角面片序号集合

2、 vector<int> vertex; // 有序边界点集合（可能有多个闭环）

3、 vector<int> edge; // 有序边界边集合

4、 vector<vector<int>> borderVerts; // 边界点集合（只有一个闭环）

代码片段展示

三维曲面的数据结构（Mesh）

```
struct Meshes { // 三维曲面
    vector<int> mesh; // 记录曲面内的三角面片序号集合
    vector<int> vertex; // 有序边界点集合（可能有多个闭环）
    vector<int> edge; // 有序边界边集合
    vector<vector<int>> borderVerts; // 边界点集合（只有一个闭环）
    int SN; // 序号
    Meshes() {}
    }
meshes[NUM];
```

## 4.3 三角面片的绘制

关键代码解释

1、 glBegin(GL_TRIANGLES);//绘制三角面片的专用函数

2、 glColor3f(1, 0, 0);// 设置绘制颜色为红色

3、 glNormal3d(vn1.x, vn1.y, vn1.z);// 提供顶点的法向量

4、 glVertex3d(v1.x, v1.y, v1.z); // 提供顶点的三维坐标

代码片段展示

三角面片的绘制

```
void drawTriangularMeshModel() {
    /*
     * 绘制三角网格
     */
    int w = glutGet(GLUT_WINDOW_WIDTH);
    int h = glutGet(GLUT_WINDOW_HEIGHT);
    Viewport(0, 0, w, h);
    for (int i = 1; i <= numF; i++) {
        Vertex v1 = vertex[facet[i].v1];
        Vertex v2 = vertex[facet[i].v2];
        Vertex v3 = vertex[facet[i].v3];
        Vertex vn1 = facet[i].vn[0];
        Vertex vn2 = facet[i].vn[1];
        Vertex vn3 = facet[i].vn[2];
        glBegin(GL_TRIANGLES);
        glColor3f(1, 0, 0);
        glNormal3d(vn1.x, vn1.y, vn1.z);
        glVertex3d(v1.x, v1.y, v1.z);
        glNormal3d(vn2.x, vn2.y, vn2.z);
        glVertex3d(v2.x, v2.y, v2.z);
        glNormal3d(vn3.x, vn3.y, vn3.z);
        glVertex3d(v3.x, v3.y, v3.z);
        glEnd();
        glFlush(); // 保证上述命令立即执行
    }
    }
```

实验结果

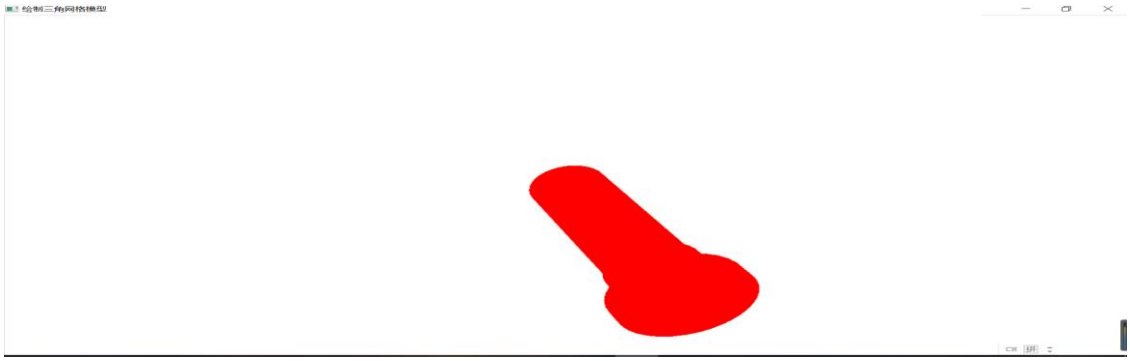


图 4-3-1 三角网格模型的绘制结果

# 4.4 边界点提取算法的设计与优化

## 4.4.1 概念定义

✧ 参考论文《一种三角网格模型的边界提取快速算法_陈有兰》 关键内容如下

设表示三维网格曲面 $M=(v_1,v_2,\cdots,v_n)$，对于 $M$ 中的任意点 $V_i=(x_i,y_i,z_i)$，用 $NI^1(v_i)$ 表示所有包含顶点 $v_i$ 的三角面片集合，则称该顶点 $v_i$ 为三角网格 $M$ 的顶点，称这些三角形为网格该顶点的邻接三角网格(邻接三角形)；将一个顶点 $v_i$ 与该顶点相连的顶点 $v_j(j\neq i)$ 组成的边 $L(v_i)=l_{ij}=\{v_j\mid\exists edge(v_i,v_j)\}$，称为该顶点的邻接边。对于顶点 $v_i$，其邻接三角形的个数用 $|NI^1(v_i)|$ 表示，其邻接边的条数用 $|L(v_i)|$ 表示，即可得到边界点的定义如下：

$v_i$为边界点$=\{v_i\mid|NI^1(v_i)|\neq|L(v_i)|\}$ (1)

特别的，如果一个顶点的邻接三角形的个数和邻接边的条数相等，则这个点为内点。否则，称之为边界点。相关定

✧ 根据文献《一种三角网格模型的边界提取快速算法》(作者：陈有兰）关于边界点的定义：“在同一三角网格曲面中，顶点邻接的三角面片数量不等于邻接的边数，则认为该顶点属于边界点。” 例如，图 4-4-1 所示，点 G 为边界点，而点 H 为内点。

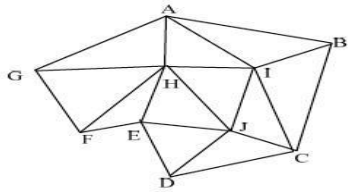



图 4-4-1 空间三角网格

## 4.4.2 边界点提取算法的设计

对于顶点 A 来说，假设其位于曲面 M 中的一个三角面片 F 中，而 F 有三个顶点 A、B、C。显而易见，边 AB、AC 为顶点 A 的邻接边，而 F 为顶点 A 的邻接三角面片。可以得到，顶点 A 的邻接边是由 A 与其处于同一面片的另外两个端点构成。

例如图 4-4-1 中，顶点 H 的邻接点集合{A、 I、 J、 E、 F、 G}，H 与其邻接点集合一对一组成的边即为其邻接边集合{AH、 GH、 FH、 EH、 JH、 IH}。

为了进行快速查找和存储便捷，通过设置一个 flag 能减少边界点的多余计算

```
if (adjacentTriangle[j] != adjacentEdge[j].size() and !flagBoundPoint[j]) {
    boundPoint.push_back(j);
    flagBoundPoint[j] = true;
}
```

而对于统计顶点的邻接三角面片则十分简单，只需统计该顶点在当前曲面内的哪些三角面片内出现即可，时间效率极高。

## 4.4.3 边界点提取算法的实现

- 关键代码解释

  if (adjacentTriangle[j] != adjacentEdge[j].size() and !flagBoundPoint[j])

  上述判断代码能判断该点是否满足边界点属性且未设置过

- 代码片段展示

**提取边界点**

```
bool flagBoundPoint[NUM];
for (int i = 0; i < NUM; i++)flagBoundPoint[i] = false;
for (int i = 1; i <= numM; i++) {
    map<int, set<int>> adjacentEdge; //顶点的邻接边
    map<int, int> adjacentTriangle; //顶点邻接三角形
    for (int j = 0; j < meshes[i].mesh.size(); j++) {
        int k = meshes[i].mesh[j]; // 当前曲面的三角面片的序号
        int v1 = facet[k].v1;
        int v2 = facet[k].v2;
        int v3 = facet[k].v3;
        //统计顶点的邻接边的另一个端点集合
        adjacentEdge[v1].insert(v2);
        adjacentEdge[v1].insert(v3);
        adjacentEdge[v2].insert(v1);
        adjacentEdge[v2].insert(v3);
        adjacentEdge[v3].insert(v1);
        adjacentEdge[v3].insert(v2);
        //统计邻接三角形面片
        adjacentTriangle[v1]++;
        adjacentTriangle[v2]++;
        adjacentTriangle[v3]++;
    }
    for (int j = 1; j <= numV; j++) {
        if (adjacentTriangle[j] == 0) {
            continue; // 未出现在该曲面上
        }
        if (adjacentTriangle[j] != adjacentEdge[j].size() and !flagBoundPoint[j]) {
            boundPoint.push_back(j);
            flagBoundPoint[j] = true;
        }
    }
     }
```

- 实验结果

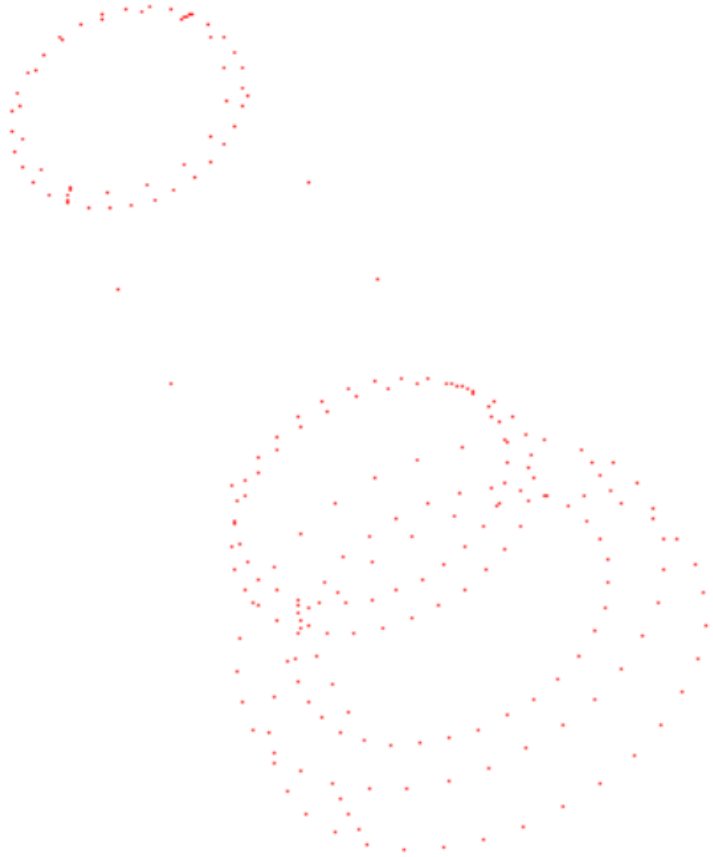


图 4-4-3 提取边界点的绘制结果

- 绘制边界点函数

**绘制边界点**

```
void showBoundPoint() {
    int w = glutGet(GLUT_WINDOW_WIDTH);
    int h = glutGet(GLUT_WINDOW_HEIGHT);
    Viewport(0, 0, w, h);
    glBegin(GL_POINTS);
    for (int i = 0; i < boundPoint.size(); ++i) {
        int t = boundPoint[i];
        glColor3f(1, 0, 0);
        glNormal3d(vertex[t].x, vertex[t].y, vertex[t].z);
        glVertex3d(vertex[t].x, vertex[t].y, vertex[t].z);
    }
    glEnd();
    glFlush(); // 保证上述命令立即执行
}
```

