Henry LEFEVRE
Louis AMORE

Géométrie Numérique - Compte Rendu

Démarche de l'algorithme :

1. La première étape a été de calculer le barycentre associée à chaque face. Un tableau contenant chaque barycentre est utilisé afin d'associer pour la face i le barycentre situé sur la case n°i du tableau.

2. Ensuite, de la même manière nous avons calculer le barycentre des tétraèdres formé pour chaque arête du maillage par les points s0, s1 (les deux sommet de l'arête) et sf0, sf1 (les deux barycentres associés aux faces adjacentes de l'arête). Un tableau a été utilisé pour stocké le résultat dans le même ordre que le parcourt de chaque arête.

3. Ensuite, un nouveau tableau de point temporaire permet cette fois-ci de sauvegarder la position des points déplacer du maillage.
Ici rien de spécial, le calcul du barycentre des faces et du barycentre des arêtes adjacentes au sommet ce fait durant l'itération sur chaque sommet.

4. Pour la dernière étape, le principe est que pour chaque point du maillage déplacé (i.e étape 3) on calcul la face associé à ce point. Pour cela on itère sur les points déplacés et on cherche pour chaque pair d'arêtes consécutives, à partir de ce point, la face commune à ces deux arêtes afin de récupérer le barycentre voulu.

Nous avons donc crée une fonction *trouverFaceCommune* (voir annexe 2) qui prend en paramètre la liste des faces communes à chacune des arêtes et deux arêtes. Cette fonction retourne le numéro i de la face du maillage qui est commune aux deux arêtes passées en paramètre.

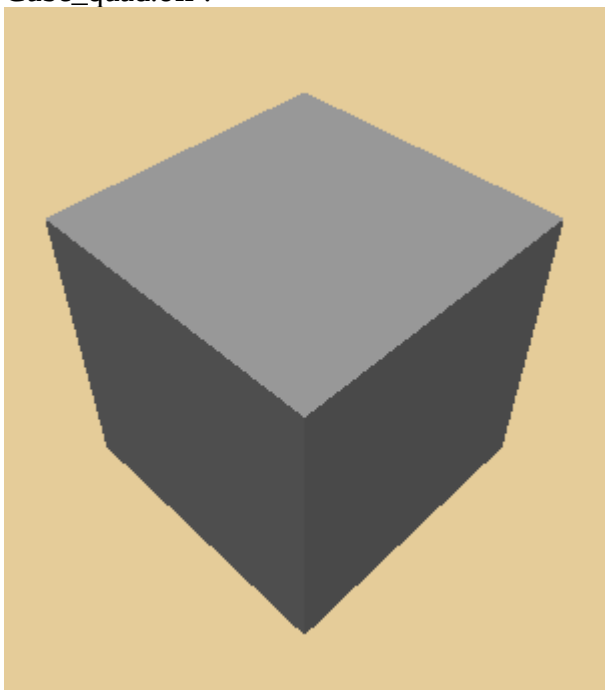
Nous avons aussi fait appelle à une deuxième fonction *trouverDoublon*, qui prend en paramètre une liste de points et un point. Cette fonction retourne l'indice i du point dans la liste s'il s'y trouve ou -1 sinon.
Cette fonction nous sert pour indicer les points à chaque face. Si la fonction retourne -1, alors on ajoute le point courant dans la liste finale des points du nouveau maillage et on prend comme indice sa position dans la liste, sinon on prend comme indice la valeur retourné par la fonction

Une fois les quatre indices des points trouvés la face peut être créée.

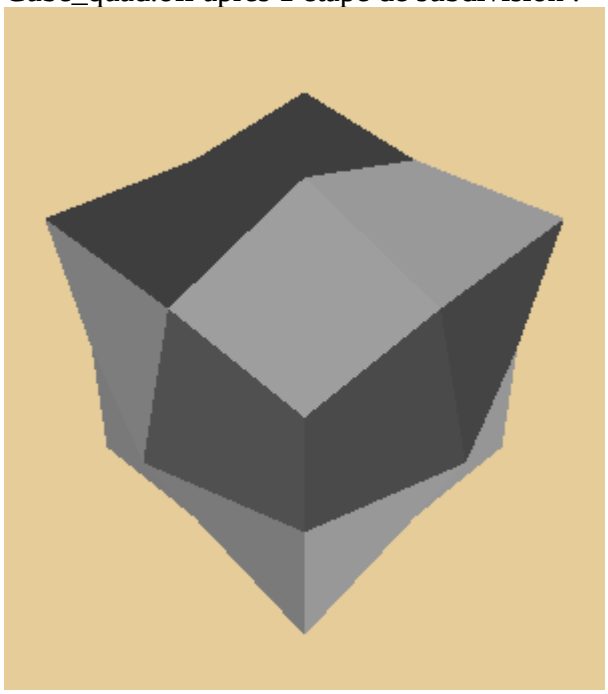
Cette méthode nous permet de faire une subdivision sur un maillage en entré, mais à partir de 2 subdivisions le résultat n'est plus cohérent, et nous n'avons pas compris pourquoi.

