班级： 软数1602

姓名： 李治兴

学号： 201692160

完成时间：2018年11月15日

计算几何与计算机图形学

算法报告

# 目录

调和映照之拓扑圆盘...................................................................................3

球体调和映照.............................................................................................12

同伦群基底.................................................................................................18

凸包映射.....................................................................................................28

额外算法.....................................................................................................34

硬件条件：所有算法时间都是在windows10系统下运行

编译器VS2018

硬件CPU i5-6300

# 拓扑圆盘调和映照

1. 算法思想

拓扑圆盘调和映照的算法思想在于，将一个非平面模型的点阵信息保角映射到一个平面圆盘上。然后慢慢的根据边缘将集中在中心的点慢慢展平，直到趋于稳定。

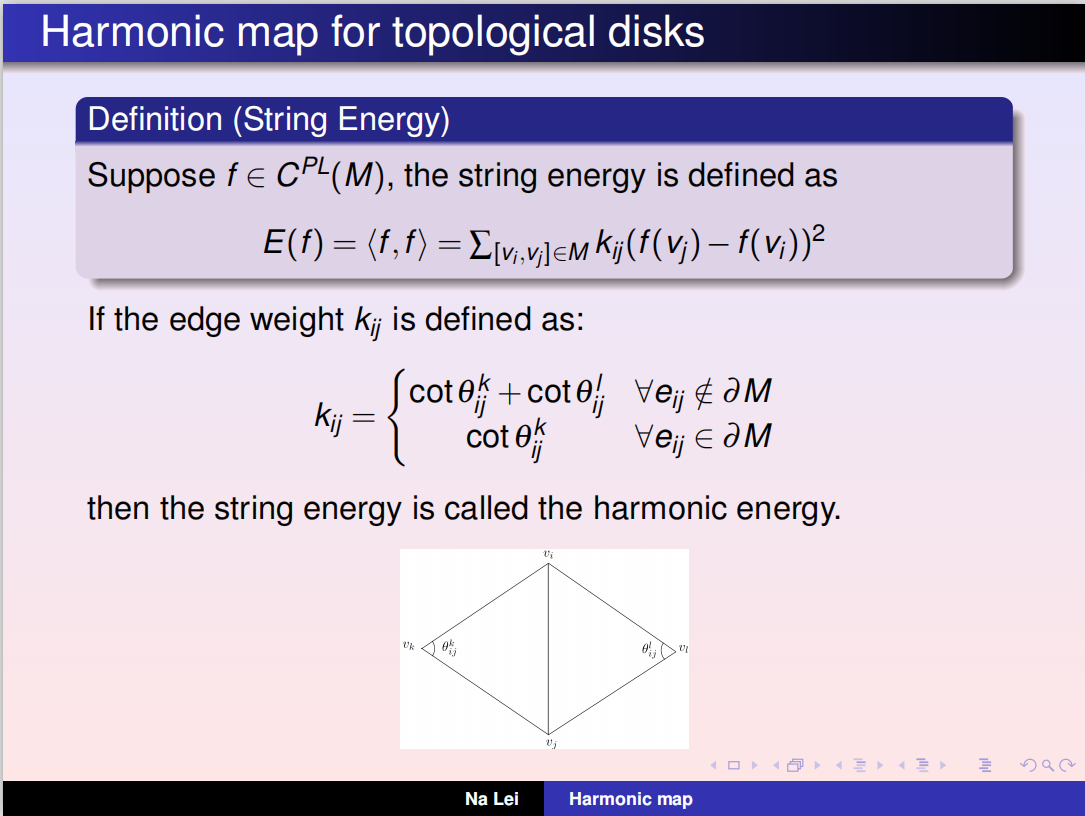
类似于将模型平面看做橡皮膜，将橡皮膜套在圆盘上，逐渐展平，让势能达到均衡。

2.算法流程

我们运用Meshlib库，以Shopie.m为例构建拓扑圆盘调和映照算法。

首先我们用边迭代器来遍历整个网格，然后算出每个边的边权重K，以及计算出每条边的长度，然后将所有在边界上的边加和，求出边界长度。然后我们用公式来计算整个模型现在的能量。

公式：



能量E=该边权重K\*该边长的平方；

然后我们将进行初始映射，先具体的做法是构建新迭代器，先找到一个边界上的半边，将这个半边的起始点映射至点（R,0），然后用顺时针优先找其余在边界上的半边，并用公式将所有边界上的这样就可以将整个边界遍历，并且将边界上的点都一一映射至圆盘上。

具体的映射公式：

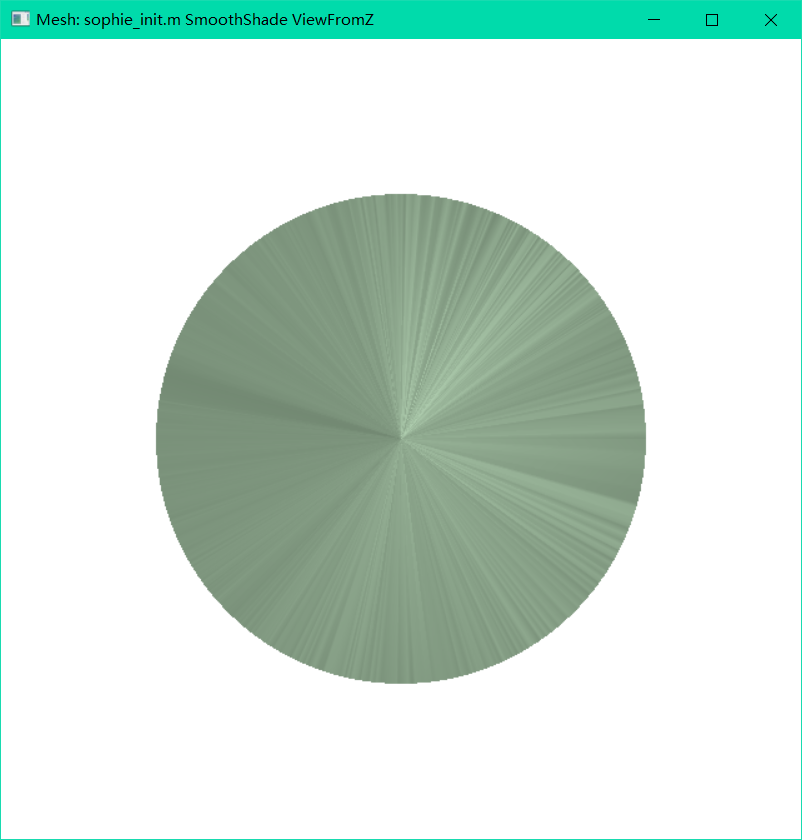
R = sum\_length1 / 2 / PI;

theit = new\_length / sum\_length1 \* PI \* 2;

NewP->changePoint(cos(theit)\*R, sin(theit)\*R, 0);

然后将所有不是边界上的点都重置为（0,0,0）。

输出一下经过初始映照之后的圆盘mesh，如下图。



之后将对初始映射之后的圆盘进行调和映照

每次迭代的步骤：先迭代所有边求出所有边的权重和sum\_k。之后，对所有不在边界上的顶点进行迭代，用半边迭代器VertexInHalfedgeIterator找出跟此顶点相关的所有半边，然后找出以此顶点为端点的边的另一端点，并将另一端点的坐标值乘上该边的权重。将所有乘积加和再除以所有边的权重和sum\_k，就得到了该顶点的新坐标。

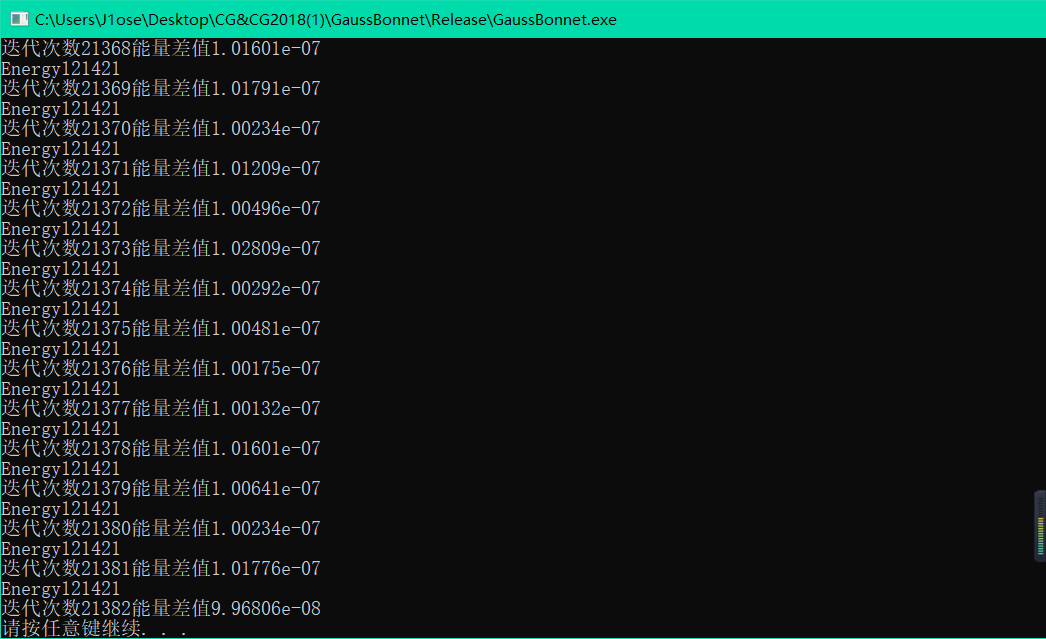
公式：

k\_sum += M\_Edge->edge()->m\_k;

New\_Pos += t\_p \* t\_k;

mviter.value()->point() = New\_Pos / k\_sum;