# 4.5 边界边提取算法的设计与优化

## 4.5.1 概念定义

✧ 参考论文《一种三角网格模型的边界提取快速算法_陈有兰》 关键内容如下

如果网格数据 $M=(v_1,v_2,\cdots,v_n)$ 中的 2 个点 $v_i,v_j$ 的连接线段 $l_{ij}$ 为某一三角网格 $\Delta v_iv_jv_k$ 的边，则称这 2 个点 $v_i,v_j$ 为这个三角网格 $\Delta v_iv_jv_k$ 的邻接点。用 $|NI^1(v_i)|$ 表示与顶点 $v_i$ 邻接的三角形网格数，则边界边可定义为：如果三角网格 $\Delta v_iv_jv_k$ 一条边 $l_{ij}$ 的邻接三角形个数 $|NI^1(v_i)|$ 为 1，则该边为边界边，否则为内边。边界边的 2 个顶点为边界点，拥有 1 个或 2 个边界点的三角形称之为边界三角形。

✧ 根据文献《一种三角网格模型的边界提取快速算法》（作者：陈有兰）关于边界边的定义：“在同一三角网格曲面中，有一条边其邻接三角面片的数量只有一个，那么这条边则为边界边，且该边的两个端点则为边界点。” 例如，图 4-5-1 所示，三个相关概念定义清晰简单。

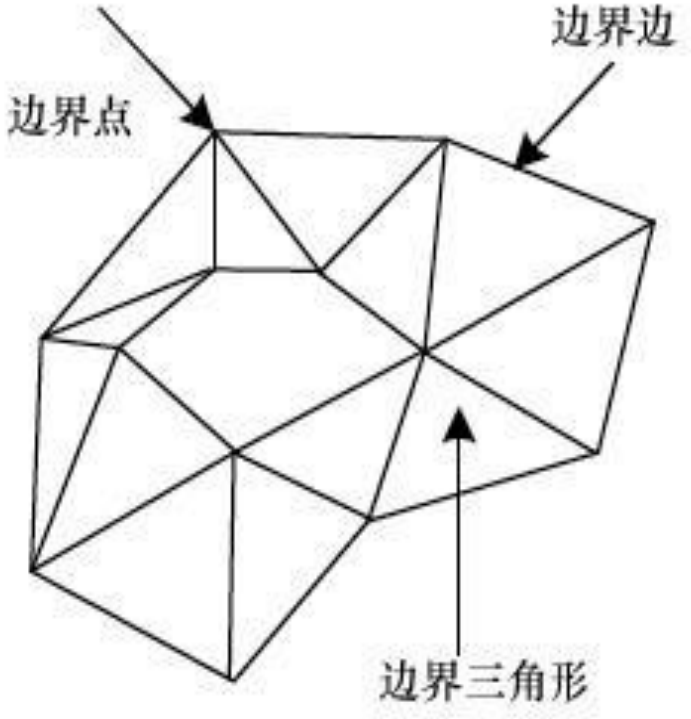
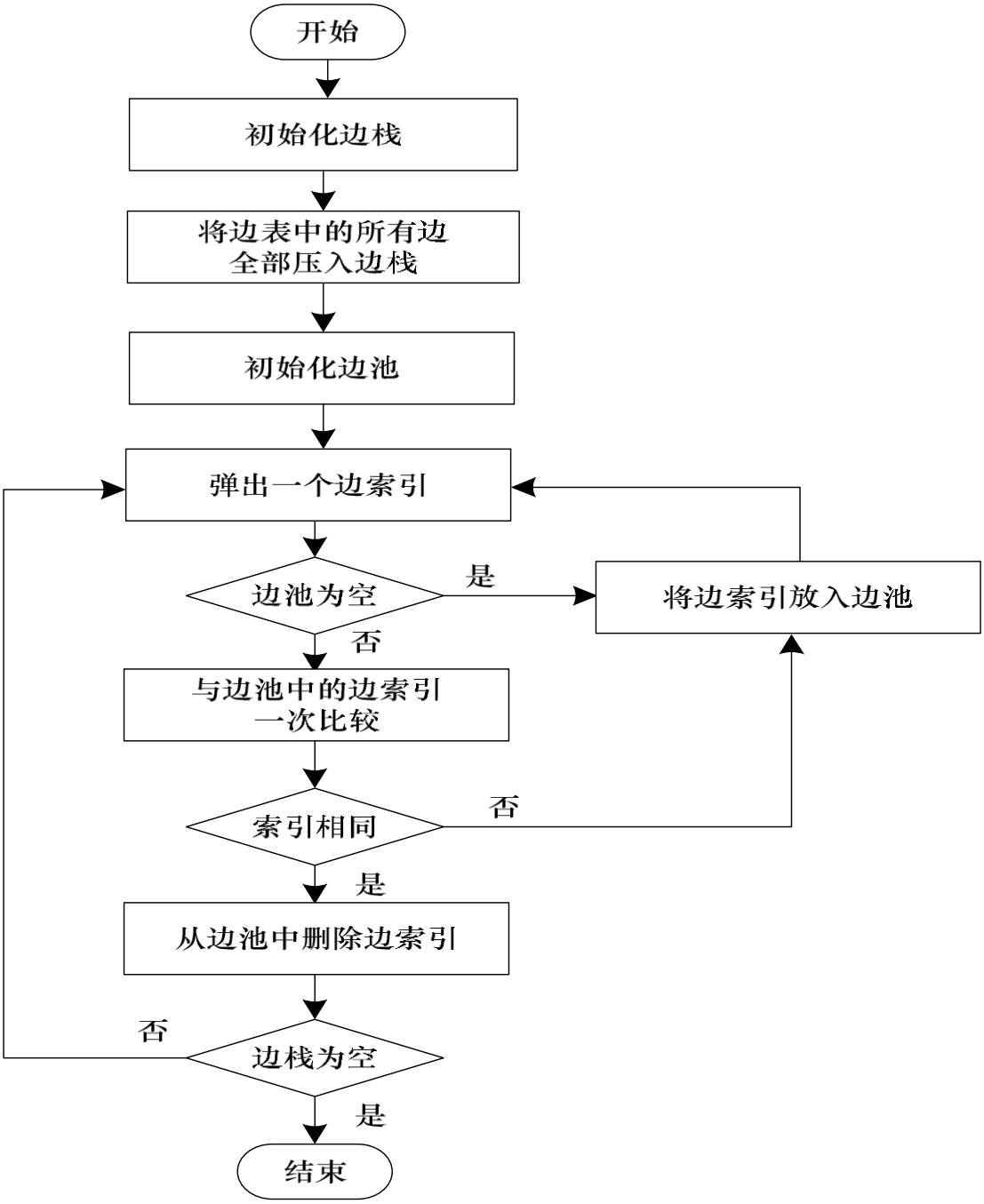



图 4-5-1 相关定义

## 4.5.2 边界边提取算法的设计




边界提取算法流程

## 4.5.3 边界边提取算法的实现

关键代码解释

1、 map<int, int> edgeAllBound; //所有边集合，用 map 实现 key-value

2、 edgeAllBound[v[0] * 10000 + v[1]]++; //实现一一映射关系

3、 sort(v, v + 3); // 利用快速排序，进行序号默认升序

4、 int v1 = v / 10000, v2 = v % 10000;//通过边 ID 进行顶点序号的提取

代码片段展示

**提取边界边**

```
void setBoundEdge(){
  for (int i = 1; i <= numM; i++) {
        map<int, int> edgeAllBound;
        for (int j = 0; j < meshes[i].mesh.size(); j++) {
            int k = meshes[i].mesh[j]; // 当前曲面的一个三角面片的序号
            int v[3] = { facet[k].v1, facet[k].v2, facet[k].v3 };
            sort(v, v + 3);
            // 顶点的个数小于10000
            edgeAllBound[v[0] * 10000 + v[1]]++;
            edgeAllBound[v[0] * 10000 + v[2]]++;
            edgeAllBound[v[1] * 10000 + v[2]]++;
        }
        for (auto item = edgeAllBound.begin(); item != edgeAllBound.end(); item++) {
            if (item->second > 1) {
                continue;
            }
            int v = item->first;
            int v1 = v / 10000, v2 = v % 10000;
            // 边界边的两个顶点一定为边界点
            numE = numE + 1;
            edge[numE].SN = numE;
            edge[numE].v1 = v1;
            edge[numE].v2 = v2;
            edge[numE].MeshSN = i; // 记录该边属于哪一曲面
            boundEdge.push_back(edge[numE].SN);
            meshes[i].edge.push_back(edge[numE].SN); //将边界边记录在该曲面中
           }
        {…}
    }
    {…}
}
```

