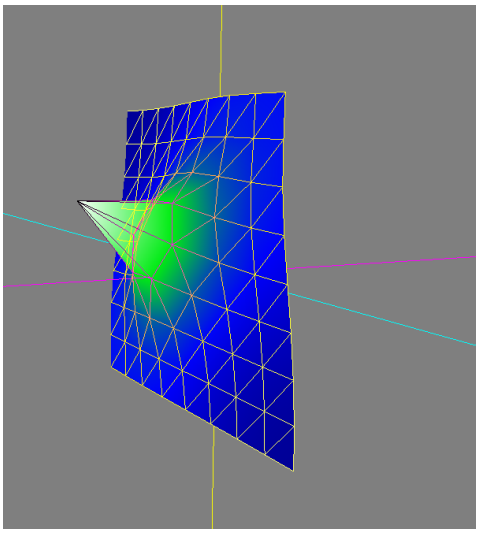
Charlie Grand

Rapport TP-IG2 - Géométrie

M2 IAFA



Contents

List of Figures

1 Introduction

1.1 Bibliothèques

La bibliothèque utilisée pour la gestion des maillages et l'affichage est le projet réalisé pendant les TP précédent d'informatique graphique et sur mon temps libre, accessible sur Github. Cette bibliothèque est écrite en C++ et utilise OpenGL pour l'affichage.

Il y a plus d'informations sur le readme du projet.

La bibliothèque Eigen est utilisée pour la résolution de système linéaire.

1.2 Organisation du code

Le code du TP est organisé en quatre fichiers à la racine du projet.

- laplacianIt.cpp

  Ce fichier implémente le calcul itératif du Laplacien par diffusion.

- laplacianMat.cpp

  Ce fichier implémente le calcul de Laplacien par résolution de système linéaire.

- laplacianSmooth.cpp

  Ce fichier implémente le lissage Laplacien itératif.

- laplacianTransform.cpp

  Ce fichier implémente une transformation pondérée par le Laplacien.

1.3 Compilation et exécution

Pour compiler le projet il suffit d'exécuter la commande suivante à la racine du projet après avoir spécifié le chemin vers Eigen à la ligne 22 du makefile.

```
make
```

Quatre exécutables correspondant respectivement aux fichiers cités plus haut sont créés dans le dossier build/.

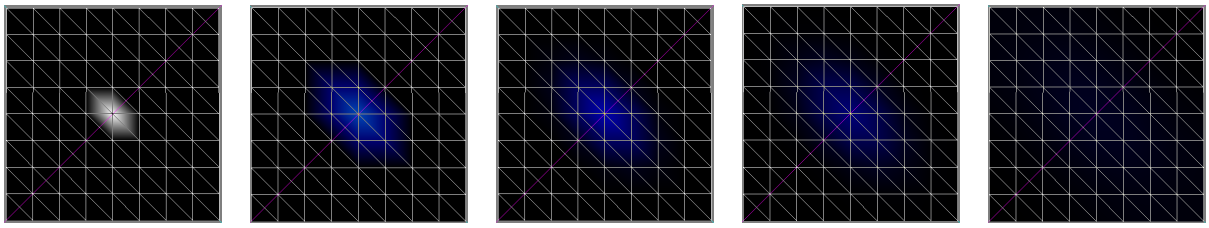
- iteratif
- mat
- smooth

- transf

Au lancement d'un de ces exécutables, les calculs respectifs sont effectués puis une instance du moteur de rendu temps réel est lancée afin de visualiser les résultats.

La navigation dans l'espace 3D se fait grâce aux touches ZQSD pour déplacer la caméra et le clique-molette pour la faire pivoter.

Dans certains exécutables la caméra ne pointe pas toujours sur les modèles 3D à l'initialisation et il faut naviguer pour les trouver dans l'espace.