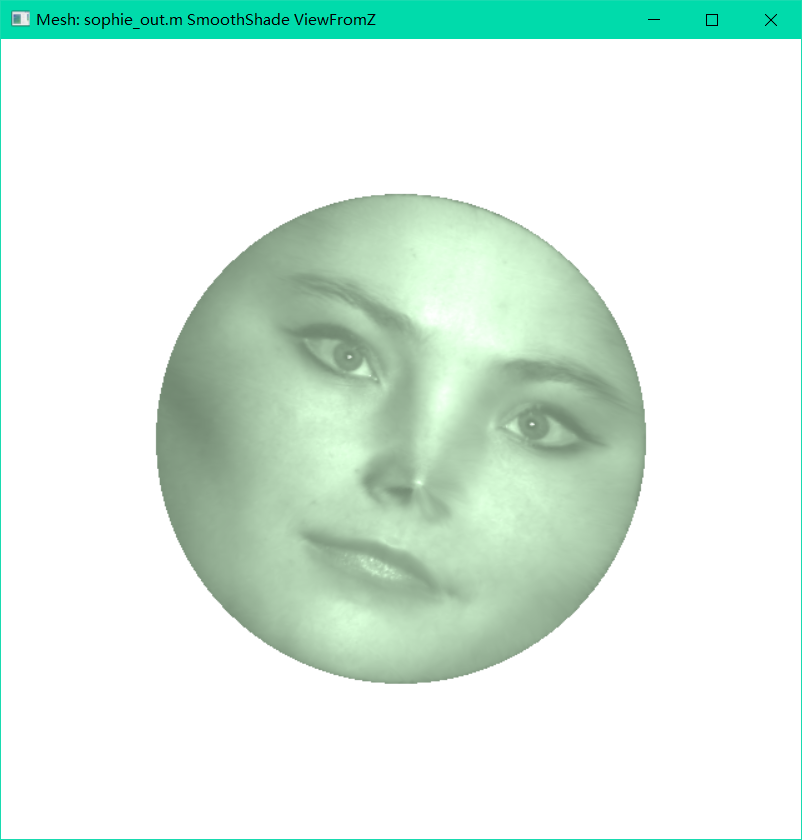
然我们计算改变之后整个模型的能量，并与之前算出的原模型的能量做差，当差值小于1e-7时，停止迭代。



导入模型sophie.m

总迭代次数21382次，时间6min,能量差值9.96806e-8

结果图片如下：



3.核心代码展示

void CGaussCurvature<M>::\_calculate\_K()

{

double sum = 0;

double E = 0;

for (M::MeshEdgeIterator me\_iter(m\_pMesh); !me\_iter.end(); ++me\_iter)

{

CEdge\* M\_Edge = me\_iter.value();

if (M\_Edge->boundary()) {

CPoint\* Vi= &M\_Edge->halfedge(0)->source()->point();

CPoint\* Vj = &M\_Edge->halfedge(0)->vertex()->point();

CPoint\* Vk = &M\_Edge->halfedge(0)->he\_prev()->source()->point();

//CPoint\* Vl = &M\_Edge->halfedge(1)->he\_prev()->source()->point();

double K = 0;

K = (((\*Vi) - (\*Vk))\*((\*Vj) - (\*Vk))) / (((\*Vi) - (\*Vk)) ^ ((\*Vj) - (\*Vk))).norm();

M\_Edge->m\_k = K;

M\_Edge->length = ((\*Vi) - (\*Vj)).norm();

cout <<"Boundary:"<<M\_Edge->m\_k<<"length"<< M\_Edge->length <<endl;

E += M\_Edge->m\_k\*M\_Edge->length\*M\_Edge->length;

}

if (!M\_Edge->boundary()) {

CPoint\* Vi = &M\_Edge->halfedge(0)->source()->point();

CPoint\* Vj = &M\_Edge->halfedge(0)->vertex()->point();

CPoint\* Vk = &M\_Edge->halfedge(0)->he\_prev()->source()->point();

CPoint\* Vl = &M\_Edge->halfedge(1)->he\_prev()->source()->point();

double K = 0;

K = (((\*Vi) - (\*Vk))\*((\*Vj) - (\*Vk))) / (((\*Vi) - (\*Vk)) ^ ((\*Vj) - (\*Vk))).norm()+ (((\*Vi) - (\*Vl))\*((\*Vj) - (\*Vl))) / (((\*Vi) - (\*Vl)) ^ ((\*Vj) - (\*Vl))).norm();

M\_Edge->m\_k = K;

M\_Edge->length = ((\*Vi) - (\*Vj)).norm();

cout << "Not Boundary:" << M\_Edge->m\_k << "length" << M\_Edge->length << endl;

E += M\_Edge->m\_k\*M\_Edge->length\*M\_Edge->length;

}

}

cout << "Energy:" << E << endl;

}

/\*

初始映射

\*/

template<typename M>

void CGaussCurvature<M>::init\_mapping()//初始映射

{

//calculate edge length

double sum\_length1 = 0;

double new\_length = 0;

CVertex\* begin\_vertex;

double theit = 0;

double R = 0;

for (M::MeshVertexIterator me\_iter(m\_pMesh); !me\_iter.end(); ++me\_iter)

{

if (me\_iter.value()->boundary()) {

begin\_vertex = me\_iter.value();

break;

}

}

for (M::MeshEdgeIterator me\_iter(m\_pMesh); !me\_iter.end(); ++me\_iter)

{

CEdge\* M\_Edge = me\_iter.value();

if (M\_Edge->boundary()) {

CPoint\* Vi = &M\_Edge->halfedge(0)->source()->point();

CPoint\* Vj = &M\_Edge->halfedge(0)->vertex()->point();

sum\_length1 += M\_Edge->length;

//CPoint\* Vl = &M\_Edge->halfedge(1)->he\_prev()->source()->point();

cout << "Length:" << M\_Edge->length << endl;

}

if (!M\_Edge->boundary()) {

CPoint\* Vi = &M\_Edge->halfedge(0)->source()->point();

CPoint\* Vj = &M\_Edge->halfedge(0)->vertex()->point();

Vi->changePoint(0, 0, 0);

Vj->changePoint(0, 0, 0);

cout << "Zero" << endl;

}

}

cout << "Sum\_Length:" << sum\_length1 << endl;

//init mapping

R = sum\_length1 / 2 / PI;

begin\_vertex->point() = CPoint(R, 0, 0);

new\_length += begin\_vertex->most\_clw\_out\_halfedge()->edge()->length;

for (CVertex \*cur\_vert= begin\_vertex->most\_clw\_out\_halfedge()->target();cur\_vert!=begin\_vertex; cur\_vert=cur\_vert->most\_clw\_out\_halfedge()->target())

{

if (cur\_vert->boundary()) {

CPoint\* NewP = &cur\_vert->point();

new\_length += cur\_vert->most\_clw\_out\_halfedge()->edge()->length;

theit = new\_length / sum\_length1 \* PI \* 2;

R = sum\_length1 / 2 / PI;

NewP->changePoint(cos(theit)\*R, sin(theit)\*R, 0);

cout << "Sucsess:"<<"X:"<<(\*NewP)(0)<<"Y:"<< (\*NewP)(1)<< endl;

}

if (!cur\_vert->boundary()) {

cout << "something wrong" << endl;

}

}

cout << new\_length << "and" << sum\_length1 << endl;

}

/\*

迭代调和映照

\*/

template<typename M>

void CGaussCurvature<M>::mapping()

{

double k\_sum=0;

double Ener\_value = 87838.2;

int count = 0;

while (1) {

double EE = 0;

for (M::MeshVertexIterator mviter(m\_pMesh); !mviter.end(); ++mviter)

{

//计算新坐标

if (!mviter.value()->boundary()) {

double k\_sum = 0;

for (M::VertexInHalfedgeIterator vi\_hiter(m\_pMesh, mviter.value()); !vi\_hiter.end(); ++vi\_hiter)

{

M::CHalfEdge\* M\_Edge = vi\_hiter.value();

k\_sum += M\_Edge->edge()->m\_k;

}

CPoint New\_Pos=CPoint(0,0,0);

for (M::VertexInHalfedgeIterator vi\_hiter(m\_pMesh, mviter.value()); !vi\_hiter.end(); ++vi\_hiter)

{

M::CHalfEdge\* M\_Edge = vi\_hiter.value();

double t\_k = M\_Edge->edge()->m\_k;

CPoint t\_p = M\_Edge->source()->point();

New\_Pos += t\_p \* t\_k;

}

mviter.value()->point() = New\_Pos / k\_sum;

//New\_Pos /= k\_sum;

//cout << "Sucsess:" << "X:" << New\_Pos(0) << "Y:" << New\_Pos(1) << "Z:" << New\_Pos(2) << endl;

//vertex->point().changePoint(New\_Pos(0), New\_Pos(1), New\_Pos(2));

//cout << "all is well" << k\_sum<<endl;

//CPoint\* Check = &vertex->point();

//cout << "Sucsess:" << "X:" << (\*Check)[0] << "Y:" << (\*Check)[1] << "Z:" << (\*Check)[2] <<endl;

}

}

//计算迭代之后的能量

for (M::MeshEdgeIterator me\_iter(m\_pMesh); !me\_iter.end(); ++me\_iter)

{

CPoint c1, c2;

c1 = me\_iter.value()->halfedge(0)->source()->point();

c2 = me\_iter.value()->halfedge(0)->target()->point();

me\_iter.value()->length() = (c1 - c2).norm();

EE += me\_iter.value()->m\_k \* me\_iter.value()->length() \* me\_iter.value()->length();

}

//Ener\_value -= EE;

cout << "Energy" << EE << endl;

count++;

cout << "迭代次数" << count << "能量差值" << fabs(Ener\_value - EE) <<endl;

if (fabs(Ener\_value - EE) < 1e-7) {

break;