- 实验结果

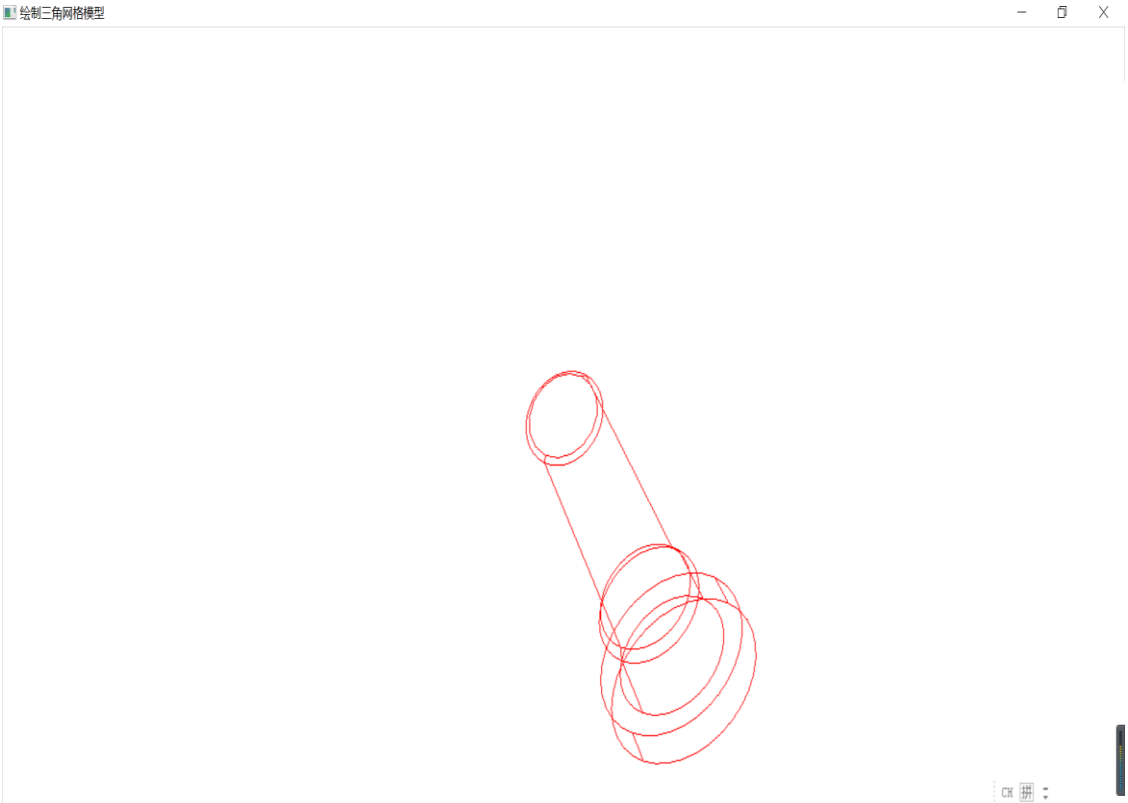



图 4-5-3 提取边界边的绘制结果

- 绘制边界点函数

**绘制边界边**

```
void showBoundEdge() {
    int w = glutGet(GLUT_WINDOW_WIDTH);
    int h = glutGet(GLUT_WINDOW_HEIGHT);
    Viewport(0, 0, w, h);
    for (int i = 0; i < boundEdge.size(); ++i) {
        Vertex t1 = vertex[edge[boundEdge[i]].v1];
        Vertex t2 = vertex[edge[boundEdge[i]].v2];
        glBegin(GL_LINES);
        glColor3f(1, 0, 0);
        glVertex3d(t1.x, t1.y, t1.z);//指定第一个点
        glVertex3d(t2.x, t2.y, t2.z);//指定第二个点
        glEnd();
    }
    glFlush(); // 保证上述命令立即执行
}
```

# 4.6 基于 B 样条的轮廓曲线拟合

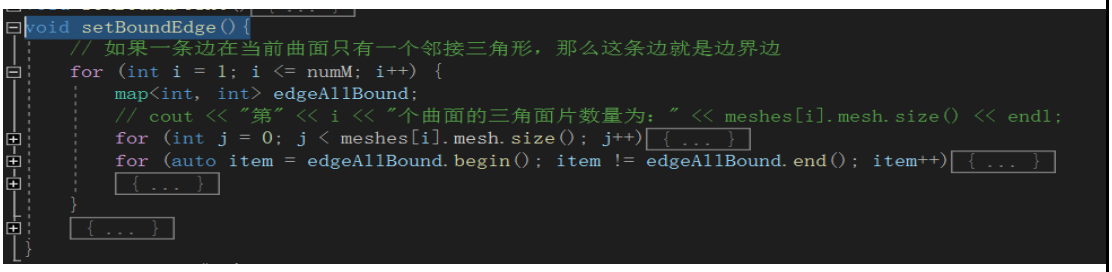
## 4.6.1 有序控制点的获取

由于我们获取了边界点，但其是一个无规律、无正常排列的点集合。我们可以想象到，只有按照边界边那样顺序的边界点集合才可以画出正常的轮廓图。

代码片段展示

**有序控制点的获取**

// 依据当前曲面点边界边集合，进行有序边界点的获取



```
void setBoundEdge(){
    // 如果一条边在当前曲面只有一个邻接三角形，那么这条边就是边界边
    for (int i = 1; i <= numM; i++) {
        map<int, int> edgeAllBound;
        // cout << "第" << i << "个曲面的三角面片数量为：" << meshes[i].mesh.size() << endl;
        for (int j = 0; j < meshes[i].mesh.size(); j++) { ... }
        for (auto item = edgeAllBound.begin(); item != edgeAllBound.end(); item++) { ... }
        { ... }
    }
    { ... }
}
```

在上图中，两个{}所包围的隐藏代码即{...}{...}借助边界边的边界点来进行排序

第一个{...}内容如下：

```
  {// 依据当前曲面点边界边集合，进行有序边界点的获取
            cout << "曲面" << i << "的边界边集合：";
            for (int j = 0; j < meshes[i].edge.size(); j++) {
                int v2 = edge[meshes[i].edge[j]].v2; //边的右端点
                for (int k = j + 1; k < meshes[i].edge.size(); k++) {
                    int tv1 = edge[meshes[i].edge[k]].v1;
                    int tv2 = edge[meshes[i].edge[k]].v2;
                    if (tv1 == v2 or tv2 == v2) {
                        int t = meshes[i].edge[j + 1];
                        meshes[i].edge[j + 1] = meshes[i].edge[k];
                        meshes[i].edge[k] = t;
                        if (tv2 == v2) {
                            t = edge[meshes[i].edge[j + 1]].v1;
                            edge[meshes[i].edge[j + 1]].v1 = edge[meshes[i].edge[j +
1]].v2;
                            edge[meshes[i].edge[j + 1]].v2 = t;
                        }
                    }
                }
                int v1 = edge[meshes[i].edge[j]].v1; //边的左端点
                meshes[i].vertex.push_back(v1);
                cout << v1 << " " << v2 << " ";
            }
            cout << endl;
            cout << "曲面" << i << "的边界点集合：";
            for (int j = 0; j < meshes[i].vertex.size(); j++) {
                cout << meshes[i].vertex[j] << " ";
            }
            cout << endl;
        }
```

第二个{...}内容如下：

```
{// 一条边的两端都是边界点的线，不一定是边界边
        cout << "边界边的数量：" << boundEdge.size() << endl;

        // 根据边界边，对边界点进行有序的排列，使其变成闭环
        /*
         * 假设现在的vector<Edge> edges = {CD、CB、EA、BA、DE}
         * 则需要将其排序为{CD、DE、EA、AB、BC}，然后遍历边界边
         * 将每条边的vIndex1加入边界点边界点集合就构造完成了，为{C、D、E、A、B}
         */
         // 一个曲面内可能有多个闭合的边界边集合
        for (int i = 1; i <= numM; i++) {
            int len = meshes[i].edge.size() - 1;
            // 若首尾不相连，意味着该曲面有多个闭环
            int leftPtr = 0; // 左指针
            int rightPtr = 2; // ptr指向最后意味着结束，闭环至少三条边，所以从2开始
            while (rightPtr <= len) {
                for (int j = rightPtr; j < meshes[i].edge.size(); j++) {
                    if (edge[meshes[i].edge[j]].v2 == edge[meshes[i].edge[leftPtr]].v1) {
                        vector<int> temp;
                        for (int k = leftPtr; k <= j; k++) {
                            temp.push_back(edge[meshes[i].edge[k]].v1);
                        }
                        meshes[i].borderVerts.push_back(temp);
                        leftPtr = j + 1;
                        rightPtr = j + 3;
                        break;
                    }
                }
            }
            cout << "曲面" << i << "边界点集合的个数：" << meshes[i].borderVerts.size() <<
endl;
        }
    }
```