(a) Initialisation (b) Itération 1 (c) Itération 2 (d) Itération 3 (e) Itération 10

Figure 1: Résultats du calcul itératif du Laplacien

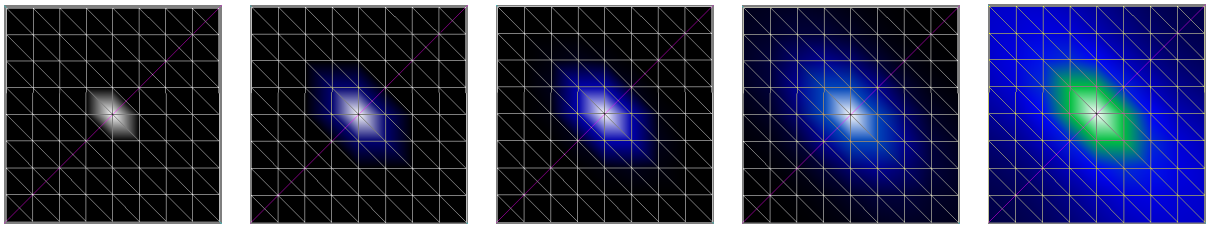
2 Calcul itératif du Laplacien par diffusion

Le calcul d'une itération du Laplacien par diffusion est donné par le code suivant (présent dans le fichier laplacianIt.cpp).

```
void vertexLaplacian(MeshVertex* v, float alpha)
{
    vector<MeshVertex*> neighbors = v->getVerticesAround();

    float sum = 0;
    for(MeshVertex* neighbor: neighbors)
    {
        sum += neighbor->getLaplacian() - (1 - alpha) * v->
            getLaplacian();
    }

    v->setLaplacian(alpha * sum / neighbors.size());
}

void laplacianIt(Mesh* mesh, MeshVertex* heatVertex, float alpha)
{
    vector<MeshVertex*> vertices = mesh->getVertices();

    for(MeshVertex* v: vertices)
    {
        vertexLaplacian(v, alpha);
    }

    // heatVertex->setLaplacian(1.0f);
    mesh->applyLaplacian();
}
```

La figure 1 montre le résultat sur un maillage triangulaire régulier avec cet appel de fonction.

```
    laplacianIt(mesh, heatVertex, 0.99);
```



(a) Initialisation (b) Itération 1 (c) Itération 2 (d) Itération 10 (e) Itération 50

Figure 2: Résultats du calcul itératif du Laplacien avec émission constante en un point

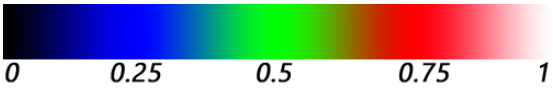


Figure 3: Expression des couleurs en fonction de la valeur du Laplacien

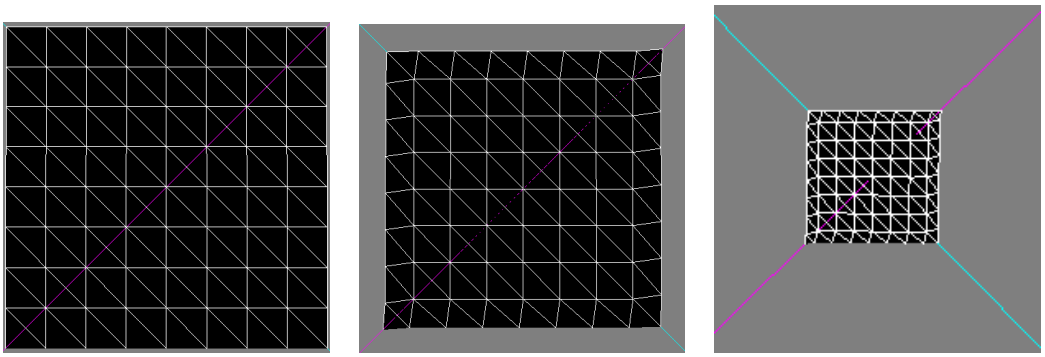
La figure 2 montre les résultats du calcul itératif du Laplacien par diffusion avec une émission constante en un point et la figure 3 montre l'échelle de couleur utilisée pour représenter la valeur du Laplacien. Pour obtenir ce résultat il faut dé-commenter la ligne 23 du code précédent.