}

Ener\_value = EE;

if (count == 5000) {

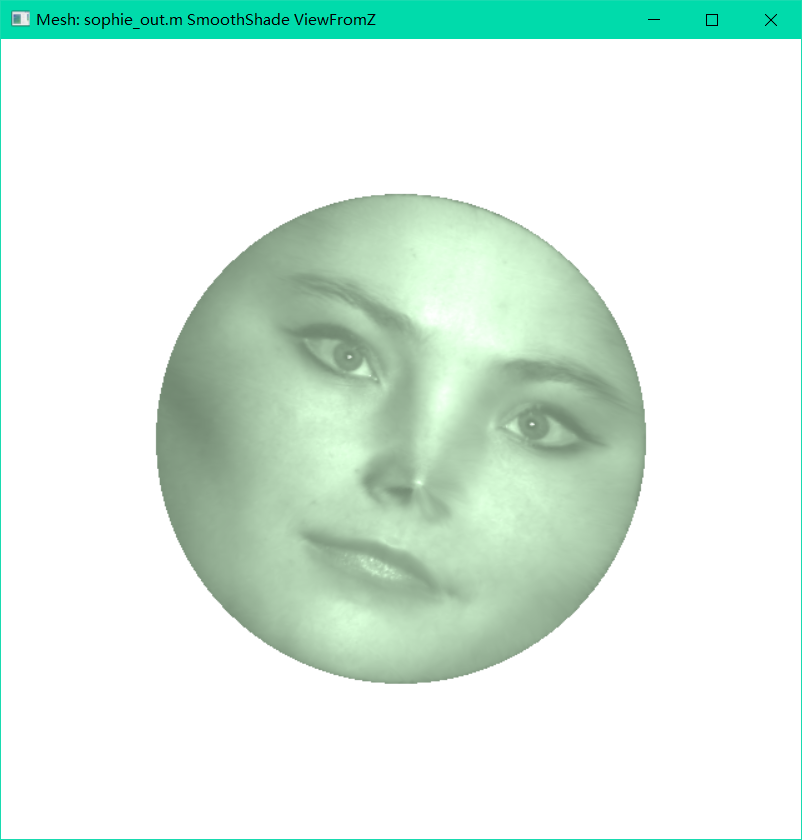
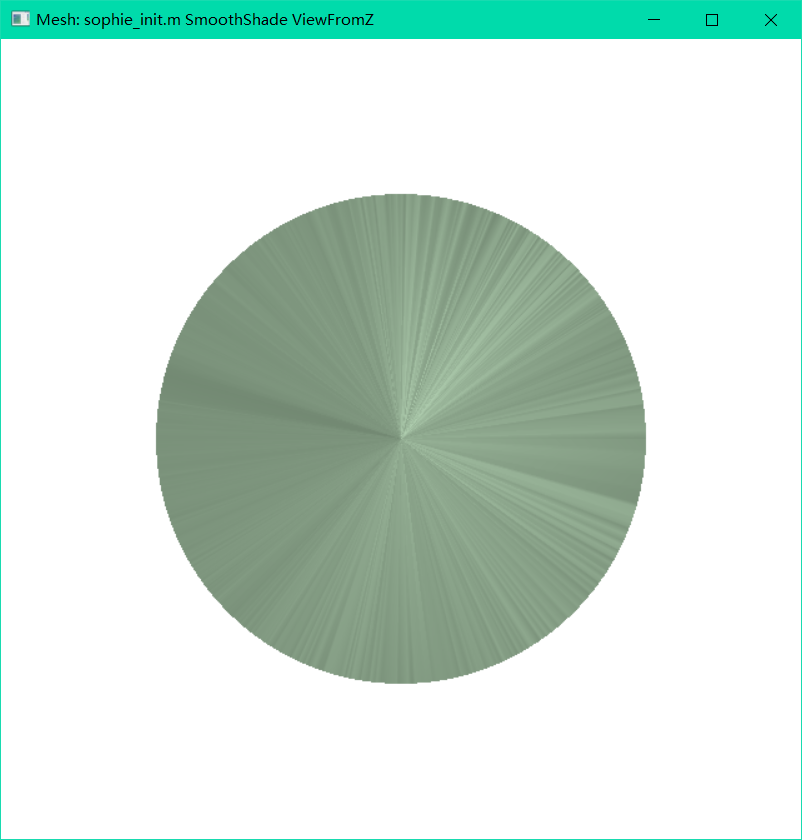
//break;

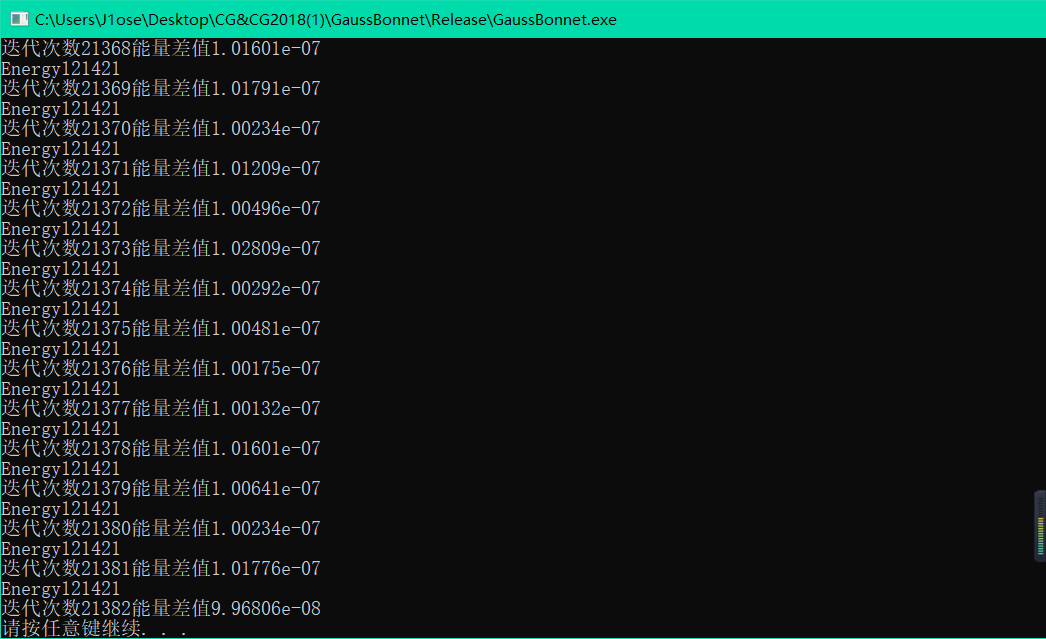
}

}

}

4.结果展示：





运行时间:6min23s.

分析总结：

拓扑圆盘映照是我对于meshlib库运用的入门，是在做这个的过程中我对meshlib类库进行了系统的阅读，也对类库中的数据结构和工具有了初步的了解，但是问题在于，我的所有算法都是在老师给的高斯-波涅定理的够工程中写的，有很多命名空间没有读懂，函数太过于模块化，期间很多指针的运用也不太正规，从这个算法之后，我向同学学习了一些代码算法写法，然后加以改进。整体性才更强了些。

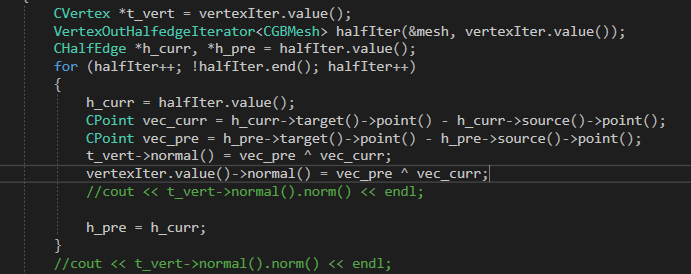
# 二、球体调和映照

1.算法思想：

球体调和映照的大体思想和拓扑圆盘思想一致，就是将原模型的点单位化映射到单位圆上，之后慢慢地调和，使能量慢慢达到平衡。不同之处可能在于初始映射至球面和圆盘的算法不同。

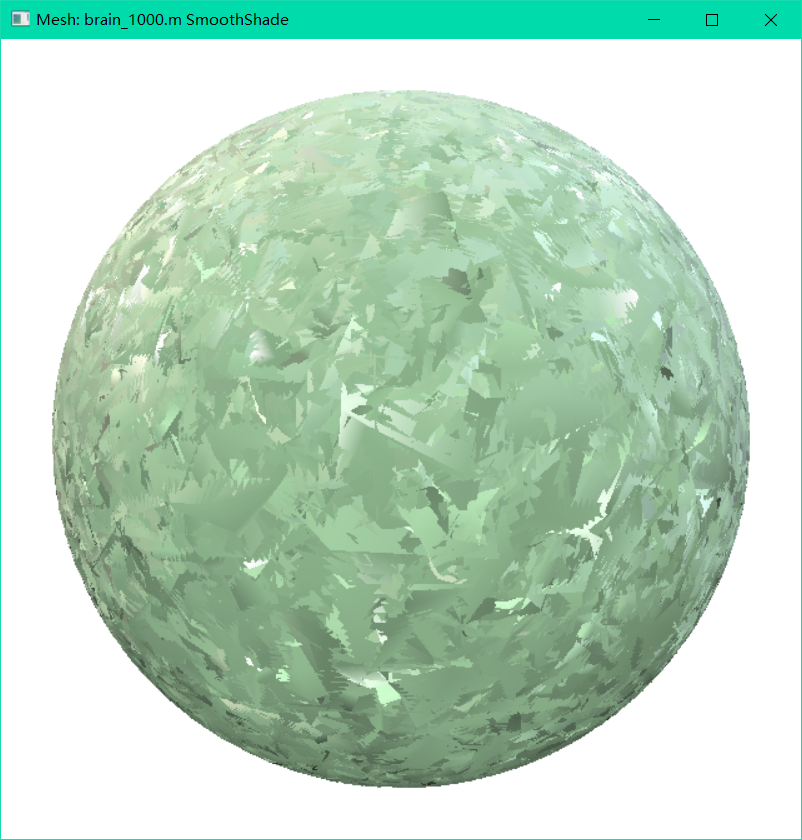
1. 算法流程

开始跟拓扑圆盘一样，先求每条边的权重K，然后要求每个顶点的初始法向量，具体求法代码如下



大概的算法就是算以该顶点为起始点的所有边相邻之间的叉乘和，得到的就是该顶点的法向量。

然后将所有顶点的法向量除以法向量的长度，将所有法向量单位化，这样所有的顶点就都会在一个单位球面上。此时的单位球面如图所示，是因为大脑的模型有很多勾回，很多面交织重叠



然后就开始了迭代过程，迭代的过程开始还是要算能量，然后要用另一种方式算向球心方向的向量，算法为以一个顶点为起点的向量和，得出的向量为V，然后用公式通过这两个向量算出该点在球面上的切向量

迭代的方式是规定一个a，我规定的是a=0.01，然后将切向量单位化，然后用公式

P=P+aP来使球面上的点向切向量方向位移一段距离。

然后开始计算质心，质心的计算方法为所有法向量和除n，然后算出的法向量就为新的法向量。

然后用新的法向量重复上述步骤，使模型在球面上慢慢展开，然后计算新的能量和原能量的能量差，当能量差小于1e-7时，迭代结束。

1. 核心代码展示

std::cout << mesh.numVertices() << " " << mesh.numEdges() << " " << mesh.numFaces() << std::endl;

std::cout << mesh.vertices().size() << std::endl;