- 实验结果

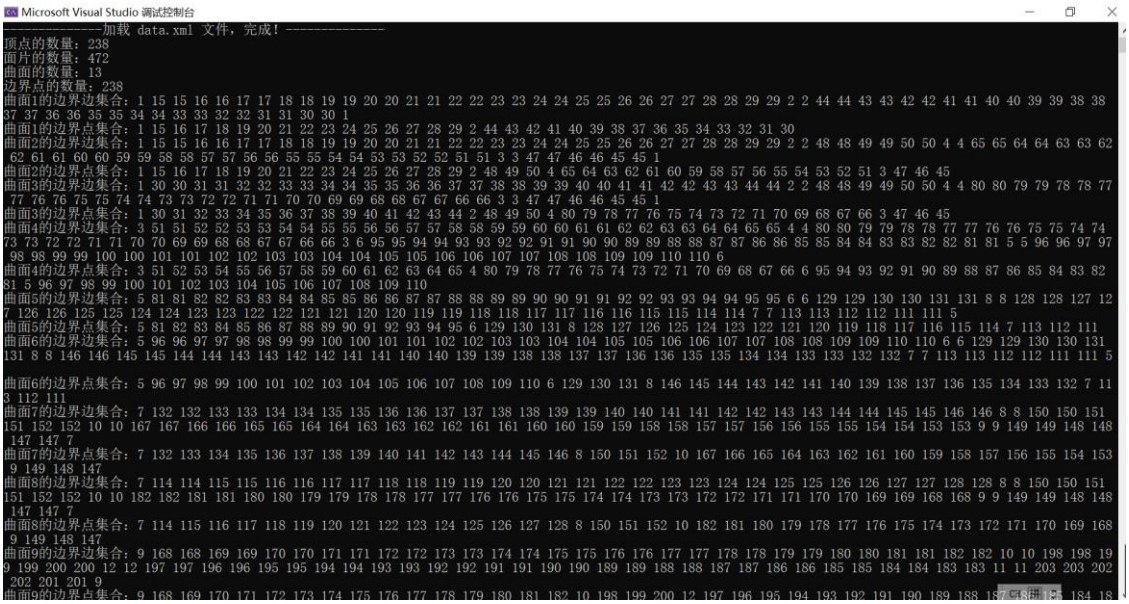



图 4-6-1 曲面的有序控制点集合结果（部分）

## 4.6.2 曲面内多个闭环集合的分离

值得一提的是，有一个很大的问题存在，实验过程中发现曲面 4（即第四个曲面），有两个闭环边界，相当于双圆环，如图 4-6-2 所示。

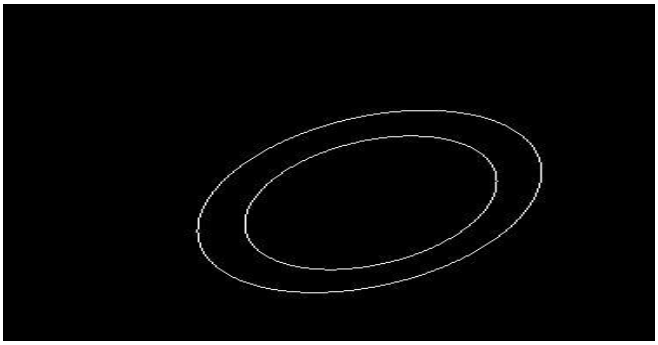


图 4-6-2 data.xml 里第四个曲面的绘制图

一般情况下，一个曲面的有序边界集合只有一个闭环，那么我们之间进行 B 样条拟合就可以了，但有两个甚至多个闭环存在时，无法直接拟合，需要进行分离，每一个闭环边界单独拟合。

解决思路十分的简单，我们只需将首尾序号相接的有序边集合的顶点纳入控制点集合就可以了，假设有多个闭环边界，那么将创建多个控制点集合存储对应的一个闭环边界。

- 代码片段展示

曲面内多个闭环集合的分离

```
// 一个曲面内可能有多个闭合的边界边集合
for (int i = 1; i <= numM; i++) {
            int len = meshes[i].edge.size() - 1;
            // 若首尾不相连，意味着该曲面有多个闭环
            int leftPtr = 0; // 左指针
            int rightPtr = 2; // ptr指向最后意味着结束，闭环至少三条边，所以从2开始
            while (rightPtr <= len) {
                for (int j = rightPtr; j < meshes[i].edge.size(); j++) {
                    if (edge[meshes[i].edge[j]].v2 ==
edge[meshes[i].edge[leftPtr]].v1) {
                        vector<int> temp;
                        for (int k = leftPtr; k <= j; k++) {
                            temp.push_back(edge[meshes[i].edge[k]].v1);
                        }
                        meshes[i].borderVerts.push_back(temp);
                        leftPtr = j + 1;
                        rightPtr = j + 3;
                        break;
                    }
                }
            }
            cout << "曲面" << i << "边界点集合的个数：" <<
meshes[i].borderVerts.size() << endl;
        }
```

- 实验结果

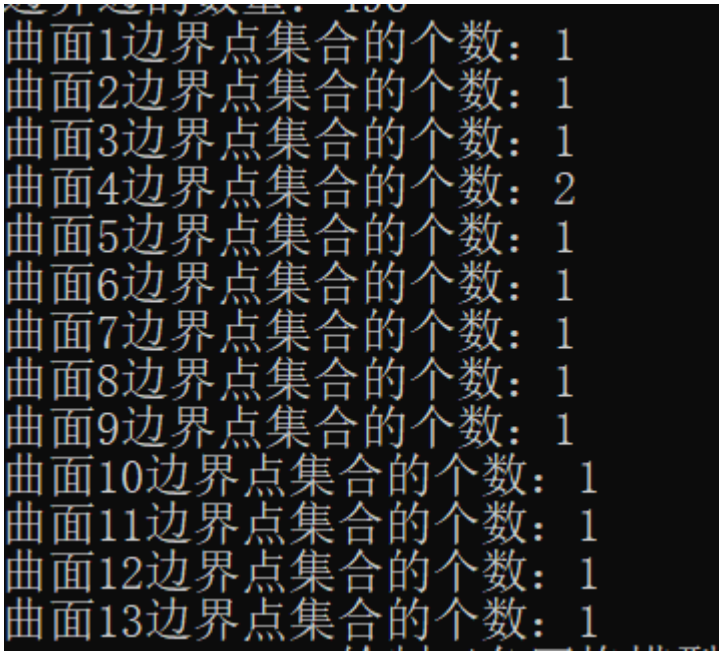



图 4-6-3 data.xml 文件中各个曲面闭环集合数量

### 4.6.3 基于 B 样条拟合轮廓

获得每一个曲面的闭环边界点有序集合后，我们就可以直接进行 B 样条曲线拟合了。此处，我们引用 OpenGL 自带函数 gluNurbsCurve 进行绘制。

- 函数 **gluNurbsCurve** 进行参数解释

| 函数 gluNurbsCurve 的参数 |
| --- |
| nobj<br>Specifies the NURBS object (created with gluNewNurbsRenderer).<br>nknots<br>Specifies the number of knots in knot. nknots equals the number of control points plus the order.<br>knot<br>Specifies an array of nknots nondecreasing knot values.<br>stride<br>Specifies the offset (as a number of single-precision floating-point values) between successive curve control points.<br>ctlarray<br>Specifies a pointer to an array of control points. The coordinates must agree with type, specified below.<br>order<br>Specifies the order of the NURBS curve. order equals degree + 1, hence a cubic curve has an order of 4. |