Résultat :

Cube_quad.off :



Cube_quad.off après 1 étape de subdivision :



Annexe 1 :

```
Mesh Mesh::subdivide() const
{
   Mesh output;



   //========================================================
   //
   // TODO : implémenter le schema de subdivision de Catmull-Clark
   //
   //========================================================


   //1. Ajouter le barycentre de chaque face du maillage

   vector<vec3> listBFaces; // liste des barycentre de chaque face. listBFaces[i] = indice barycentre de faces[i] dans la liste des vertices
   vector<vec3> listBEdges; // liste des barycentre de chaque face. listBEdges[i] = indice barycentre de get_edges()[i] dans la liste des vertices
   for (int i = 0; i < faces.size(); ++i) {

      vec3 somme(0, 0, 0);

      //Recherche du barycentre b de la face f
      for (int j = 0; j < get_face(i).size(); ++j) {
         somme += get_vertex(get_face(i)[j]);
      }

      somme /= (float) get_face(i).size();
      listBFaces.push_back(somme);
   }

   //2. Ajouter le barycentre du tetraedre

   cout <<"1" << endl;
   vector< Edge > edges = get_edges();
   cout <<"2" << endl;
   vector< vector< unsigned int > > facesOfEdge = get_edge_faces(edges);

   vector<vec3> listBTetraedre;

   for (int i = 0; i < edges.size(); ++i) {
      Edge e = edges[i];
      unsigned int s0 = e.m_i0;
      unsigned int s1 = e.m_i1;
      unsigned int f0 = facesOfEdge[i][0];
      unsigned int f1 = facesOfEdge[i][1];

      // Barycentre situe au meme indice que l'arete correspondante
      vec3 barycentre;
```

```
    barycentre += get_vertex(s0);
    barycentre += get_vertex(s1);
    barycentre += listBFaces[f0];
    barycentre += listBFaces[f1];
    barycentre /= 4.0;

    listBTetraedre.push_back(barycentre);
}


//3. Déplacer les points du maillage

vector< vector< unsigned int > > listVertexFaces = get_vertex_faces();
vector< vector< unsigned int > > listVertexEdges = get_vertex_edges(get_edges());

vector< vec3 > listVertexTmp;

for (int i = 0; i < vertices.size(); ++i) {
    vector< unsigned int > listFaces = listVertexFaces[i];
    vector< unsigned int > listEdges = listVertexEdges[i];

    // Barycentre des sfi
    vec3 F;
    for (int k = 0; k < listFaces.size(); ++k) {
        F += listBFaces[listFaces[k]];
    }
    F /= (float) listFaces.size();

    // Barycentre des sai
    vec3 A;
    for (int k = 0; k < listEdges.size(); ++k) {
        //A += listBEdges[k];
        A += listBTetraedre[listEdges[k]];
    }
    A /= (float) listEdges.size();

    vec3 s = get_vertex(i);
    int n = vertices.size();
    float x = (F[0] + 2*A[0] + (n - 3) * s[0]) / n;
    float y = (F[1] + 2*A[1] + (n - 3) * s[1]) / n;
    float z = (F[2] + 2*A[2] + (n - 3) * s[2]) / n;

    listVertexTmp.push_back(vec3(x, y, z));
}


//4. Former les faces avec les quadruplets

vector< unsigned int > faces;
vector< vec3 > vertices_mesh;
vector< vector< unsigned int > > faces_mesh;
```

```
    for (int i = 0; i < listVertexTmp.size(); ++i) {
        vec3 s = listVertexTmp[i];

        int nbArete = listVertexEdges[i].size();
        for (int j = 0; j < nbArete; ++j) {
            int e1 = listVertexEdges[i][j];
            int e2 = listVertexEdges[i][(j+1)%nbArete];
            int f = trouverFaceCommune(facesOfEdge, e1, e2);

            vec3 s1 = listBTetraedre[e1];
            vec3 s2 = listBFaces[f];
            vec3 s3 = listBTetraedre[e2];

            int ts = trouverDoublon(vertices_mesh, s);
            if (ts == -1) {
                vertices_mesh.push_back(s);
                ts = vertices_mesh.size()-1;
            }
            int ts1 = trouverDoublon(vertices_mesh, s1);
            if (ts1 == -1) {
                vertices_mesh.push_back(s1);
                ts1 = vertices_mesh.size()-1;
            }
            int ts2 = trouverDoublon(vertices_mesh, s2);
            if (ts2 == -1) {
                vertices_mesh.push_back(s2);
                ts2 = vertices_mesh.size()-1;
            }
            int ts3 = trouverDoublon(vertices_mesh, s3);
            if (ts3 == -1) {
                vertices_mesh.push_back(s3);
                ts3 = vertices_mesh.size()-1;
            }

            faces.clear();
            faces.push_back(ts);
            faces.push_back(ts3);
            faces.push_back(ts2);
            faces.push_back(ts1);

            faces_mesh.push_back(faces);
        }
    }

    // Creation du nouveau maillage
    output.vertices = vertices_mesh;
    output.faces = faces_mesh;

    return output;
}
```

Annexe 2 :

```
unsigned int trouverFaceCommune(vector< vector< unsigned int > > &facesOfEdge, unsigned int e1, unsigned int e2) {

    if (facesOfEdge[e1][0] == facesOfEdge[e2][0]) {
        return facesOfEdge[e1][0];

    } else if (facesOfEdge[e1][0] == facesOfEdge[e2][1]) {
        return facesOfEdge[e1][0];

    } else if (facesOfEdge[e1][1] == facesOfEdge[e2][0]) {
        return facesOfEdge[e1][1];

    } else if (facesOfEdge[e1][1] == facesOfEdge[e2][1]) {
        return facesOfEdge[e1][1];
    }

    return -1;
}
```

Annexe 3 :

```
int trouverDoublon(vector< vec3 > &vertices, vec3 vertex) {
    for (int i = 0; i < vertices.size(); ++i) {
        vec3 p = vertices[i];
        if (p.x == vertex.x && p.y == vertex.y && p.z == vertex.z) {
            return i;
        }
    }
    return -1;
}
```