//求K值

int b\_edge\_sum = 0;

for (MeshEdgeIterator<CGBMesh> edgeIter(&mesh); !edgeIter.end(); edgeIter++)

{

CHalfEdge \*halfEdge\_0 = edgeIter.value()->halfedge(0);

CHalfEdge \*halfEdge\_1 = edgeIter.value()->halfedge(1);

CPoint vi, vj, vk, vl;

vi = halfEdge\_0->source()->point();

vj = halfEdge\_0->target()->point();

vk = halfEdge\_0->he\_next()->target()->point();

vl = halfEdge\_1->he\_next()->target()->point();

CPoint px = vi - vk;

CPoint py = vj - vk;

CPoint pm = vi - vl;

CPoint pn = vj - vl;

double k1 = (px \* py) / (px^py).norm();

double k2 = (pm \* pn) / (pm^pn).norm();

edgeIter.value()->m\_k = k1 + k2;

edgeIter.value()->m\_k = k1 + k2;

//cout << edgeIter.value()->m\_k << std::endl;

}

//构造初始映射

//遍历所有点，求向量x乘和

for (MeshVertexIterator<CGBMesh> vertexIter(&mesh); !vertexIter.end(); vertexIter++)

{

CVertex \*t\_vert = vertexIter.value();

VertexOutHalfedgeIterator<CGBMesh> halfIter(&mesh, vertexIter.value());

CHalfEdge \*h\_curr, \*h\_pre = halfIter.value();

for (halfIter++; !halfIter.end(); halfIter++)

{

h\_curr = halfIter.value();

CPoint vec\_curr = h\_curr->target()->point() - h\_curr->source()->point();

CPoint vec\_pre = h\_pre->target()->point() - h\_pre->source()->point();

t\_vert->normal() = vec\_pre ^ vec\_curr;

vertexIter.value()->normal() = vec\_pre ^ vec\_curr;

//cout << t\_vert->normal().norm() << endl;

h\_pre = h\_curr;

}

//cout << t\_vert->normal().norm() << endl;

}

//将所有点映射到单位球

for (MeshVertexIterator<CGBMesh> vertexIter(&mesh); !vertexIter.end(); vertexIter++) {

//cout << vertexIter.value()->normal().norm() << endl;

vertexIter.value()->point() = vertexIter.value()->normal() / vertexIter.value()->normal().norm();

//标准化

}

mesh.write\_m("START.m");

//迭代

double k1 = 0;

for (MeshEdgeIterator<CGBMesh> edgeIter(&mesh); !edgeIter.end(); edgeIter++)

{

CPoint c1, c2;

c1 = edgeIter.value()->halfedge(0)->source()->point();

c2 = edgeIter.value()->halfedge(0)->target()->point();

edgeIter.value()->length() = (c1 - c2).norm();

//验证

// cout << edgeIter.value()->length()<< edgeIter.value()->m\_k << endl;

k1 += edgeIter.value()->m\_k \* edgeIter.value()->length() \* edgeIter.value()->length();

//cout << k1<<endl;

}

for (int i = 0; true; i++)

{

//遍历所有点，向切向量方向位移一小段

for (MeshVertexIterator<CGBMesh> vertexIter(&mesh); !vertexIter.end(); vertexIter++)

{

CVertex \*t\_vert = vertexIter.value();

CHalfEdge \*h\_curr;

CPoint v\_inter; //半边向量和

for (VertexOutHalfedgeIterator<CGBMesh> halfIter(&mesh, vertexIter.value()); !halfIter.end(); halfIter++)

{

h\_curr = halfIter.value();

CPoint t\_v = h\_curr->target()->point() - h\_curr->source()->point();

v\_inter += t\_v;

}

CPoint v\_tangent = v\_inter - vertexIter.value()->normal() \* (v\_inter \* vertexIter.value()->normal()); //切向量

vertexIter.value()->point() += v\_tangent \* a;

vertexIter.value()->point() /= vertexIter.value()->point().norm(); //规格化

}

//规格化法向量

CPoint p\_center; //质心

for (MeshVertexIterator<CGBMesh> vertexIter(&mesh); !vertexIter.end(); vertexIter++)

p\_center += vertexIter.value()->point();

p\_center /= mesh.numVertices();

for (MeshVertexIterator<CGBMesh> vertexIter(&mesh); !vertexIter.end(); vertexIter++)

{

CPoint v\_norm = vertexIter.value()->point() - p\_center;

vertexIter.value()->normal() = v\_norm / v\_norm.norm();

vertexIter.value()->point() = v\_norm / v\_norm.norm();

}

std::cout << "第 " << i << " 次迭代" << std::endl;

double k2 = 0;

for (MeshEdgeIterator<CGBMesh> edgeIter(&mesh); !edgeIter.end(); edgeIter++)

{

CPoint c1, c2;

c1 = edgeIter.value()->halfedge(0)->source()->point();

c2 = edgeIter.value()->halfedge(0)->target()->point();

edgeIter.value()->length() = (c1 - c2).norm();

k2 += edgeIter.value()->m\_k \* edgeIter.value()->length() \* edgeIter.value()->length();

}

//std::cout << "k1=" << k1 << std::endl;

//std::cout << "k2=" << k2 << std::endl;

cout << k1 << k2 << endl;

std::cout << "E=" << fabs(k1 - k2) << std::endl;

if (fabs(k1 - k2) < 1e-7)

break;

k1 = k2;

}

//输出

mesh.write\_m(outName);

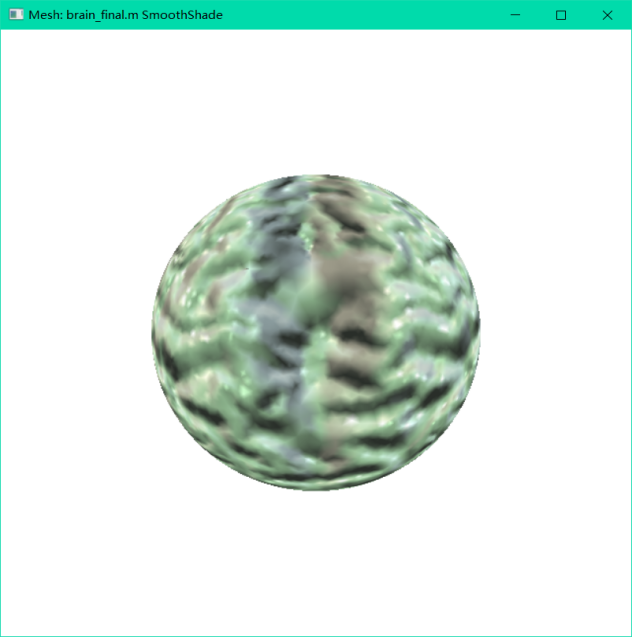
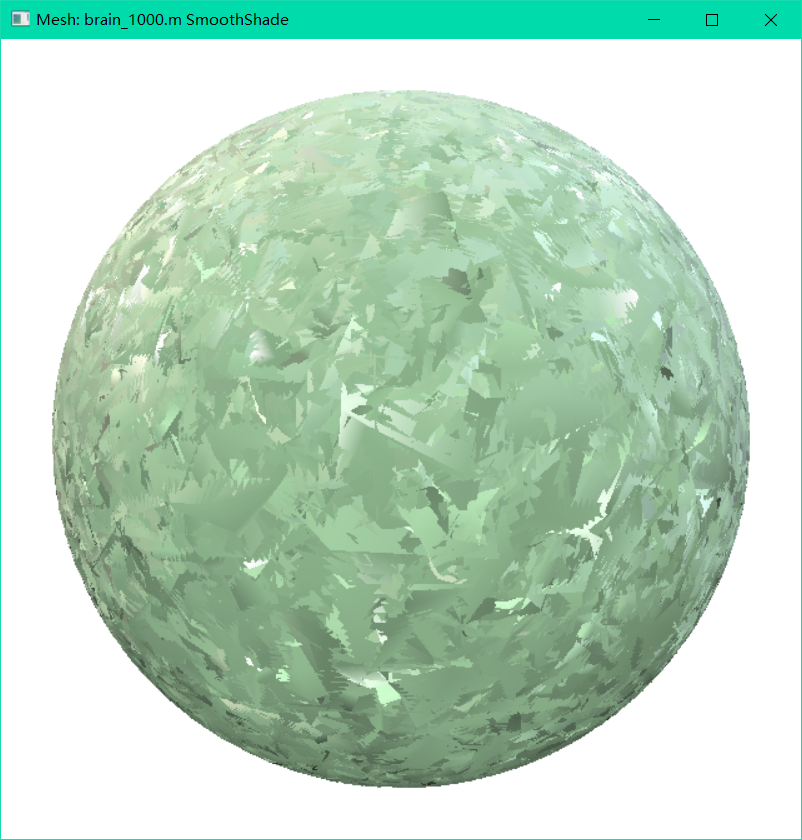
std::cout << "输出.m" << std::endl;

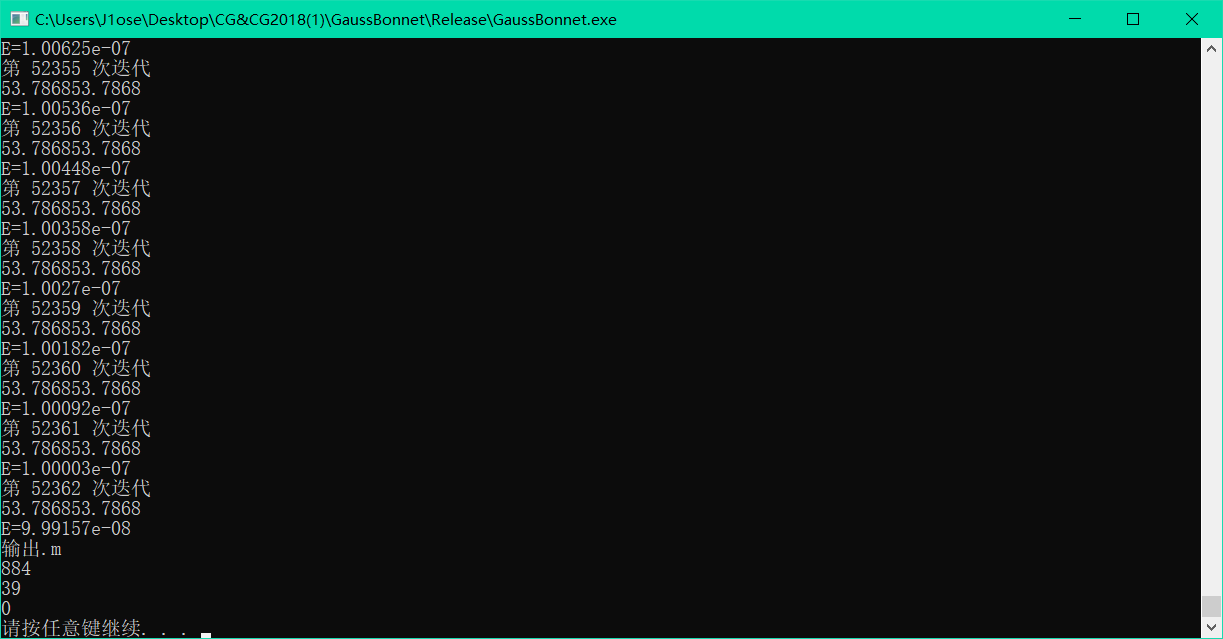
time(&end);

cout << difftime(end, start) << endl;