代码片段展示

B 样条曲线拟合轮廓

```
GLUnurbsObj* BCurve(float* knot, int nctls, float* ctls, int order)
{
    GLUnurbsObj* oBCurve;
    int nknots = nctls + order;
    oBCurve = gluNewNurbsRenderer();
    gluNurbsProperty(oBCurve, GLU_SAMPLING_TOLERANCE, 1);
    glEnable(GL_MAP1_VERTEX_3);
    gluBeginCurve(oBCurve);
    gluNurbsCurve(oBCurve, nknots, knot, 3, ctls, order, GL_MAP1_VERTEX_3);
    gluEndCurve(oBCurve);
    return oBCurve;
}
//我使用了4种方法实现B样条绘制，其中showBcurve1()showBcurve2()使用了GL函数库的自带函数实现；此外，showBcurve3()showBcurve4()用底层数学知识来实现绘制。
void showBcurve1() {
    float* knots=new float [boundPoint.size()];
    int n=boundPoint.size();
    Uniknots(knots, n, 4);
    float** p = new float* [boundPoint.size()];
    for (int i = 0; i < boundPoint.size(); i++) {
        p[i] = new float[3];
        p[i][0] = vertex[i].x; //把控制点数据传给变量
        p[i][1] = vertex[i].y;
        p[i][2] = vertex[i].z;
    }
    glColor3f(0.58, 0.58, 0.58);
    GLUnurbsObj* nobj;
    nobj = BCurve(knots, n, &p[0][0], 4);
}
================================================================================
void showbcurve2() {
    float** ctrlpoints;
    for (const meshes& tm : meshes)
    {
        for (const vector<int>& vertsgroup : tm.borderverts)
        {
            int cpnum = vertsgroup.size();
            for (int i = 0; i < cpnum; i++)
            {
                const vertex& v = vertex[vertsgroup[i]];
                ctrlpoints[i][0] = v.x;
                ctrlpoints[i][1] = v.y;
                ctrlpoints[i][2] = v.z;
            }

            for (int i = 0; i < power; i++, cpnum++)
            {
                const vertex& v = vertex[vertsgroup[i]];
                ctrlpoints[cpnum][0] = v.x;
                ctrlpoints[cpnum][1] = v.y;
                ctrlpoints[cpnum][2] = v.z;
```

```
            }

            int knum = cpnum + power + 1;
            glfloat interval = 1.0 / (knum - 1);
            glfloat ui = 0.0;
            for (int i = 0; i < knum; i++, ui += interval)
            {
                knots[i] = ui;
            }

            //绘制b样条曲线
            glunurbsobj* thenurb;
            glubegincurve(thenurb);
            glunurbscurve(thenurb, knum, knots, 3, &ctrlpoints[0][0], 4,
gl_map1_vertex_3);
            gluendcurve(thenurb);
        }

    }
}
================================================================================
void showBcurve3() {
    for (int i = 1; i <= numM; i++)
    {
        for (int k = 0; k < meshes[i].borderVerts.size(); ++k)
        {
            vector<int> vector1 = meshes[i].borderVerts[k];
            unsigned long cpNum = vector1.size();
            GLfloat ctrlPoints[1000][3];
            GLfloat knots[10000];
            for (int j = 0; j < cpNum; j++)
            {
                const Vertex& v = vertex[vector1[j]];
                ctrlPoints[j][0] = v.x;
                ctrlPoints[j][1] = v.y;
                ctrlPoints[j][2] = v.z;
            }
            for (int j = 0; j < 3; j++, cpNum++)
            {
                const Vertex& v = vertex[vector1[j]];
                ctrlPoints[cpNum][0] = v.x;
                ctrlPoints[cpNum][1] = v.y;
                ctrlPoints[cpNum][2] = v.z;
            }
            unsigned long kNum = cpNum + 3 + 1;
            for (int j = 0; j < kNum; j++)
            {
                knots[j] = j;
            }
            //绘制B样条曲线
            glColor3f(1, 0, 1);
            GLUnurbsObj* theNurb = gluNewNurbsRenderer();
            gluNurbsProperty(theNurb, GLU_SAMPLING_TOLERANCE, 1);
            glEnable(GL_MAP1_VERTEX_3);
            gluBeginCurve(theNurb);
```

```
            gluNurbsCurve(theNurb, kNum, knots, 3, &ctrlPoints[0][0], 4,
GL_MAP1_VERTEX_3);
            gluEndCurve(theNurb);
            glFlush();
        }
    }
}
================================================================================
void showBcurve4() {
    for (int i = 0; i < numM; i++) {
        OnDraw(i);
    }
}
void OnDraw(int num) {
    glClear(GL_COLOR_BUFFER_BIT);
    glLoadIdentity();
    glColor3f(0.3, 0, 0.5);
    glBegin(GL_LINE_STRIP);
    for (int k = 0; k < meshes[num].borderVerts.size(); k++) {
        vector<int> temp = meshes[num].borderVerts[k];
        cout << num << ":";
        for (int i = 0; i < temp.size(); i++) {
            cout << temp[i] << " ";
        }
        cout << endl;
        int size = temp.size();
        for (int start_cv = 0, j = 0; j < size; j++, start_cv++)
        {
            for (int i = 0; i != LOD; ++i) {
                float t = (float)i / LOD;
                float it = 1.0f - t;
                float b0 = it * it * it / 6.0f;
                float b1 = (3 * t * t * t - 6 * t * t + 4) / 6.0f;
                float b2 = (-3 * t * t * t + 3 * t * t + 3 * t + 1) / 6.0f;
                float b3 = t * t * t / 6.0f;
                float x = b0 * vertex[temp[GetPointNum(start_cv + 0, size)]].x+
                    b1 * vertex[temp[GetPointNum(start_cv + 1, size)]].x +
                    b2 * vertex[temp[GetPointNum(start_cv + 2, size)]].x +
                    b3 * vertex[temp[GetPointNum(start_cv + 3, size)]].x;
                float y = b0 * vertex[temp[GetPointNum(start_cv + 0, size)]].y +
                    b1 * vertex[temp[GetPointNum(start_cv + 1, size)]].y +
                    b2 * vertex[temp[GetPointNum(start_cv + 2, size)]].y +
                    b3 * vertex[temp[GetPointNum(start_cv + 3, size)]].y;
                float z = b0 * vertex[temp[GetPointNum(start_cv + 0, size)]].y +
                    b1 * vertex[temp[GetPointNum(start_cv + 1, size)]].z +
                    b2 * vertex[temp[GetPointNum(start_cv + 2, size)]].z +
                    b3 * vertex[temp[GetPointNum(start_cv + 3, size)]].z;
                glVertex3f(x, y,z);
            }
        }
    }
    glEnd();
    glFlush();

}
```

实验结果

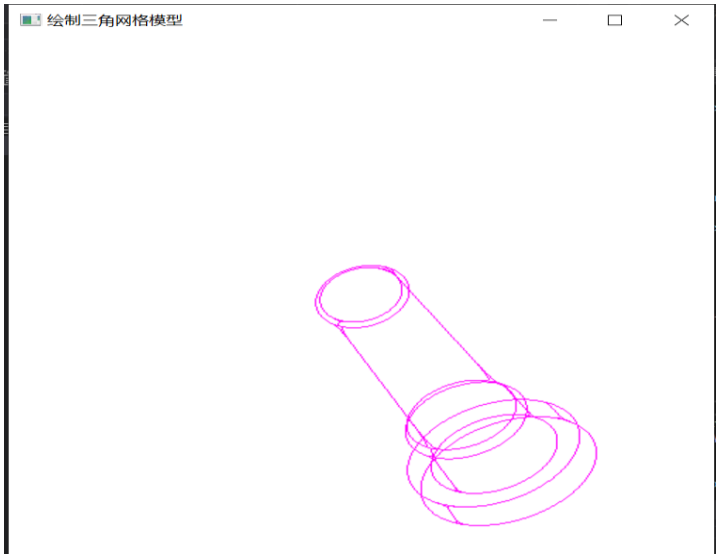



图 4-6-3 B 样条曲线拟合结果图

## 4.7 扩展性测试

根据教师提供的 2.xml 和 3.xml 文件，我进行如下绘制测试，其中 2.xml 和 3.xml 文件大小均为 11.1MB，而 data.xml 文件大小为 135KB。

