```
// FeatureDetector.cpp: FeatureDetector クラスのインプリメンテーション
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "MeshEditor.h"
#include "FeatureDetector.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

//////////////////////////////////////////////////////////////////////
// 構築/消滅
//////////////////////////////////////////////////////////////////////

FeatureDetector::FeatureDetector()
{

}

FeatureDetector::~FeatureDetector()
{

}

void FeatureDetector::ridge()
{
    int i;
    int vertex_N = mesh->vertex_N;
    //allocate memory
    if(mesh->isRidge != NULL)
        delete[] mesh->isRidge;
    mesh->isRidge = new BOOL[vertex_N];
    BOOL *isRidge = mesh->isRidge;

    if(mesh->isRavine != NULL)
        delete[] mesh->isRavine;
    mesh->isRavine = new BOOL[vertex_N];
    BOOL *isRavine = mesh->isRavine;

    if(mesh->k_max == NULL || mesh->k_min == NULL
        || mesh->t_max == NULL || mesh->t_min == NULL)
        mesh->computePrincipal();

    float *k_max = mesh->k_max;
    float *k_min = mesh->k_min;
    float (*t_max)[3] = mesh->t_max;
    float (*t_min)[3] = mesh->t_min;

    if(mesh->normal == NULL)
        mesh->computeNormal();

    float (*normal)[3] = mesh->normal;
```

```
    float (*vertex)[3] = mesh->vertex;

    for(i=0; i<vertex_N; i++){
        //skip boundary points
        if(mesh->isBound[i]){
            isRidge[i] = isRavine[i] = false;
            continue;
        }

        int nei_N, *nei;
        mesh->getConsistent1Ring(i, nei, nei_N);
        if(nei_N == 0){
            isRidge[i] = isRavine[i] = false;
            continue;
        }

        float k_max_max1, k_max_max2, k_max_min1, k_max_min2;
        float k_min_max1, k_min_max2, k_min_min1, k_min_min2;
        float l_max1, l_max2, l_min1, l_min2;

        float n1[3];
        MeshData::CROSS(n1, mesh->normal[i], mesh->t_max[i]);
        float n2[3];
        MeshData::CROSS(n2, mesh->normal[i], mesh->t_min[i]);

        for(int m=0; m<nei_N; m++){
            int j = nei[m];
            int k = nei[(m+1)%nei_N];

            float t1[3];
            MeshData::VEC(t1, vertex[j], vertex[i]);
            float t2[3];
            MeshData::VEC(t2, vertex[k], vertex[i]);

            float in1 = MeshData::DOT(n1, t1);
            float in2 = MeshData::DOT(n1, t2);
            if(in1 * in2 <= 0){
                float v[3];
                BOOL isOne = true;
                if(in1 < 0)
                    in1 = -in1;
                if(in2 < 0){
                    in2 = -in2;
                    isOne = false;
                }
                v[0] = (in2*t1[0] + in1*t2[0])/(in1 + in2);
                v[1] = (in2*t1[1] + in1*t2[1])/(in1 + in2);
                v[2] = (in2*t1[2] + in1*t2[2])/(in1 + in2);

                float o[3] = {0,0,0};
                float area1 = MeshData::AREA(o, t1, v);
                float area2 = MeshData::AREA(o, t2, v);
                float k_max_in = (area2*k_max[j] + area1*k_max[k])/(area1+area
2);
                float k_min_in = (area2*k_min[j] + area1*k_min[k])/(area1+area
```

```
2);

                if(isOne){
                    k_max_max1 = k_max_in;
                    k_min_max1 = k_min_in;
                    l_max1 = v[0]*v[0] + v[1]*v[1] + v[2]*v[2];
                }
                else{
                    k_max_max2 = k_max_in;
                    k_min_max2 = k_min_in;
                    l_max2 = v[0]*v[0] + v[1]*v[1] + v[2]*v[2];
                }
            }

            in1 = MeshData::DOT(n2, t1);