}

1. 结果展示





输入brain.m迭代次数52362次windows统计时间14min50s

分析总结：这次跟同学探讨改变了写法，把整个算法都整合到了一个函数里，通过参数输入输出文件，加上注释可读性还可以，指针的问题还时有出现，不过基本解决，这个算法让我对meshlib有了更加深刻的了解。

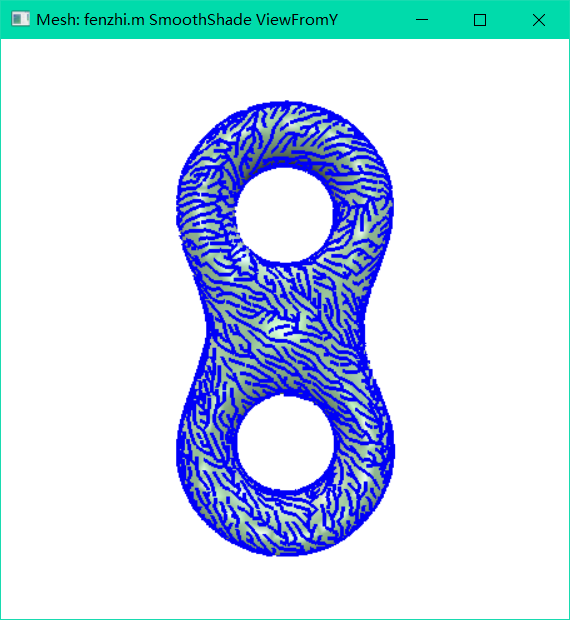
三、同伦群基底

1. 算法思想，同伦群基底所求出的边界，就是该模型沿着这条边界裁剪开来，必能得到一个平面。

这个算法的理论依据是火烧法，想象在模型的任意一面放火，火势蔓延只会蔓延到相邻的地方，所以保证火烧过的地方必定属于一个平面。

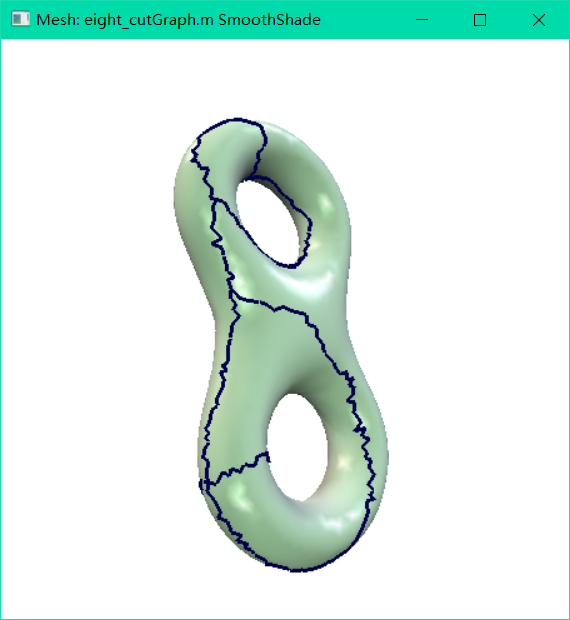
那么最后模型的所有面经过火烧之后，剩下的边界就是一个平面的边界，所以沿着该平面剪开必定能得到一个平面。

1. 算法步骤，给面添加三个状态标识IN\_FIRE,ON\_FIRE,OFF\_FIRE，先选择一个起始面作为火烧法的起点，将其标识改变为ON\_FIRE，之后开始BFS广度优先遍历，遍历过程中，遍历到新节点，则原节点由火边缘变为火内，新边缘变为火边缘，如果两个半边都被烧过或正在燃烧，则将此条边高亮。然后直到火烧过所有面，跳出，查看火烧后的模型结果。



可以看到有很多的毛边，这些毛边是因为火烧的过程中很有可能先行遍历了一些面，让某些边的两侧面从两个方向被火烧过，使得这些边一直处于ON\_FIRE的状态，而这些边是可以整合成多条特定边的，接下来我们就来去掉这些分支。

具体的算法是，遍历所有顶点，如果该顶点周围只有一条火边缘，则去掉该条火边缘。原因是，火边缘是最后裁开该模型的边界，可以知道裁开的边界应该是一个闭合路径，所以在边界上的所有点周围至少都有两条火边缘上的边。按照这个方法去分支后：



之后我们会建立二叉树分别储存cut-graph。具体的操作是选择同伦基底上的一个顶点，以此为根节点任选一个方向开始遍历，如果遇到有顶点有3个或3个以上的同伦基底边界，则变为子节点然后展开两个查找分支，当发现没有未遍历的下一条边时，将该节点记做叶子节点，自此二叉树形成，从各个叶子节点向回找到最近的一个父节点闭合，则为一个同伦基底，从每一个叶子节点往回找，直到遇到父节点，之后单独储存，输出。

1. 核心代码

//同伦群基底

void homotopy(char \*fileName, char \*outName)

{

time\_t start, end;

time(&start);

CGBMesh mesh;

mesh.read\_m(fileName);

std::cout << mesh.numVertices() << " " << mesh.numEdges() << " " << mesh.numFaces() << std::endl;

std::vector<CFace\*> f\_face;

//起始面

MeshFaceIterator<CGBMesh> it(&mesh);

it.value()->state() = ON\_FIRE;

//it.value()->string() = "sharp = true";

f\_face.push\_back(it.value());

//BFS

for (int m = 0; true; m++)

{

//遍历所有火内的面

int t\_size = f\_face.size();

std::cout << t\_size << endl;

if (t\_size >= mesh.numFaces())

{

break;

}

cout << mesh.numFaces() << endl;

for (int i = 0; i < t\_size; i++) //如果在火边缘

{

CFace \*t\_face = f\_face[i];

if (t\_face->state() == ON\_FIRE)

{

cout << "GOGO" << endl;

t\_face->state() = IN\_FIRE; //改变face状态至火内

//每个Face三个halfEdge， 一圈， 三个halfEdge与一个Face相对应

for (FaceHalfedgeIterator<CGBMesh> it((MeshLib::CGBMesh::CFace\*)t\_face); !it.end(); it++)

{

CHalfEdge \*t\_h1 = it.value();

CHalfEdge \*t\_h2 = t\_h1->edge()->other(t\_h1); //半边相对的另一条半边

CFace \*t\_f = t\_h2->face();

if (t\_f->state() == OUT\_FIRE)

{

t\_f->state() = ON\_FIRE;

f\_face.push\_back(t\_f);

//判断是否烧到了另一边

for (FaceHalfedgeIterator<CGBMesh> itt((MeshLib::CGBMesh::CFace\*)t\_f); !itt.end(); itt++)

{

CHalfEdge \*o\_half = itt.value()->edge()->other(itt.value());

if (o\_half->face()->state() != OUT\_FIRE && o\_half->face() != t\_face)

{

o\_half->edge()->string() = "sharp = true";

}

}

}

}

}

}

}

std::cout << "Finish iter" << endl;

mesh.write\_m("fenzhi.m");

//去分支

while (true)

{

bool flag = false;

for (MeshVertexIterator<CGBMesh> it(&mesh); !it.end(); it++)

{

CVertex \*t\_vert = it.value();

int e\_sum = 0;

CEdge \*r\_edge;

for (VertexOutHalfedgeIterator<CGBMesh> halfIt(&mesh, it.value()); !halfIt.end(); halfIt++)

{

CEdge \*t\_e = halfIt.value()->edge();

if (t\_e->string() == "sharp = true")

{

r\_edge = t\_e;

e\_sum++;

}

}

if (e\_sum == 1) //顶点周围只有一条sharp边，去掉

{

flag = true;

r\_edge->string() = "";

}

}

if (flag == false)

break;

}

mesh.write\_m(outName);

//////////////////////////////////////////////////////////////////////////////////////////////////

//创建二叉树储存Cut-Graph

CEdge \*root\_edge = NULL; //根边

for (MeshEdgeIterator<CGBMesh> edgeIt(&mesh); !edgeIt.end(); edgeIt++)

{

if (edgeIt.value()->string() == "sharp = true")

{

root\_edge = edgeIt.value();

//root\_edge->string() = "";

break;

}

}

root\_edge->isVisited() = true;

CVertex \*root\_vert = root\_edge->halfedge(0)->source(); //根顶点

root\_vert->isVisited() = true;

root\_vert->isNode() = true;

vector<CEdge\*> res\_edges; //叶子节点之间的余下的边

//DFS

stack<CEdge\*> e\_stack;

e\_stack.push(root\_edge);

while (!e\_stack.empty())

{

CEdge \*cur\_edge = e\_stack.top(); //当前边

cur\_edge->isVisited() = true;

e\_stack.pop();

CVertex \*cur\_vert = NULL; //当前边的out点

if (cur\_edge->halfedge(0)->target()->isVisited())

{

cur\_vert = cur\_edge->halfedge(0)->source();

}

else

{

cur\_vert = cur\_edge->halfedge(0)->target();

}

cur\_vert->isVisited() = true;