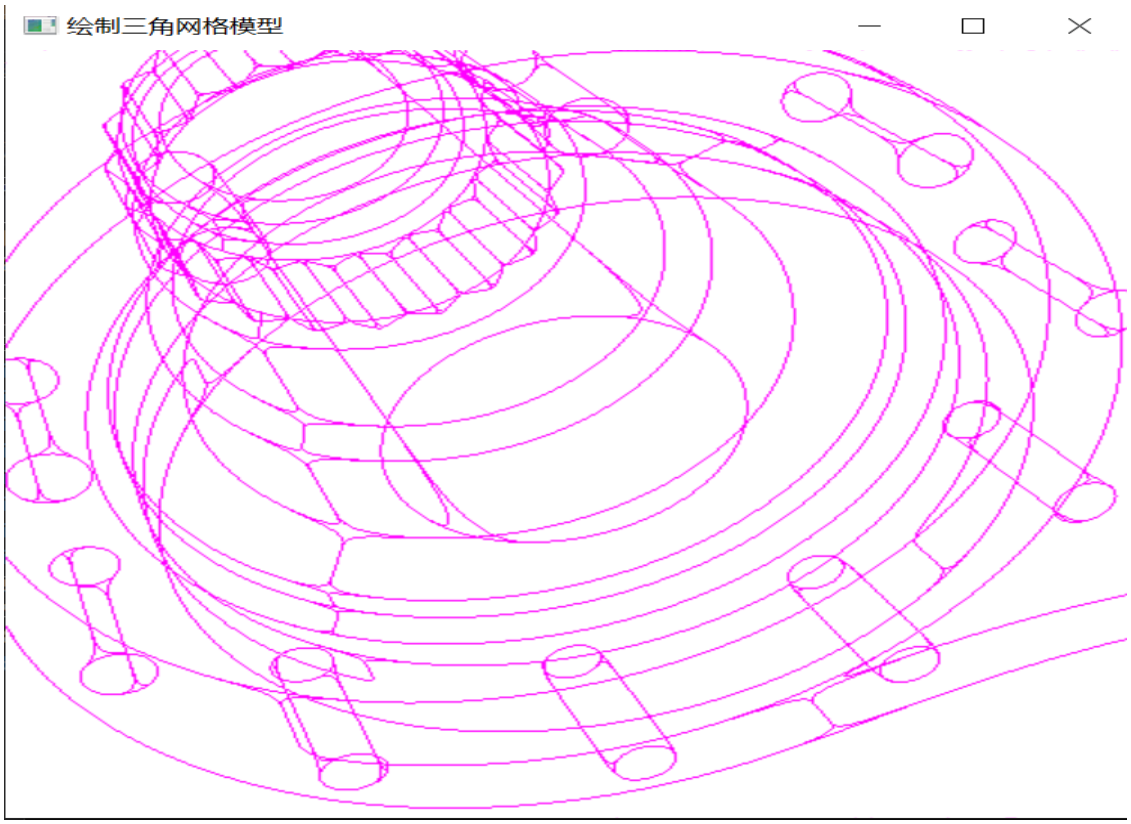



图 4-7-1 文件 2.xml 绘制结果图（耗时 0.271621 秒）

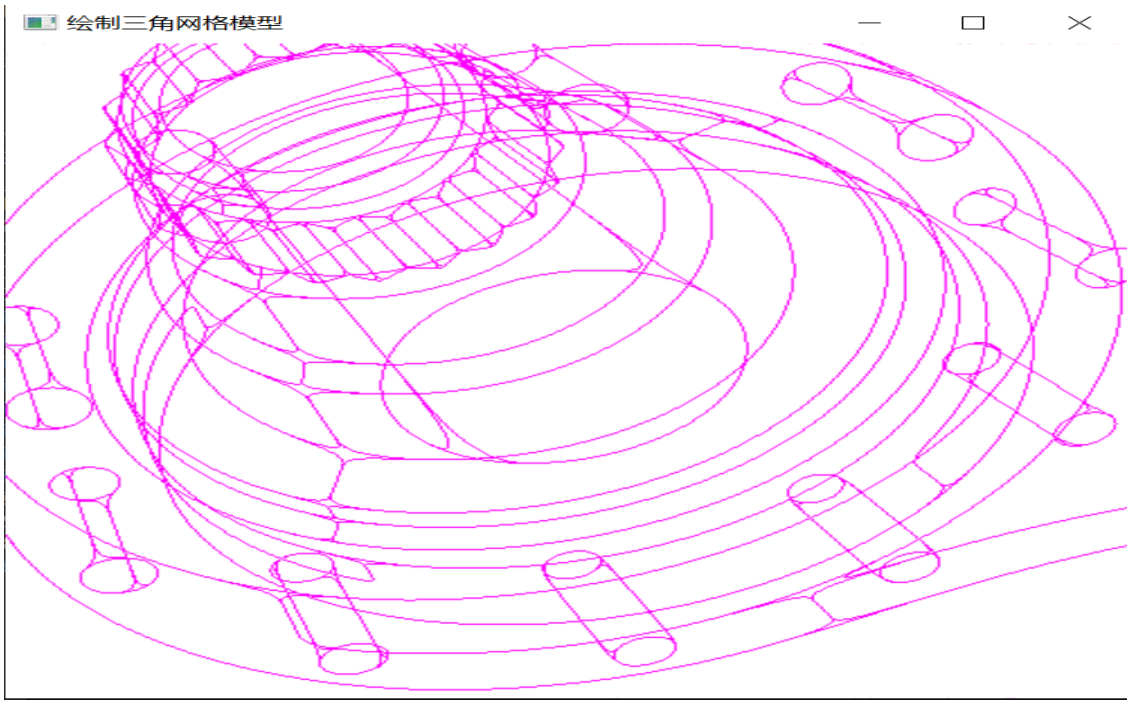


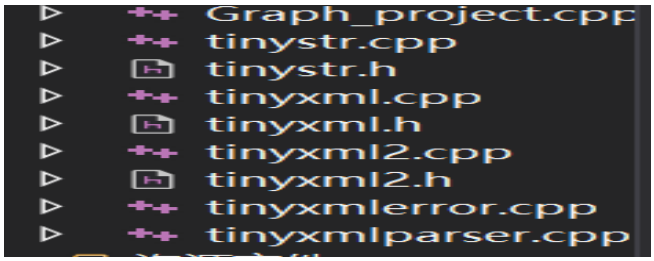

图 4-7-2 文件 3.xml 绘制结果图（耗时 0.250564 秒）


图 4-7-3 文件 data.xml 绘制结果图（耗时 0.032184 秒）

由此可见，文件数据量扩大了约 84 倍，拟合绘制时间增加了约 8.5 倍，算法效率较为良好，主要原因是算法对 key-value、map、set 以及堆的应用。

# 五、实验中遇到的问题

问题一：**TinyXML** 环境配置无效




解决方案：通过上网查阅相关资料，发现 VX2019 中，需要将相关头文件导入到项目属性附加目录中。并且我还添加了 tinyxml2 版本作为备用。

问题二：B 样条曲线绘制失败
解决方案：通过上网查阅相关资料，对 gluNurbsCurve（）函数的相关参数进行了解。

**PARAMETERS**

*nobj*
Specifies the NURBS object (created with **gluNewNurbsRenderer**).

*nknots*
Specifies the number of knots in *knot*. *nknots* equals the number of control points plus the order.

*knot*
Specifies an array of *nknots* nondecreasing knot values.

*stride*
Specifies the offset (as a number of single-precision floating-point values) between successive curve control points.

*ctlarray*
Specifies a pointer to an array of control points. The coordinates must agree with *type*, specified below.

*order*
Specifies the order of the NURBS curve. *order* equals degree + 1, hence a cubic curve has an order of 4.

*type*
Specifies the type of the curve. If this curve is defined within a **gluBeginCurve** / **gluEndCurve** pair, then the type can be any of the valid one-dimensional evaluator types (such as **GL_MAP1_VERTEX_3** or **GL_MAP1_COLOR_4**). Between a **gluBeginTrim** / **gluEndTrim** pair, the only valid types are **GLU_MAP1_TRIM_2** and **GLU_MAP1_TRIM_3**.

问题三：c++二级指针出错
解决方案：通过上网查阅相关资料，我了解到了，c++二维数组使用时需要先初始化分配空间；否则在作为函数参数传入时会出错。

# 六、实验总结

本次期末大实验锻炼了我在图形学课程方面收到实践技能。本学期的图形学课程只有 8 节课，但所学知识较为复杂，其中也涉及到了大量的数学知识和相关函数库。在平时上课过程中，知识点较多，通过实验和大作业弥补了我在实战应用方面的欠缺。

在本次大作业中，我进行了三角网格模型绘制、边界边边界点提取以及 B 样条曲线绘制。通过这些实验的步骤和过程，我进一步了解了 GLUT 的相关函数库，也对相关算法实现有了更多的学习。

在对边界点和边界边提取算法的优化时，我主要参考了老师所给的论文和科学上网，在代码效率和优化上，我使用了映射函数、key-value、set 等方法和数据结构在查询和存储方面进行深度优化。

总的来说，通过这次图形学课程，我学习到了图形学的基础知识并在实战中加以应用。同时对 GLUT 函数库有了更多的了解，了解到了许多实现函数，也了解了某一些函数的具体实现。在课程中，复杂的数学知识给我留下了较深的印象，由于画图会参考比较多的数学知识，如曲线方程、底层实现等，所以学好数学知识可以帮助我们更好地学习图形学课程也可以帮助我们优化函数底层的实现。

最后，希望在未来的生活中，能够学习到更多的图形学知识并应用到生活实践中。同时感谢简老师在学业上的指导，成为我图形学学习道路上的引路人。通过本次大作业，为我本学期的图形学课程画下了圆满的句号。