3 Lissage itératif Laplacien

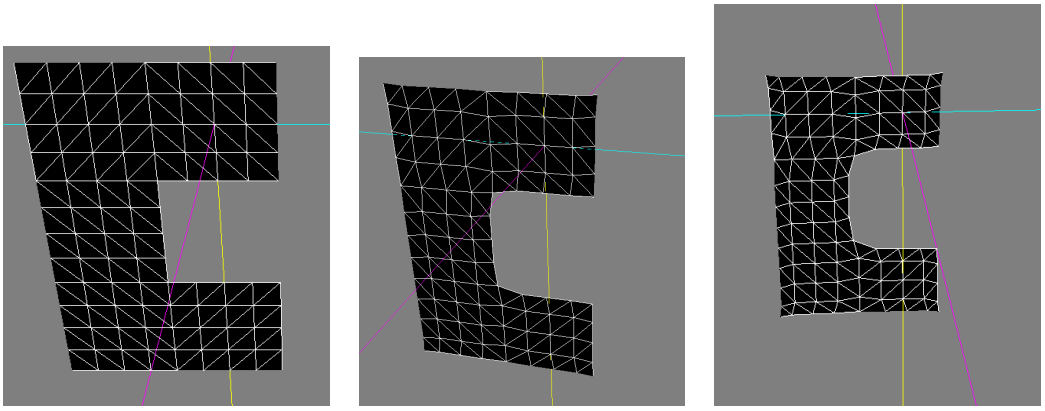
Sur le même principe que la diffusion du Laplacien itératif voici l'algorithme du lissage itératif Laplacien (présent dans le fichier laplacianSmooth.cpp).

```
void laplacianSmooth(Mesh* mesh, float alpha, float lambda)
{
    vector<MeshVertex*> vertices = mesh->getVertices();
    vector<vec3> laplacians;

    for(MeshVertex* v: vertices)
    {
        vector<MeshVertex*> neighbors = v->getVerticesAround();

        vec4 sum = vec4(0);
        for(MeshVertex* n: neighbors)
        {
            sum += n->getAttribute(0) - (1 - alpha) * v->
               getAttribute(0);
        }

       laplacians.push_back((alpha/neighbors.size()) * vec3(sum.x
          , sum.y, sum.z) );
    }

    for(int i = 0; i < laplacians.size(); i++)
    {
        mat4 transform = translate(laplacians[i]);
        vertices[i]->setCoord(lambda * transform * vertices[i]->
           getAttribute(0));
    }
}
```

La figure 4 montre les effets du lissage itératif Laplacien sur deux maillages différents. Ce n'est pas très visible sur les images mais le volume diminue à chaque itération.



(a) Initialisation (b) Itération 1 (c) Itération 2



(d) Initialisation (e) Itération 1 (f) Itération 2

Figure 4: Résultats du lissage Laplacien itératif

4 Calcul par résolution de système linéaire du Laplacien

La structure de données utilisée pour stocker les points dont la valeur du Laplacien est fixée est un map de la bibliothèque standard C++ et la clé utilisée pour chaque point est son indice du VBO OpenGL afin de s'assurer de son unicité. Ce n'est sûrement pas la meilleure solution mais cela fonctionne dans ce cas précis et permet de comparer deux points facilement.

La sélection de point est un peu rudimentaire mais fonctionne. Il faut appeler la fonction

```
vector<MeshVertex*> Mesh::getVertex(vec3 coord)
```

permettant d'obtenir la liste des points du maillage ayant ces coordonnées.