//cur\_vert->string() = "sharp = true";

vector<CEdge\*> next\_edges; //储存下一边

for (VertexOutHalfedgeIterator<CGBMesh> halfIt(&mesh, (CGaussBonnetVertex\*)cur\_vert); !halfIt.end(); halfIt++)

{

CHalfEdge \*next\_half = halfIt.value();

if (next\_half->edge()->isVisited() == false && next\_half->edge()->string() == "sharp = true") //未被访问的G山的吓一跳边

{

next\_edges.push\_back(next\_half->edge());

}

}

if (next\_edges.size() == 0) //没有下一条边，叶子节点

{

res\_edges.push\_back(cur\_edge);

}

if (next\_edges.size() >= 2) //分支节点

{

cur\_vert->isNode() = true;

}

for (int i = 0; i < next\_edges.size(); i++)

{

next\_edges[i]->preEdge() = cur\_edge; //赋值”前“边

e\_stack.push(next\_edges[i]);

}

//////////////////////

//mesh.write\_m(outName);

}

//删除res\_edges中重复的边

vector<CEdge\*> result\_edges;

for (int i = 0; i < res\_edges.size(); i++)

{

bool flag = false;

for (int j = 0; j < result\_edges.size(); j++)

{

if (result\_edges[j] == res\_edges[i])

flag = true;

}

if (flag == false)

result\_edges.push\_back(res\_edges[i]);

}

//从叶子节点往回找 求4个基底

vector<vector<CEdge\*>> bases; //储存所有基底

for (int i = 0; i < result\_edges.size(); i++)

{

CEdge\* r\_edge = result\_edges[i]; //叶子边

CVertex \*r\_v1 = r\_edge->halfedge(0)->target();

CVertex \*r\_v2 = r\_edge->halfedge(0)->source();

CVertex \*beg\_v = NULL; //叶子顶点

if (r\_v1->isNode() == true)

beg\_v = r\_v2;

else

beg\_v = r\_v1;

vector<CEdge\*> b\_edges; //储存一个基底

b\_edges.push\_back(r\_edge);

CEdge \*cur\_edge = r\_edge;

CVertex \*cur\_vt = beg\_v;

CVertex \*pre\_vt = NULL;

while (true)

{

cur\_edge = cur\_edge->preEdge(); //往回找

b\_edges.push\_back(cur\_edge);

if (cur\_edge->halfedge(0)->target() == cur\_vt)

pre\_vt = cur\_edge->halfedge(0)->source();

else

pre\_vt = cur\_edge->halfedge(0)->target();

cur\_vt = pre\_vt;

if (pre\_vt == r\_v1 || pre\_vt == r\_v2)

{

break;

}

}

bases.push\_back(b\_edges);

}

//输出四个基底

for (int i = 0; i < bases.size(); i++)

{

//清理mesh

for (MeshEdgeIterator<CGBMesh> it(&mesh); !it.end(); it++)

it.value()->string() = "";

//高亮一个同伦群基底

vector<CEdge\*> t\_base = bases[i];

for (int j = 0; j < t\_base.size(); j++)

{

t\_base[j]->string() = "sharp = true";

}

//写

string s = "BASE\_";

char c = i + 49;

s = s + c;

s = s + '.';

s = s + 'm';

mesh.write\_m(s.c\_str());

}

time(&end);

cout << difftime(end, start) << endl;

////高亮 分支节点 res边

//for (MeshVertexIterator<CGBMesh> i(&mesh); !i.end(); i++)

//{

// if (i.value()->isNode() == true)

// i.value()->string() = "sharp = true";

//}

//for (int i = 0; i < res\_edges.size(); i++)

//{

// res\_edges[i]->string() = "";

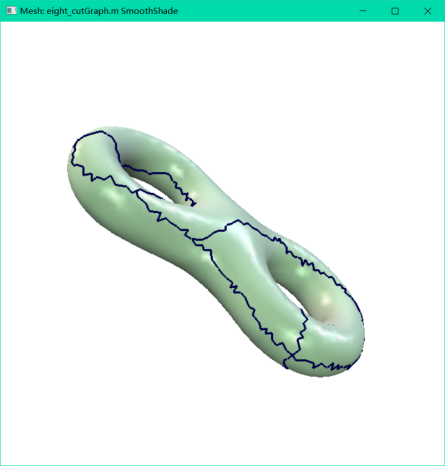
//}

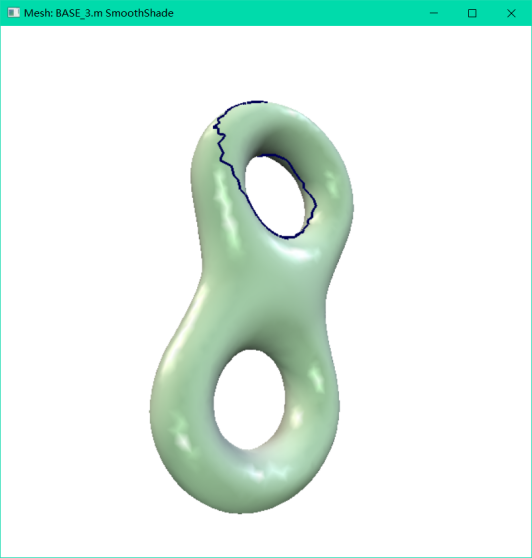
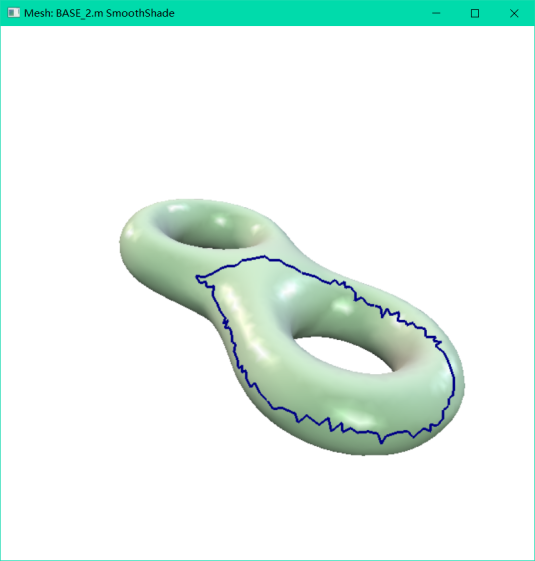
//mesh.write\_m(outName);

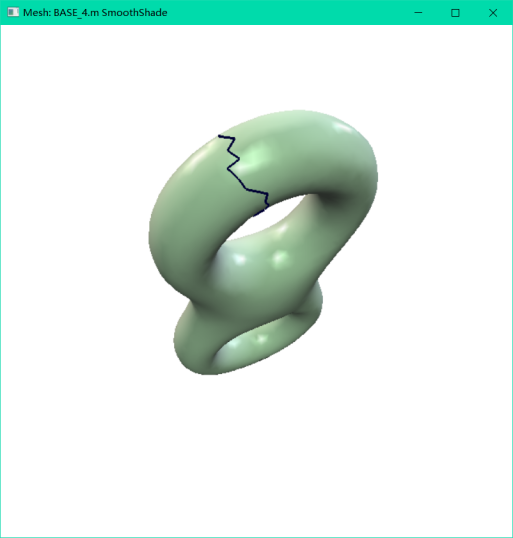
}

4.结果展示(输入eight.m,输出eight\_cutGraph.m BASE\_1.m

BASE\_2.m BASE\_3.m BASE\_4.m)





用时1min

5.分析总结：同伦群是很难的一个算法，开始在课上关于同伦群的概念就比较难理解，不过火烧法还是很形象的，然后树的遍历算法也不简单，火烧法可能比较容易实现，但是去分支，之后遍历分开不同的基底也非常有难度，不过也掌握了很多拓扑知识。

四、凸包映射

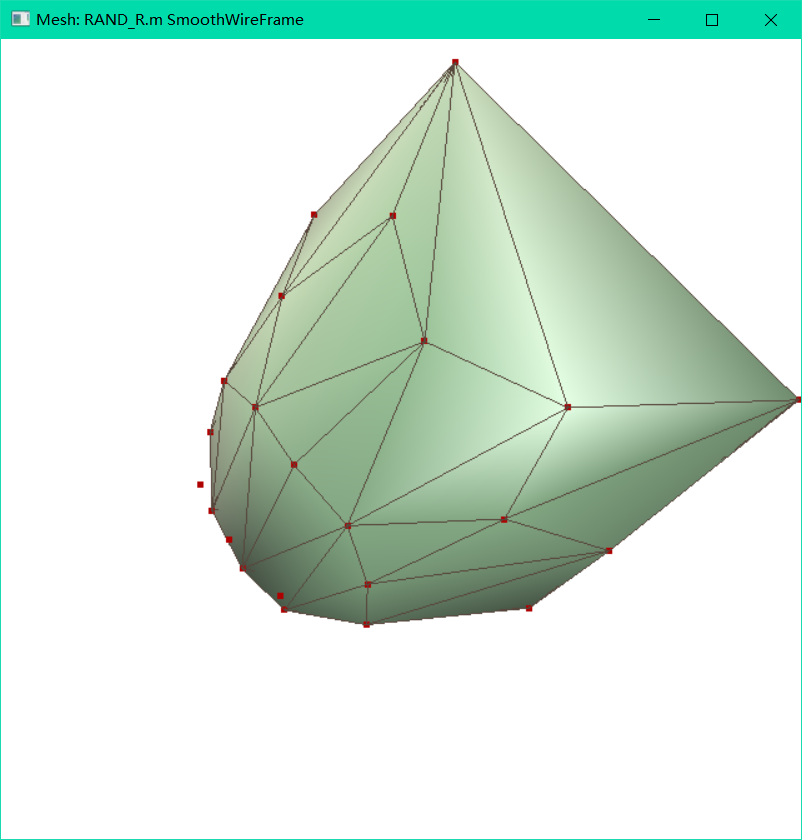
1. 算法思想

凸包：在二维欧几里得空间中，凸包可以想象为一条刚好包着所有点的橡皮圈，用不严谨的话来讲，给定二维平面上的点集，凸包就是将最外层的点连接起来构成的凸多边形，它能包含点集中所有的点。