# 七、代码附录

```
//Graph_project.Cpp ===主要代码
#include <iostream>
#include"tinyxml.h"
#include"tinystr.h"
#include"tinyxml2.h"
#include<sstream>
#include<string>
#include<vector>
#include<GL/glut.h>
#include<map>
#include<algorithm>
#include<fstream>
#include<windows.h>
#include<ctime>
#include<set>
#define NUM 10000
unsigned int LOD = 20;
using namespace std;
using namespace tinyxml2;

int numV = 0; // 统计顶点的数量
int numE = 0; // 统计边的数量
int numF = 0; // 统计三角面片的数量
int numM = 0; // 统计曲面的数量

vector<int> boundPoint; //边界点序号集合
vector<int> boundEdge; //边界边序号集合
```

```
struct Vertex { // 顶点
    double x, y, z;
    int SN; // Serial number
    Vertex() {};
    /*
     * 每一个顶点都有所属面片的序号集合
     * 每一个顶点都有所属边的序号集合
     * 每一个顶点都有所属曲面的序号集合
     */
    vector<int> facetSN; // 邻接三角面片的序号集合
    vector<int> edgeSN; // 邻接边的序号集合

    Vertex(double tx, double ty, double tz, int tSN) {
        x = tx;
        y = ty;
        z = tz;
        SN = tSN;
    }
} vertex[NUM];

struct Edge { // 边
    int v1, v2; // 为了区分无向边，规定 v1 始终小于 v2
    int SN; // Serial number
    int MeshSN; // 所属曲面序号
    Edge() {};

    Edge(int tv1, int tv2) {
        v1 = tv1;
        v2 = tv2;
        SN = 0;
```

```
    MeshSN = 0;
  };
} edge[NUM];

struct Facet { // 三角面片
  int v1, v2, v3; // 顶点序号
  int SN; // Serial number
  Vertex fn; // 面法向量
  Vertex vn[3]; // 三个顶点的法向量
  Facet() {}
} facet[NUM];

struct Meshes { // 三维曲面
  vector<int> mesh; // 记录曲面内的三角面片序号集合
  vector<int> vertex; // 有序边界点集合（可能有多个闭环）
  vector<int> edge; // 有序边界边集合
  vector<vector<int>> borderVerts; // 边界点集合（只有一个闭环）
  int SN; // 序号
  Meshes() {}
} meshes[NUM];

void loadXML() {
  /*
  * 加载 XML 文件
  */
  TiXmlDocument doc;//申明一个文档类型变量，用来存储读取的 xml 文档
  //检测 xml 文档是否存在
  if(!doc.LoadFile("E:\\data\\data.xml"))
  {
    cerr << doc.ErrorDesc() << endl;
```

```
}
TiXmlElement* root = doc.FirstChildElement();//指向 xml 文档的根元素
if (root == NULL)//检测根元素存在
{
   cerr << "Failed to load file: No root element." << endl;
   doc.Clear();
}
//elem 指向根的第一个孩子元素
// root is <step-assembly root="id34830">
for (TiXmlElement* elem = root->FirstChildElement(); elem != NULL; elem =
   elem->NextSiblingElement()) {
   const char* arr = elem->Attribute("id"); //通过元素属性寻找元素
   // elem is <shell id="id34808" color="959595">
   if (strcmp(arr, "id34808") == 0) {
      TiXmlElement* elem1 = elem->FirstChildElement();
      // Step1：提取顶点坐标，并顺序标记
      // elem1 is <verts>
      for (TiXmlElement* elem2 = elem1->FirstChildElement(); elem2 != NULL;
         elem2 = elem2->NextSiblingElement()) {
         // elem2 is <v p="0 -15 75"/>
         const char* attr0 = elem2->Attribute("p");
         stringstream ss0(attr0);
         numV = numV + 1;
         vertex[numV].SN = numV; // 记录顶点的序号
         ss0 >> vertex[numV].x;
         ss0 >> vertex[numV].y;
         ss0 >> vertex[numV].z;
      }
      // Step2：提取面片
      for (TiXmlElement* elem3 = elem1->NextSiblingElement();
```

```
elem3 != NULL; elem3 = elem3->NextSiblingElement()) {
// elem3 is <facets color="959595">
numM = numM + 1;
meshes[numM].SN = numM;
for (TiXmlElement* elem4 = elem3->FirstChildElement();
  elem4 != NULL; elem4 = elem4->NextSiblingElement()) {
  // elem4 is <f v="43 28 1" fn="0 4.93054498004026e-014 1">
  // v is 顶点序号
  const char* attr1 = elem4->Attribute("v");
  stringstream ss1(attr1);
  numF = numF + 1;
  facet[numF].SN = numF;
  ss1 >> facet[numF].v1;
  ss1 >> facet[numF].v2;
  ss1 >> facet[numF].v3;
  // xml 文档里的顶点序号从 0 开始，需要更新
  facet[numF].v1 = facet[numF].v1 + 1;
  facet[numF].v2 = facet[numF].v2 + 1;
  facet[numF].v3 = facet[numF].v3 + 1;
  // fn is 面片的法向量
  const char* attr2 = elem4->Attribute("fn");
  stringstream ss2(attr1);
  ss2 >> facet[numF].fn.x;
  ss2 >> facet[numF].fn.y;
  ss2 >> facet[numF].fn.z;
  // n is 顶点的法向量
  int i = 0;
  for (TiXmlElement* elem5 = elem4->FirstChildElement();
    elem5 != NULL; elem5 = elem5->NextSiblingElement()) {
    // elem5 is <n d="0 0 1"/>
    const char* attr3 = elem5->Attribute("d");
```

```
                stringstream ss3(attr3);
                ss3 >> facet[numF].vn[i].x;
                ss3 >> facet[numF].vn[i].y;
                ss3 >> facet[numF].vn[i].z;
                i++;
            }
            // 将三角面片加入当前曲面
            meshes[numM].mesh.push_back(facet[numF].SN);
        }
      }
    }
  }
  cout << "--------------加载 data.xml 文件，完成！--------------" << endl;
  cout << "顶点的数量：" << numV << endl;
  cout << "面片的数量：" << numF << endl;
  cout << "曲面的数量：" << numM << endl;

}
void Viewport(int x, int y, int w, int h)
{
  glViewport(x, y, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(30, w / (double)h, 1, 1000);
  glTranslatef(0, 0, -4);
  glScaled(0.015, 0.015, 0.015);
  glRotatef(45, 1, 1, -1);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
```

```
}
void showBoundPoint() {
   int w = glutGet(GLUT_WINDOW_WIDTH);
   int h = glutGet(GLUT_WINDOW_HEIGHT);
   Viewport(0, 0, w, h);
   glBegin(GL_POINTS);
   for (int i = 0; i < boundPoint.size(); ++i) {
      int t = boundPoint[i];
      glColor3f(1, 0, 0);
      glNormal3d(vertex[t].x, vertex[t].y, vertex[t].z);
      glVertex3d(vertex[t].x, vertex[t].y, vertex[t].z);
   }
   glEnd();
   glFlush(); // 保证上述命令立即执行
}

void showBoundEdge() {
   int w = glutGet(GLUT_WINDOW_WIDTH);
   int h = glutGet(GLUT_WINDOW_HEIGHT);
   Viewport(0, 0, w, h);
   for (int i = 0; i < boundEdge.size(); ++i) {
      Vertex t1 = vertex[edge[boundEdge[i]].v1];
      Vertex t2 = vertex[edge[boundEdge[i]].v2];
      glBegin(GL_LINES);
      glColor3f(1, 0, 0);
      glVertex3d(t1.x, t1.y, t1.z);//指定第一个点
      glVertex3d(t2.x, t2.y, t2.z);//指定第二个点
      glEnd();
   }
   glFlush(); // 保证上述命令立即执行
}
```

```
void drawTriangularMeshModel() {
    /*
     * 绘制三角网格
     */
    int w = glutGet(GLUT_WINDOW_WIDTH);
    int h = glutGet(GLUT_WINDOW_HEIGHT);
    Viewport(0, 0, w, h);

    for (int i = 1; i <= numF; i++) {

        Vertex v1 = vertex[facet[i].v1];
        Vertex v2 = vertex[facet[i].v2];
        Vertex v3 = vertex[facet[i].v3];
        Vertex vn1 = facet[i].vn[0];
        Vertex vn2 = facet[i].vn[1];
        Vertex vn3 = facet[i].vn[2];

        glBegin(GL_TRIANGLES);
        glColor3f(1, 0, 0);
        glNormal3d(vn1.x, vn1.y, vn1.z);
        glVertex3d(v1.x, v1.y, v1.z);
        glNormal3d(vn2.x, vn2.y, vn2.z);
        glVertex3d(v2.x, v2.y, v2.z);
        glNormal3d(vn3.x, vn3.y, vn3.z);
        glVertex3d(v3.x, v3.y, v3.z);
        glEnd();
        glFlush(); // 保证上述命令立即执行
    }
}

void setBoundPoint(){
```

```
bool flagBoundPoint[NUM];
for (int i = 0; i < NUM; i++)flagBoundPoint[i] = false;
for (int i = 1; i <= numM; i++) {
   map<int, set<int>> adjacentEdge; //顶点的邻接边
   map<int, int> adjacentTriangle; //顶点邻接三角形
   for (int j = 0; j < meshes[i].mesh.size(); j++) {
      int k = meshes[i].mesh[j]; // 当前曲面的三角面片的序号
      int v1 = facet[k].v1;
      int v2 = facet[k].v2;
      int v3 = facet[k].v3;
      //统计顶点的邻接边的另一个端点集合
      adjacentEdge[v1].insert(v2);
      adjacentEdge[v1].insert(v3);
      adjacentEdge[v2].insert(v1);
      adjacentEdge[v2].insert(v3);
      adjacentEdge[v3].insert(v1);
      adjacentEdge[v3].insert(v2);
      //统计邻接三角形面片
      adjacentTriangle[v1]++;
      adjacentTriangle[v2]++;
      adjacentTriangle[v3]++;
   }
   for (int j = 1; j <= numV; j++) {
      if (adjacentTriangle[j] == 0) {
         continue; // 未出现在该曲面上
      }
      if (adjacentTriangle[j] != adjacentEdge[j].size() and !flagBoundPoint[j]) {
         boundPoint.push_back(j);
         flagBoundPoint[j] = true;
      }
```

```
    }
  }
  cout << "边界点的数量：" << boundPoint.size() << endl;
}
void setBoundEdge(){
  // 如果一条边在当前曲面只有一个邻接三角形，那么这条边就是边界边
  for (int i = 1; i <= numM; i++) {
    map<int, int> edgeAllBound;
    // cout << "第" << i << "个曲面的三角面片数量为：" << meshes[i].mesh.size() << endl;
    for (int j = 0; j < meshes[i].mesh.size(); j++) {
      int k = meshes[i].mesh[j]; // 当前曲面的一个三角面片的序号
      int v[3] = { facet[k].v1, facet[k].v2, facet[k].v3 };
      sort(v, v + 3);
      // 顶点的个数小于 10000
      edgeAllBound[v[0] * 10000 + v[1]]++;
      edgeAllBound[v[0] * 10000 + v[2]]++;
      edgeAllBound[v[1] * 10000 + v[2]]++;
    }
    for (auto item = edgeAllBound.begin(); item != edgeAllBound.end(); item++) {
      if (item->second > 1) {
        continue;
      }
      int v = item->first;
      int v1 = v / 10000, v2 = v % 10000;
      // 边界边的两个顶点一定为边界点
      numE = numE + 1;
      edge[numE].SN = numE;
      edge[numE].v1 = v1;
      edge[numE].v2 = v2;
      edge[numE].MeshSN = i; // 记录该边属于哪一曲面
```

```
    boundEdge.push_back(edge[numE].SN);
    meshes[i].edge.push_back(edge[numE].SN); //将边界边记录在该曲面中
}
{// 依据当前曲面点边界边集合，进行有序边界点的获取
    cout << "曲面" << i << "的边界边集合： ";
    for (int j = 0; j < meshes[i].edge.size(); j++) {
        int v2 = edge[meshes[i].edge[j]].v2; //边的右端点
        for (int k = j + 1; k < meshes[i].edge.size(); k++) {
            int tv1 = edge[meshes[i].edge[k]].v1;
            int tv2 = edge[meshes[i].edge[k]].v2;
            if (tv1 == v2 or tv2 == v2) {
                int t = meshes[i].edge[j + 1];
                meshes[i].edge[j + 1] = meshes[i].edge[k];
                meshes[i].edge[k] = t;
                if (tv2 == v2) {
                    t = edge[meshes[i].edge[j + 1]].v1;
                    edge[meshes[i].edge[j + 1]].v1 = edge[meshes[i].edge[j + 1]].v2;