Une fois construite la map contenant les points à valeur Laplacienne fixée on peut utiliser la fonction suivante (présente dans le fichier laplacianMat.cpp) pour le calcul de toutes les valeurs.

```
void laplacian(Mesh* mesh, map<int, std::pair<float, MeshVertex
    *>> heatPoints)
{
    vector<MeshVertex*> vertices = mesh->getVertices();
    int nVertices = vertices.size();

    // init matrices
    Eigen::MatrixXf vertexFactors(nVertices,nVertices);
    Eigen::VectorXf values(nVertices);

    for(int i = 0; i < nVertices; i++)
    {
        if(mapContains(heatPoints, vertices[i]->getNumber()))
        {
            values[i] = heatPoints.find(vertices[i]->getNumber())
                ->second.first;

            for(int j = 0; j < nVertices; j++)
            {
                if(i==j)
                    vertexFactors(i,j) = 1.f;
                else
                    vertexFactors(i,j) = 0.f;
            }
        }
        else if(isBoundary(vertices[i]))
```

```
25          {
26              values[i] = 0.f;
27
28              for(int j = 0; j < nVertices; j++)
29              {
30                  if(i==j)
31                      vertexFactors(i,j) = 1.f;
32                  else
33                      vertexFactors(i,j) = 0.f;
34              }
35          }
36          else
37          {
38              int nNeighbors = 0;
39              for(int j = 0; j < nVertices; j++)
40              {
41                  if(areNeighbors(vertices[i], vertices[j]))
42                  {
43                      std::pair<MeshVertex*, MeshVertex*>
                            sharedNeighbors = getSharedNeighbors(
                            vertices[i], vertices[j]);
44
45                      if(sharedNeighbors.first != nullptr &&
                            sharedNeighbors.second != nullptr)
46                      {
47                          vec4 ca = vertices[i]->getAttribute(0) -
                                sharedNeighbors.first->getAttribute(0)
                                ;
48                          vec4 cb = vertices[j]->getAttribute(0) -
                                sharedNeighbors.first->getAttribute(0)
                                ;
49                          vec4 da = vertices[i]->getAttribute(0) -
                                sharedNeighbors.second->getAttribute
                                (0);
50                          vec4 db = vertices[j]->getAttribute(0) -
                                sharedNeighbors.second->getAttribute
                                (0);
51
52                          vertexFactors(i,j) = (float) ((dot(ca, cb
                                ) + dot(da, db)));
53                      } else
54                      {
55                          vertexFactors(i,j) = 0.f;
56                      }
57                      nNeighbors++;
58                  }
59                  else
```

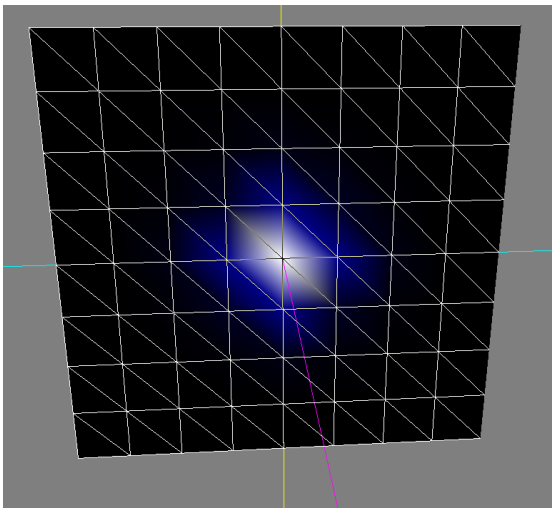


Figure 5: Calcul du Laplacien par résolution d'un système linéaire sur un maillage triangulaire régulier

```
60                      vertexFactors(i,j) = 0.f;
61              }
62              vertexFactors(i,i) = -nNeighbors;
63              values[i] = 0.f;
64          }
65      }
66
67      // compute X matrix
68      // SVD decomposition of a
69      Eigen::BDCSVD<Eigen::MatrixXf> svd(vertexFactors, Eigen::
            ComputeFullU+Eigen::ComputeFullV);
70
71      // solve linear system ax = b
72      Eigen::VectorXf x = svd.solve(values);
73
74      // set laplacian value to vertices
75      for(int i = 0; i < nVertices; i++)
76      {
77          vertices[i]->setLaplacian(x[i]);
78          vertices[i]->applyLaplacian();
79      }
80  }
```

La figure 5 montre le résultat du calcul sur un maillage triangulaire régulier.