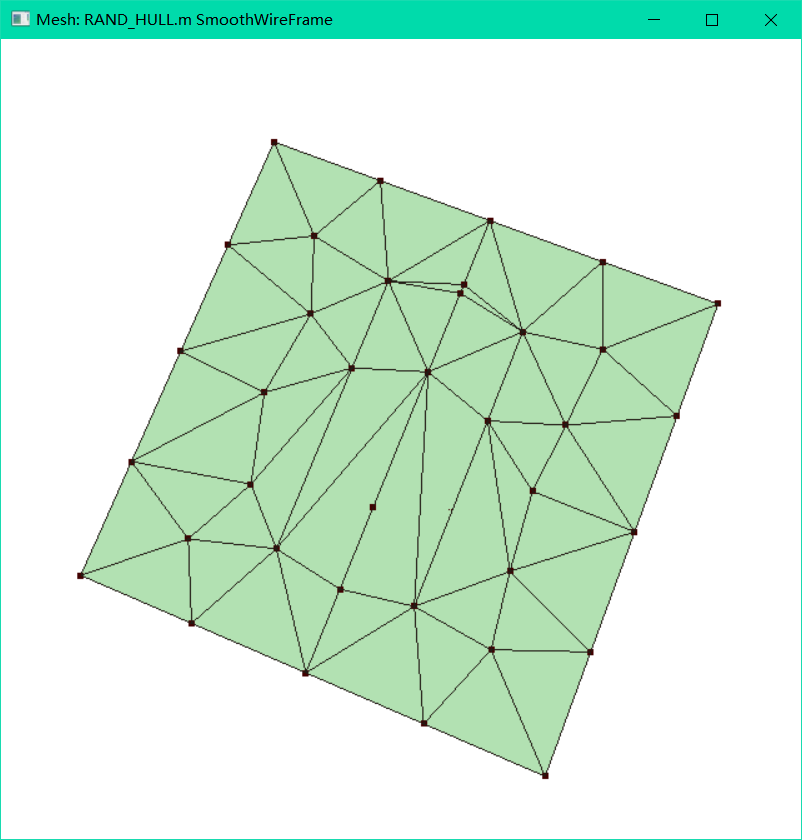
我们将会用随机点构建凸包，然后经过操作后的网格符合德洛内三角网格。

1. 算法操作

首先我们随机生成在xyz坐标都在[-1,1]之间的多个随机点，之后按照.m文件的结构来储存为一个.m文件，之后我们参照meshlib库里的读入，写入文件的函数，以及修改创建面和删除面的函数写一个针对我们文件的函数操作。在操作中为了不让凸包建立的时候面因为法向量错误没法显示，我们不采用半边结构。然后我们将对读入的mesh里的点进行映射，将这些点的z轴坐标全部映射为x2+y2。然后，我们将会在这些点中选择初始的四个点，以这四个点为基础构造一个四面体，也就是说要生成四个面，在创建面的时候要注意输入点的顺序，这样才能保证创建的边法向量正确，能够正确显示。然后我们将选择一个四面体顶点之外的点，用行列式来算出他和四面体顶点任意三个顶点组成的四面体的体积，若为负则要删除掉该面，然后跟该顶点形成新的四面体。然后重复这个方法直到所有的点都被遍历完，这个时候生成的凸包如图



将所有凸包映射回平面，就完成了德洛内三角网格的生成，如图。



1. 核心代码展示

void hull(char \*fileName, char \*outName)

{

time\_t start, end;

time(&start);

CGBMesh mesh;

mesh.read\_vertices\_m(fileName);

cout << mesh.numVertices() << endl;

//提升到3D

for (list<CGaussBonnetVertex\*>::iterator it = mesh.vertices().begin(); it != mesh.vertices().end(); it++)

{

CVertex \*t\_vertex = \*it;

t\_vertex->string() = "sharp = true";

CPoint t\_pot = t\_vertex->point();

t\_pot = CPoint(t\_pot[0], t\_pot[1], t\_pot[0] \* t\_pot[0] + t\_pot[1] \* t\_pot[1]); //(x, y, x^2+y^2)

t\_vertex->point() = t\_pot;

}

//mesh.write\_m("meshes/VVVVVV.m");

//初始四个点

CGaussBonnetVertex \*ver1 = mesh.idVertex(31);

//ver1->string() = "sharp = true";

CGaussBonnetVertex \*ver2 = mesh.idVertex(0);

//ver2->string() = "sharp = true";

CGaussBonnetVertex \*ver3 = mesh.idVertex(1);

//ver3->string() = "sharp = true";

CGaussBonnetVertex \*ver4 = mesh.idVertex(9);

//ver4->string() = "sharp = true";

//创建四面体

int faceId = 0;

CGaussBonnetVertex \*face1[3] = { ver1, ver2, ver3 }; //注意点顺序

CGaussBonnetVertex \*face2[3] = { ver1, ver3, ver4 };

CGaussBonnetVertex \*face3[3] = { ver1, ver4, ver2 };

CGaussBonnetVertex \*face4[3] = { ver4, ver3, ver2 };

mesh.createHullFace(face1, faceId++);

mesh.createHullFace(face2, faceId++);

mesh.createHullFace(face3, faceId++);

mesh.createHullFace(face4, faceId++);

mesh.writeHull\_m(outName);

//Smesh.write\_m(outName);

//建凸包

//初始化四个点

ver1->state() = IN\_FIRE;

ver2->state() = IN\_FIRE;

ver3->state() = IN\_FIRE;

ver4->state() = IN\_FIRE;

//开始遍历其他点

for (list<CGaussBonnetVertex\*>::iterator vertexIt = mesh.vertices().begin(); vertexIt != mesh.vertices().end(); vertexIt++)

{

CVertex \*t\_ver = \*vertexIt;

//t\_ver->string() = "";

///////////////////////////////

//mesh.writeHull\_m(outName);

///////////////////////////////

if (t\_ver->state() != IN\_FIRE) //未被访问

{

t\_ver->state() = IN\_FIRE; //标记为访问过

vector<CFace\*> det\_faces;//用于储存det为负的面

for (list<CGaussBonnetFace\*>::iterator faceIt = mesh.faces().begin(); faceIt != mesh.faces().end(); faceIt++) //遍历所有面 判断是否朝向当前点

{

CFace \*t\_face = \*faceIt;

CVertex \*t\_vers[3];//当前面上的三个点

for (int i = 0; i < 3; i++)

{

t\_vers[i] = t\_face->faceVertices(i);

}

int x = 0, y = 1, z = 2;

int j = 0, k = 1, l = 2;

//行列式！！！

double det = (t\_vers[j]->point()[x] - t\_ver->point()[x])\*

(t\_vers[k]->point()[y] - t\_ver->point()[y])\*

(t\_vers[l]->point()[z] - t\_ver->point()[z]) +

(t\_vers[l]->point()[x] - t\_ver->point()[x])\*

(t\_vers[j]->point()[y] - t\_ver->point()[y])\*

(t\_vers[k]->point()[z] - t\_ver->point()[z]) +

(t\_vers[j]->point()[z] - t\_ver->point()[z])\*

(t\_vers[k]->point()[x] - t\_ver->point()[x])\*

(t\_vers[l]->point()[y] - t\_ver->point()[y]) -

(t\_vers[j]->point()[z] - t\_ver->point()[z])\*

(t\_vers[k]->point()[y] - t\_ver->point()[y])\*

(t\_vers[l]->point()[x] - t\_ver->point()[x]) -

(t\_vers[j]->point()[x] - t\_ver->point()[x])\*

(t\_vers[k]->point()[z] - t\_ver->point()[z])\*

(t\_vers[l]->point()[y] - t\_ver->point()[y]) -

(t\_vers[l]->point()[z] - t\_ver->point()[z])\*

(t\_vers[j]->point()[y] - t\_ver->point()[y])\*

(t\_vers[k]->point()[x] - t\_ver->point()[x]);

if (det < 0) //面积为负 建面

{

//mesh.deleteFace((CGaussBonnetFace\*)t\_face); //不能直接删，否则空指针异常！！！

det\_faces.push\_back(t\_face); //先存起来

//t\_face->string() = "sharp = true";

}

}

//便利所有det为负的面

vector<vector<CVertex\*>> edges\_line; //存储建面序列

for (int d = 0; d < det\_faces.size(); d++)

{

CFace \*det\_t\_face = det\_faces[d];

for (int i = 0; i < 3; i++) //两个点两个点地加入

{

vector<CVertex\*> two\_v;

two\_v.push\_back(det\_t\_face->faceVertices(i));

two\_v.push\_back(det\_t\_face->faceVertices((i + 1) % 3));

edges\_line.push\_back(two\_v);

}

mesh.deleteHullFace((CGaussBonnetFace\*)det\_t\_face); //删除原有面

}

///////////////////////////////

//mesh.writeHull\_m(outName);

///////////////////////////////

//遍历所有边界边 （建面序列中非重复边）

for (int i = 0; i < edges\_line.size(); i++)

{

vector<CVertex\*> create\_face\_line = edges\_line[i];

bool flag = false; //不是重复的

for (int j = 0; j < edges\_line.size(); j++)

{

vector<CVertex\*> com\_line = edges\_line[j];

if (com\_line[0]->id() == create\_face\_line[1]->id() && com\_line[1]->id() == create\_face\_line[0]->id())

{

flag = true; //重复边

}

}

if (flag == false)

{

CGaussBonnetVertex \*new\_face\_vers[3] = { (CGaussBonnetVertex\*)create\_face\_line[0], (CGaussBonnetVertex\*)create\_face\_line[1], (CGaussBonnetVertex\*)t\_ver };

mesh.createHullFace(new\_face\_vers, faceId++);

}

}

///////////////////////////////

//mesh.writeHull\_m(outName);

///////////////////////////////

}

}

//mesh.writeHull\_m("meshes/VV.m");

//删掉向上的面

vector<CGaussBonnetFace\*> nor\_up\_faces;

for (list<CGaussBonnetFace\*>::iterator faceIt = mesh.faces().begin(); faceIt != mesh.faces().end(); faceIt++)

{

CFace \*t\_f = \*faceIt;

CPoint vi = t\_f->faceVertices(0)->point() - t\_f->faceVertices(1)->point();

CPoint vj = t\_f->faceVertices(1)->point() - t\_f->faceVertices(2)->point();

CPoint f\_n = vi ^ vj;

if (f\_n \* CPoint(0, 0, 1) > 0)

{

nor\_up\_faces.push\_back((CGaussBonnetFace\*)t\_f);