            in2 = MeshData::DOT(n2, t2);
            if(in1 * in2 <= 0){
                float v[3];
                BOOL isOne = true;
                if(in1 < 0)
                    in1 = -in1;
                if(in2 < 0){
                    in2 = -in2;
                    isOne = false;
                }
                v[0] = (in2*t1[0] + in1*t2[0])/(in1 + in2);
                v[1] = (in2*t1[1] + in1*t2[1])/(in1 + in2);
                v[2] = (in2*t1[2] + in1*t2[2])/(in1 + in2);

                float o[3] = {0,0,0};
                float area1 = MeshData::AREA(o, t1, v);
                float area2 = MeshData::AREA(o, t2, v);
                float k_max_in = (area2*k_max[j] + area1*k_max[k])/(area1+area
2);
                float k_min_in = (area2*k_min[j] + area1*k_min[k])/(area1+area
2);

                if(isOne){
                    k_max_min1 = k_max_in;
                    k_min_min1 = k_min_in;
                    l_min1 = v[0]*v[0] + v[1]*v[1] + v[2]*v[2];
                }
                else{
                    k_max_min2 = k_max_in;
                    k_min_min2 = k_min_in;
                    l_min2 = v[0]*v[0] + v[1]*v[1] + v[2]*v[2];
                }
            }
        }
        delete[] nei;

        float d1 = (float)sqrt(l_min1);
        float d2 = (float)sqrt(l_min2);

        float b = (k_max_min2-k_max[i])/d2 + (k_max[i]-k_max_min1)/d1
            - (k_max_min2-k_max_min1)/(d1+d2);
```

```
        float K1 = k_max_max1 + 0.5f*l_max1*b*b/(k_max[i]-k_min[i]);
        float K2 = k_max_max2 + 0.5f*l_max2*b*b/(k_max[i]-k_min[i]);

        isRidge[i] = (k_max[i] > K1) && (k_max[i] > K2);

        d1 = (float)sqrt(l_max1);
        d2 = (float)sqrt(l_max2);

        b = (k_min_max2-k_min[i])/d2 + (k_min[i]-k_min_max1)/d1
            - (k_min_max2-k_min_max1)/(d1+d2);

        K1 = k_min_min1 - 0.5f*l_min1*b*b/(k_max[i]-k_min[i]);
        K2 = k_min_min2 - 0.5f*l_min2*b*b/(k_max[i]-k_min[i]);

        isRavine[i] = (k_min[i] < K1) && (k_min[i] < K2);
    }
}

void FeatureDetector::constantThresholding(float T)
{
    int vertex_N = mesh->vertex_N;
    BOOL *isRidge = mesh->isRidge;
    BOOL *isRavine = mesh->isRavine;
    float *k_max = mesh->k_max;
    float *k_min = mesh->k_min;
    for(int i=0; i<vertex_N; i++){
        isRidge[i] &= (k_max[i] > T);
        isRavine[i] &= (k_min[i] < -T);
    }
}

void FeatureDetector::setMeshData(MeshData *mesh)
{
    this->mesh = mesh;
}

void FeatureDetector::automaticThresholding(float hi, float low)
{
    int vertex_N = mesh->vertex_N;
    float *k_max = mesh->k_max;
    float *k_min = mesh->k_min;
    BOOL *isRidge = mesh->isRidge;
    BOOL *isRavine = mesh->isRavine;
    int *isBound = mesh->isBound;

    BOOL* weak_ridge = new BOOL[vertex_N];
    BOOL* weak_ravine = new BOOL[vertex_N];
    for(int i=0; i<vertex_N; i++){
        weak_ridge[i] = false;
        weak_ravine[i] = false;
    }

    float k_max_ave = 0;
    float k_min_ave = 0;
```

```
float k_max_var = 0;
float k_min_var = 0;

for(i=0; i<vertex_N; i++){
    if(mesh->vertex_link[i]->next != NULL && !isBound[i]){
        k_max_ave += k_max[i];
        k_min_ave += k_min[i];
        k_max_var += k_max[i]*k_max[i];
        k_min_var += k_min[i]*k_min[i];
    }
}
k_max_ave /= (float)vertex_N;
k_min_ave /= (float)vertex_N;

k_max_var = k_max_var/((float)vertex_N) - k_max_ave*k_max_ave;
k_min_var = k_min_var/((float)vertex_N) - k_min_ave*k_min_ave;

float ridge_hi = k_max_ave + 0.25f*(float)sqrt(k_max_var);
float ridge_ro = k_max_ave - 0.53f*(float)sqrt(k_max_var);