                    edge[meshes[i].edge[j + 1]].v2 = t;
                }
            }
        }
        int v1 = edge[meshes[i].edge[j]].v1; //边的左端点
        meshes[i].vertex.push_back(v1);
        cout << v1 << " " << v2 << " ";
    }
    cout << endl;
    cout << "曲面" << i << "的边界点集合： ";
    for (int j = 0; j < meshes[i].vertex.size(); j++) {
        cout << meshes[i].vertex[j] << " ";
    }
```

```
        cout << endl;
    }
}
{// 一条边的两端都是边界点的线，不一定是边界边
    cout << "边界边的数量：" << boundEdge.size() << endl;

    // 根据边界边，对边界点进行有序的排列，使其变成闭环
    /*
     * 假设现在的 vector<Edge> edges = {CD、CB、EA、BA、DE}
     * 则需要将其排序为{CD、DE、EA、AB、BC}，然后遍历边界边
     * 将每条边的 vIndex1 加入边界点边界点集合就构造完成了，为{C、D、E、A、B}
     */
    // 一个曲面内可能有多个闭合的边界边集合
    for (int i = 1; i <= numM; i++) {
        int len = meshes[i].edge.size() - 1;
        // 若首尾不相连，意味着该曲面有多个闭环
        int leftPtr = 0; // 左指针
        int rightPtr = 2; // ptr 指向最后意味着结束，闭环至少三条边，所以从 2 开始
        while (rightPtr <= len) {
            for (int j = rightPtr; j < meshes[i].edge.size(); j++) {
                if (edge[meshes[i].edge[j]].v2 == edge[meshes[i].edge[leftPtr]].v1) {
                    vector<int> temp;
                    for (int k = leftPtr; k <= j; k++) {
                        temp.push_back(edge[meshes[i].edge[k]].v1);
                    }
                    meshes[i].borderVerts.push_back(temp);
                    leftPtr = j + 1;
                    rightPtr = j + 3;
                    break;
                }
```

```
            }
          }
          cout << "曲面" << i << "边界点集合的个数：" << meshes[i].borderVerts.size() << endl;
      }
   }
}
int  GetPointNum(int i,int num) {

   // return 1st point

   if (i < 0) {

      return     0;

   }




   if (i < num)
      return i;
   // return last point

   return i+1-num;

}
void OnDraw(int num) {
   // clear the screen & depth buffer
   glClear(GL_COLOR_BUFFER_BIT);
   // clear the previous transform
   glLoadIdentity();
   // set the camera position
```

```
//    gluLookAt(     1,10,30,  //     eye pos
//              0,0,0,      //     aim point
//              0,1,0);     //     up direction
//    glColor3f(0.5,0.2,0);
   //
   // // draw curve hull
   glColor3f(0.3, 0, 0.5);
   /*glBegin(GL_LINE_STRIP);
   for (int i = 0; i != NUM_POINTS; ++i) {
      glVertex3fv(Points[i]);
   }
   glEnd();
   glColor3f(0, 1, 0);*/
   // begin drawing our curve
   glBegin(GL_LINE_STRIP);
   for (int k = 0; k < meshes[num].borderVerts.size(); k++) {
      vector<int> temp = meshes[num].borderVerts[k];
      cout << num << ":";
      for (int i = 0; i < temp.size(); i++) {
         cout << temp[i] << " ";
      }
      cout << endl;
      int size = temp.size();
      for (int start_cv = 0, j = 0; j < size; j++, start_cv++)
      {
         // for each section of curve, draw LOD number of divisions
         for (int i = 0; i != LOD; ++i) {
            // use the parametric time value 0 to 1 for this curve
            // segment.
            float t = (float)i / LOD;
            // the t value inverted
```

```
            float it = 1.0f - t;
            // calculate blending functions for cubic bspline
            float b0 = it * it * it / 6.0f;
            float b1 = (3 * t * t * t - 6 * t * t + 4) / 6.0f;
            float b2 = (-3 * t * t * t + 3 * t * t + 3 * t + 1) / 6.0f;
            float b3 = t * t * t / 6.0f;
            // calculate the x,y and z of the curve point
            float x = b0 * vertex[temp[GetPointNum(start_cv + 0, size)]].x+
               b1 * vertex[temp[GetPointNum(start_cv + 1, size)]].x +
               b2 * vertex[temp[GetPointNum(start_cv + 2, size)]].x +
               b3 * vertex[temp[GetPointNum(start_cv + 3, size)]].x;
            float y = b0 * vertex[temp[GetPointNum(start_cv + 0, size)]].y +
               b1 * vertex[temp[GetPointNum(start_cv + 1, size)]].y +
               b2 * vertex[temp[GetPointNum(start_cv + 2, size)]].y +
               b3 * vertex[temp[GetPointNum(start_cv + 3, size)]].y;
            float z = b0 * vertex[temp[GetPointNum(start_cv + 0, size)]].y +
               b1 * vertex[temp[GetPointNum(start_cv + 1, size)]].z +
               b2 * vertex[temp[GetPointNum(start_cv + 2, size)]].z +
               b3 * vertex[temp[GetPointNum(start_cv + 3, size)]].z;
            // specify the point
            glVertex3f(x, y,z);
         }
      }
   }
   glEnd();
   glFlush();

}



void statistics() {
```

```
    /*
     * 统计邻接三角面片
     * 统计邻接边
     * 统计边界点
     * 统计边界边
     */
     // 统计顶点的领接三角面片序号集合以及顶点的邻接边序号集合
    setBoundPoint();
    setBoundEdge();
}
void Uniknots(float* knots, int npts, int order)//节点向量
{
    int i, nknots;
    nknots = npts + order;
    for (i = 0; i < nknots; i++)knots[i] = i;
}
GLUnurbsObj* BCurve(float* knot, int nctls, float* ctls, int order)
{
    GLUnurbsObj* oBCurve;
    int nknots = nctls + order;
    oBCurve = gluNewNurbsRenderer();
    gluNurbsProperty(oBCurve, GLU_SAMPLING_TOLERANCE, 1);
    glEnable(GL_MAP1_VERTEX_3);
    gluBeginCurve(oBCurve);
    gluNurbsCurve(oBCurve, nknots, knot, 3, ctls, order, GL_MAP1_VERTEX_3);
    gluEndCurve(oBCurve);
    return oBCurve;
}
void showBcurve1() {
    float* knots=new float [boundPoint.size()];
```

```
    int n=boundPoint.size();
    Uniknots(knots, n, 4);
    float** p = new float* [boundPoint.size()];
    for (int i = 0; i < boundPoint.size(); i++) {
        p[i] = new float[3];
        p[i][0] = vertex[i].x; //把控制点数据传给变量
        p[i][1] = vertex[i].y;
        p[i][2] = vertex[i].z;
    }
    glColor3f(0.58, 0.58, 0.58);
    GLUnurbsObj* nobj;
    nobj = BCurve(knots, n, &p[0][0], 4);
}
//void showBcurve2() {
//    float** ctrlPoints;
//    for (const Meshes& tm : meshes)
//    {
//        for (const vector<int>& vertsGroup : tm.borderVerts)
//        {
//            int cpNum = vertsGroup.size();
//            for (int i = 0; i < cpNum; i++)
//            {
//                const Vertex& v = vertex[vertsGroup[i]];
//                ctrlPoints[i][0] = v.x;
//                ctrlPoints[i][1] = v.y;
//                ctrlPoints[i][2] = v.z;
//            }
//
//            for (int i = 0; i < POWER; i++, cpNum++)
//            {
//                const Vertex& v = vertex[vertsGroup[i]];
```

```
//            ctrlPoints[cpNum][0] = v.x;
//            ctrlPoints[cpNum][1] = v.y;
//            ctrlPoints[cpNum][2] = v.z;
//        }
//
//        int kNum = cpNum + POWER + 1;
//        GLfloat interval = 1.0 / (kNum - 1);
//        GLfloat ui = 0.0;
//        for (int i = 0; i < kNum; i++, ui += interval)
//        {
//            knots[i] = ui;
//        }
//
//        //绘制 B 样条曲线
//        GLUnurbsObj* theNurb;
//        gluBeginCurve(theNurb);
//        gluNurbsCurve(theNurb, kNum, knots, 3, &ctrlPoints[0][0], 4, GL_MAP1_VERTEX_3);
//        gluEndCurve(theNurb);
//    }
//
//  }
//}
void showBcurve3() {
  for (int i = 1; i <= numM; i++)
  {
    for (int k = 0; k < meshes[i].borderVerts.size(); ++k)
    {
      vector<int> vector1 = meshes[i].borderVerts[k];
      unsigned long cpNum = vector1.size();
      GLfloat ctrlPoints[1000][3];
      GLfloat knots[10000];
```

```
    for (int j = 0; j < cpNum; j++)
    {
        const Vertex& v = vertex[vector1[j]];
        ctrlPoints[j][0] = v.x;
        ctrlPoints[j][1] = v.y;
        ctrlPoints[j][2] = v.z;
    }

    for (int j = 0; j < 3; j++, cpNum++)
    {
        const Vertex& v = vertex[vector1[j]];
        ctrlPoints[cpNum][0] = v.x;
        ctrlPoints[cpNum][1] = v.y;
        ctrlPoints[cpNum][2] = v.z;
    }

    unsigned long kNum = cpNum + 3 + 1;
    for (int j = 0; j < kNum; j++)
    {
        knots[j] = j;
    }
    //绘制 B 样条曲线
    glColor3f(1, 0, 1);
    GLUnurbsObj* theNurb = gluNewNurbsRenderer();
    gluNurbsProperty(theNurb, GLU_SAMPLING_TOLERANCE, 1);
    glEnable(GL_MAP1_VERTEX_3);
    gluBeginCurve(theNurb);
    gluNurbsCurve(theNurb, kNum, knots, 3, &ctrlPoints[0][0], 4, GL_MAP1_VERTEX_3);
    gluEndCurve(theNurb);
    glFlush();
}
```

```
    }
}
void showBcurve4() {
    for (int i = 0; i < numM; i++) {
        OnDraw(i);
    }
}
void initWindow(int& argc, char* argv[], int width, int height)   //初始化并显示到屏幕中央
{
    glutInit(&argc, argv);  //初始化 GLUT
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowPosition(100, 100);
    glutInitWindowSize(width, height); //指定窗口大小
    glutCreateWindow("绘制三角网格模型");
    glClearColor(1, 1, 1, 0);
}

void Reshape(int w, int h)     //两个参数：窗口被移动后大小
{
    w = glutGet(GLUT_WINDOW_WIDTH);
    h = glutGet(GLUT_WINDOW_HEIGHT);
    Viewport(0, 0, w, h);
}
void display() {
    //drawTriangularMeshModel();
   /* cout << "----------绘制三角网格模型，完成！----------" << endl;
    showBoundPoint();
    cout << "----------显示边界点，完成！----------" << endl;
   showBoundEdge();
   cout << "----------显示边界边，完成！----------" << endl;
```

```
    showBcurve1();
    cout << "----------显示 B 样条 1，完成！----------" << endl;*/
    //showBcurve2();
    //cout << "----------显示 B 样条 2，完成！----------" << endl;
    showBcurve3();
    cout << "----------显示 B 样条 3，完成！----------" << endl;
    /*showBcurve4();
    cout << "----------显示 B 样条 4，完成！----------" << endl;*/
}
int main(int argc, char* argv[]) {
    loadXML();//加载 xml 数据
    statistics();//进行点、线、面的统计
    initWindow(argc, argv, 500, 500);
    glutDisplayFunc(display);
    glutReshapeFunc(Reshape);
    glutMainLoop();
    return 0;
}
```