}

}

for (int i = 0; i < nor\_up\_faces.size(); i++)

{

mesh.deleteHullFace(nor\_up\_faces[i]);

}

mesh.writeHull\_m("RAND\_R.m");

//投影回平面

for (list<CGaussBonnetVertex\*>::iterator vertexIt = mesh.vertices().begin(); vertexIt != mesh.vertices().end(); vertexIt++)

{

CVertex \*t\_v = \*vertexIt;

CPoint t\_p = t\_v->point();

t\_v->point() = CPoint(t\_p[0], t\_p[1], 0);

}

//mesh.writeHull\_m(outName);

mesh.writeHull\_m(outName);

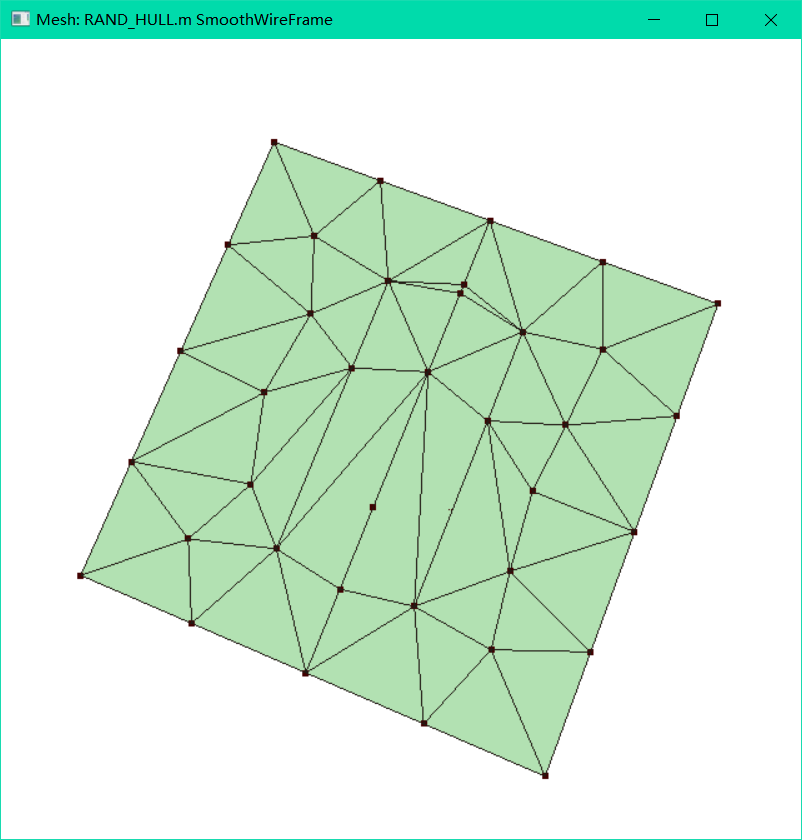
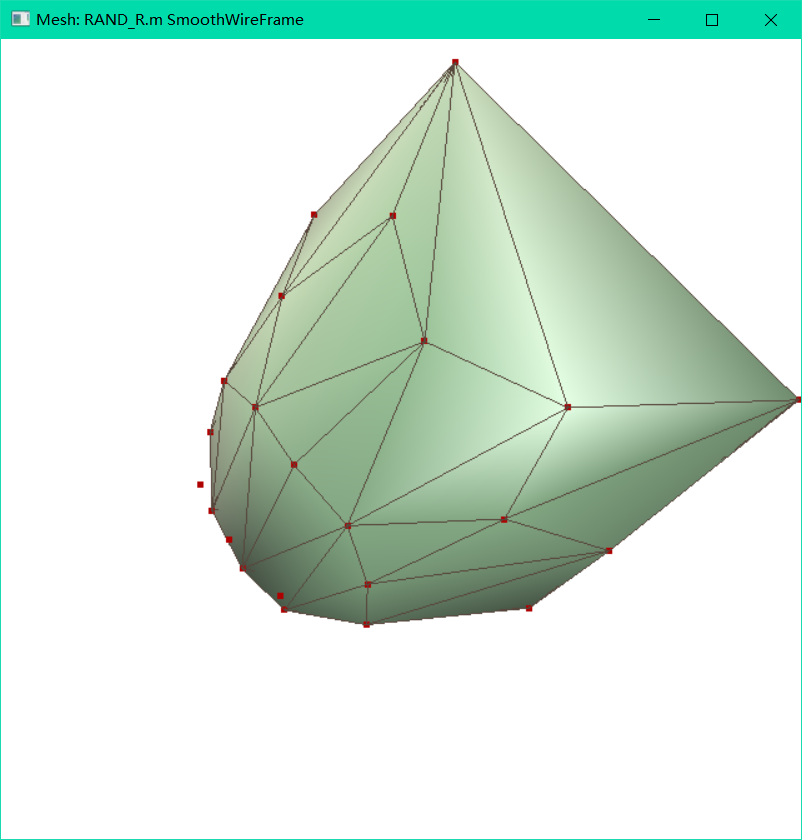
time(&end);

cout << difftime(end, start) << endl;

system("pause");

}

4.结果展示

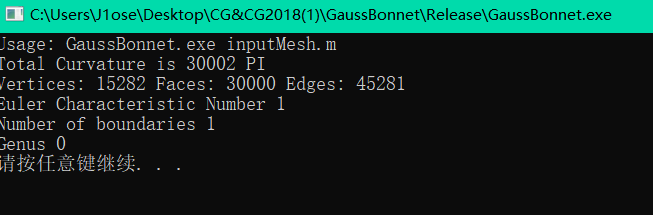


用时15s

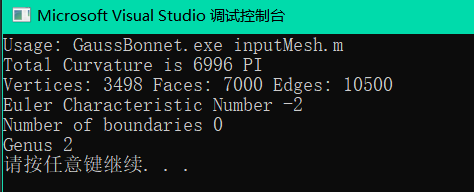
分析总结：我上网查阅了很多关于德洛内三角网格的资料，在以前我们上设计基础课的时候，LowPoly（低多边形）风格的图片很有设计感，当时我就在想有没有方法可以用程序来实现普通图片到LowPoly的转变，现在经过很多专业课的学习，我觉得可能用OpenCV等类库提取特征点，之后用这个算法可能就会很快的实现转化。这个算法难度不高，但是参考meshlib类库的修改给我造成了不小的困扰，最后在同学的帮助下成功的修改并跑通了。收获还是很多的。

1. 额外算法
2. 高斯-波涅定理

高斯波涅定理是我们的入门算法，开始的时候还给我们造成了不小的困扰,实际的简化说明就是，一个mesh的顶点数加面数-边数就等于它的欧拉示性数。欧拉示性数还可以用来计算亏格。对于我们的拓扑学基础知识了解很有帮助。



Sophie.m的欧拉示性数和亏格



Eight.m的欧拉示性数和亏格

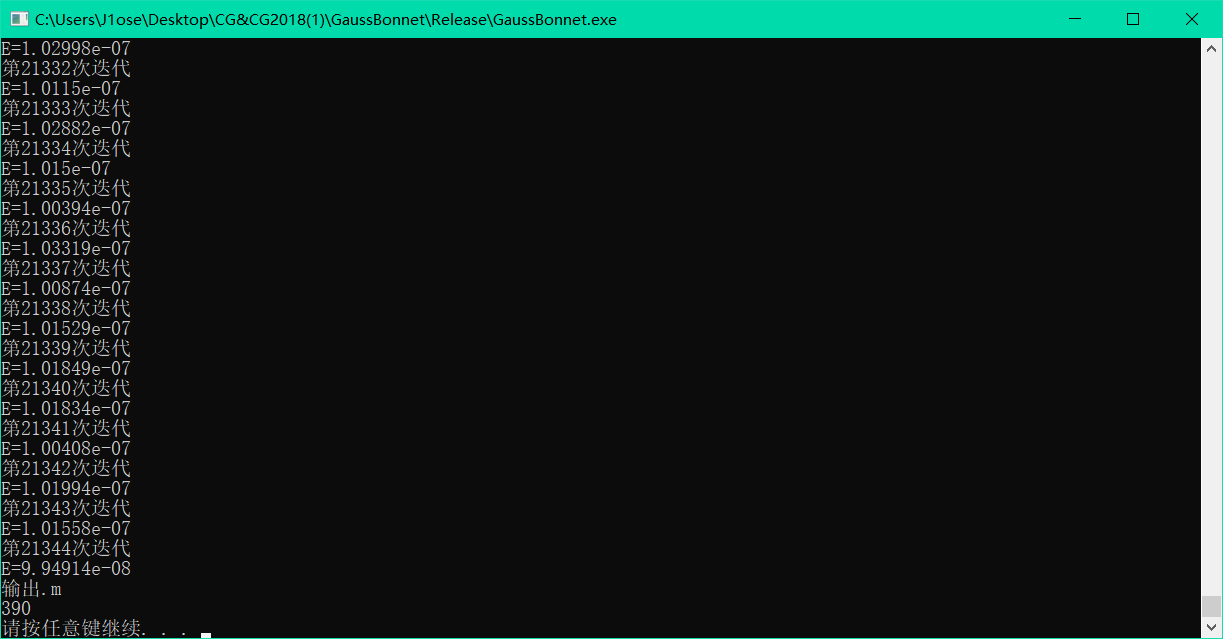
代码在项目中

1. 另一种调和映照拓扑圆盘的写法

运用了不同的K值算法，不同的迭代器，不同的能量计算。

虽然本质上没有什么大的改进，只是我和同学讨论时候他的另一种写法。

代码在项目中

结果稍微快了一点

写在最后：

虽然开始的几节课没有跟上给我后续的学习带来了不小的困扰，但是还是通过老师和同学的帮助，弄懂了四个要求的算法，也认识到这门课确实是非常难得的一门课，这门课的内容在模型网格处理上确实是非常艰深的。很少有学校有这方面的课程。在学习的过程中对拓扑学，模型网格处理，数据结构有了很多的帮助。同时也对以3D模型操作的原理有了初步的认识，发现虽然Maya等软件整合出的功能易于学习，但是真正能够实现功能背后是需要很过硬的数学知识的。这节课让我对这方面产生了浓厚的兴趣，也查阅了很多资料，也在课上问过老师一些问题,希望能在入门的基础上走的更远。