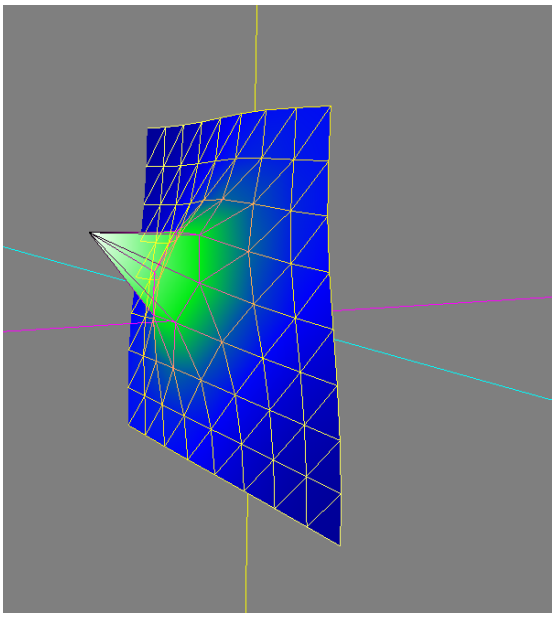


Figure 6: Transformation pondérée par le Laplacien

5 Transformation pondérée par le Laplacien

La transformation pondérée par le Laplacien est une fonction appliquant une transformation sur un maillage (ici une translation) pondérée par la valeur du Laplacien en tout point du maillage calculée au préalable par une des deux méthodes exposées plus haut. Voici son code (présent dans le fichier laplacianTransform.cpp).

```
1  void transformLaplacian(Mesh* mesh, vec3 tr)
2  {
3      vector<MeshVertex*> vertices = mesh->getVertices();
4
5      for(MeshVertex* v: vertices)
6      {
7          mat4 transform = translate(v->getLaplacian() * tr);
8          v->setCoord(transform * v->getAttribute(0));
9      }
10 }
```

La figure 6 montre le résultat de la transformation sur un maillage triangulaire régulier.

5.1 Fonction de transfert

Une fonction de transfert va modifier la valeur de la pondération par le Laplacien afin de modifier le profil de la déformation. Pour utiliser une fonction de transfert il faut remplacer la ligne 7 de la fonction de transformation par la ligne suivante.

```
1  mat4 transform = translate(transfertFunction(v->getLaplacian()) *
       tr);
```

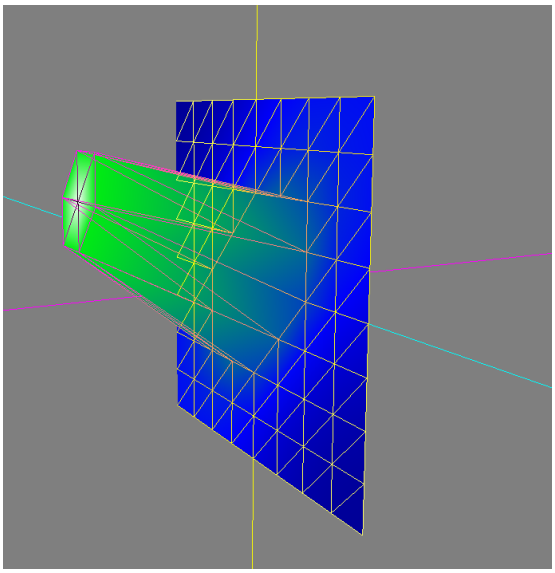


Figure 7: Transformation pondérée par le Laplacien avec une fonction de transfert porte

Dans le cas de la figure 7, une fonction porte est utilisée comme fonction de transfert pour bien visualiser son effet. En voici son code (présent dans le fichier laplacianTransform.cpp).

```
1 float transfertFunction(float x)
2 {
3     if(x < 0.4)
4         return 0.f;
5     return 1.f;
6 }
```

6 Conclusion

Toutes les méthodes implémentées fonctionnent assez bien, sauf la méthode par résolution de système linéaire. En effet, après le calcul on peut constater que la valeur du Laplacien décroît très vite et les valeurs sont très proches de zéro.

Un meilleur choix de couleur pour afficher le Laplacien m'aurait sans doute aider à mieux comprendre ce qu'il se passe.

L'utilisation de mon propre code pour la gestion de maillages et l'affichage 3D m'a rajouté beaucoup de travail mais m'a permis de comprendre beaucoup plus de chose sur la structure d'un moteur de rendu 3D temps réel et sur la programmation C++.