float ravine_hi = k_min_ave - 0.25f*(float)sqrt(k_min_var);
float ravine_ro = k_min_ave + 0.53f*(float)sqrt(k_min_var);

for(i=0; i<vertex_N; i++){
    if(isBound[i])
        continue;

    if(isRidge[i]){
        if(k_max[i] > ridge_hi)
            isRidge[i] = true;
        else
            isRidge[i] = false;
        if(k_max[i] > ridge_ro)
            weak_ridge[i] = true;
        else
            weak_ridge[i] = false;
    }
    else
        weak_ridge[i] = false;

    if(isRavine[i]){
        if(k_min[i] < ravine_hi)
            isRavine[i] = true;
        else
            isRavine[i] = false;
        if(k_min[i] < ravine_ro)
            weak_ravine[i] = true;
        else
            weak_ravine[i] = false;
    }
    else
        weak_ravine[i] = false;
}

BOOL growth_flag = true;
while(growth_flag){
```

```
        growth_flag = false;
        for(i=0; i<vertex_N; i++){
            if(isBound[i])
                continue;

            if(isRidge[i]){
                int *nei, nei_N;
                mesh->get1Ring(i, nei, nei_N);

                for(int j=0; j<nei_N; j++){
                    int k = nei[j];
                    if(weak_ridge[k] && !isRidge[k]){
                        isRidge[k] = true;
                        growth_flag = true;
                    }
                }
                delete[] nei;
            }

            if(isRavine[i]){
                int *nei, nei_N;
                mesh->get1Ring(i, nei, nei_N);

                for(int j=0; j<nei_N; j++){
                    int k = nei[j];
                    if(weak_ravine[k] && !isRavine[k]){
                        isRavine[k] = true;
                        growth_flag = true;
                    }
                }
                delete[] nei;
            }
        }
    }

    delete[] weak_ridge;
    delete[] weak_ravine;
}

void FeatureDetector::precisionRidge()
{
    /*
    int vertex_N = mesh->vertex_N;
    int face_N = mesh->face_N;
    float (*vertex)[3] = mesh->vertex;
    float (*normal)[3] = mesh->normal;
    float *k_max = mesh->k_max;
    float *k_min = mesh->k_min;
    float (*t_max)[3] = mesh->t_max;
    float (*t_min)[3] = mesh->t_min;
    int* isBound = mesh->isBound;
    BOOL* isRidge = mesh->isRidge;
    BOOL* isRavine = mesh->isRavine;
    int i;

    if(mesh->ridge_tri == NULL)
```

```
    mesh->ridge_tri = new BOOL[mesh->face_N];
BOOL* ridge_tri = mesh->ridge_tri;

if(mesh->ravine_tri == NULL)
    mesh->ravine_tri = new BOOL[mesh->face_N];
BOOL* ravine_tri = mesh->ravine_tri;

for(i=0; i<face_N; i++){
    ridge_tri[i] = false;
    ravine_tri[i] = false;
}

for(i=0; i<vertex_N; i++){
    if(isBound[i])
        continue;

    if(isRidge[i]){
        int nei_N, *nei;
        mesh->getConsistent1Ring(i, nei, nei_N);

        int faceA = -1;
        int faceB = -1;
        float PA, PB, ka, kb;
        float n[3];
        int cA = 0;
        int cB = 0;
        MeshData::CROSS(n, normal[i], t_max[i]);
        for(int m=0; m<nei_N; m++){
            int j = nei[m];
            int k = nei[(m+1)%nei_N];

            float t1[3];
            MeshData::VEC(t1, vertex[j], vertex[i]);
            float t2[3];
            MeshData::VEC(t2, vertex[k], vertex[i]);

            float in1 = MeshData::DOT(n, t1);
            float in2 = MeshData::DOT(n, t2);
            if(in1 * in2 <= 0){
                float v[3];
                BOOL isA = true;
                if(in1 < 0){
                    in1 = -in1;
                }
                if(in2 < 0){
                    in2 = -in2;
                    isA = false;
                }
                v[0] = (in2*t1[0] + in1*t2[0])/(in1 + in2);
                v[1] = (in2*t1[1] + in1*t2[1])/(in1 + in2);
                v[2] = (in2*t1[2] + in1*t2[2])/(in1 + in2);

                float o[3] = {0,0,0};
                float area1 = MeshData::AREA(o, t1, v);
                float area2 = MeshData::AREA(o, t2, v);
                float k_jk = (area2*k_max[j] + area1*k_max[k])/(area1+area
```

```
2);

                    if(isA){
                        cA++;
                        faceA = face_id;
                        PA = v[0]*v[0] + v[1]*v[1] + v[2]*v[2];
                        ka = k_max[i] - k_jk;
                    }
                    else{
                        cB++;
                        faceB = face_id;
                        PB = v[0]*v[0] + v[1]*v[1] + v[2]*v[2];
                        kb = k_max[i] - k_jk;
                    }
                }
            }
            if(faceA >= 0 && faceB >= 0){
                if(PB*ka < PA*kb)
                    ridge_tri[faceA] = true;
                else
                    ridge_tri[faceB] = true;
            }
            if(faceA >= 0 && faceB < 0)
                ridge_tri[faceA] = true;
            else if(faceA < 0 && faceB >= 0)
                ridge_tri[faceB] = true;
        }

        if(isRavine[i]){
            int nei_N, *nei;
            mesh->getConsistent1Ring(i, nei, nei_N);

            int faceA = -1;
            int faceB = -1;
            float PA, PB, ka, kb;
            float n[3];
            MeshData::CROSS(n, normal[i], t_min[i]);
            for(int m=0; m<nei_N; m++){
                int j = nei[m];
                int k = nei[(m+1)%nei_N];

                float t1[3];
                MeshData::VEC(t1, vertex[j], vertex[i]);
                float t2[3];
                MeshData::VEC(t2, vertex[k], vertex[i]);

                float in1 = MeshData::DOT(n, t1);
                float in2 = MeshData::DOT(n, t2);
                if(in1 * in2 <= 0){
                    float v[3];
                    BOOL isA = true;
                    if(in1 < 0){
                        in1 = -in1;
                    }
                    if(in2 < 0){
                        in2 = -in2;
```

```
                    isA = false;
                }
                v[0] = (in2*t1[0] + in1*t2[0])/(in1 + in2);
                v[1] = (in2*t1[1] + in1*t2[1])/(in1 + in2);
                v[2] = (in2*t1[2] + in1*t2[2])/(in1 + in2);

                float o[3] = {0,0,0};
                float area1 = MeshData::AREA(o, t1, v);
                float area2 = MeshData::AREA(o, t2, v);
                float k_jk = (area2*k_min[j] + area1*k_min[k])/(area1+area
2);

                if(isA){
                    faceA = face_id;
                    PA = v[0]*v[0] + v[1]*v[1] + v[2]*v[2];
                    ka = k_min[i] - k_jk;
                }
                else{
                    faceB = face_id;
                    PB = v[0]*v[0] + v[1]*v[1] + v[2]*v[2];
                    kb = k_min[i] - k_jk;
                }
            }
        }
        if(faceA >= 0 && faceB >= 0){
            if(PB*ka < PA*kb)
                ravine_tri[faceA] = true;
            else
                ravine_tri[faceB] = true;
        }
        if(faceA >= 0 && faceB < 0)
            ravine_tri[faceA] = true;
        else if(faceA < 0 && faceB >= 0)
            ravine_tri[faceB] = true;
    }
  }*/
}
```