Sup Galilée
Mathématiques Appliquées et Calcul Scientifique
Deuxième année
Année universitaire 2018–2019

Langage C++(G4SCCP)
Cours : X. Juvigny
TPs : S. Landier
samlandier@gmail.com

TP « projet »

Les bases du langage C++

https://github.com/Macs1718/Promotion_2020
https://en.cppreference.com/w/
https://stackoverflow.com/

Exercice 14 (classes Triangle, Quadrangle et interface Polygon)

L'objectif de cet exercice est de mettre en place la hiérarchie de classe pour les éléments surfaciques de maillage permettant de spécialiser les calculs et d'abstraire les algorithmes.

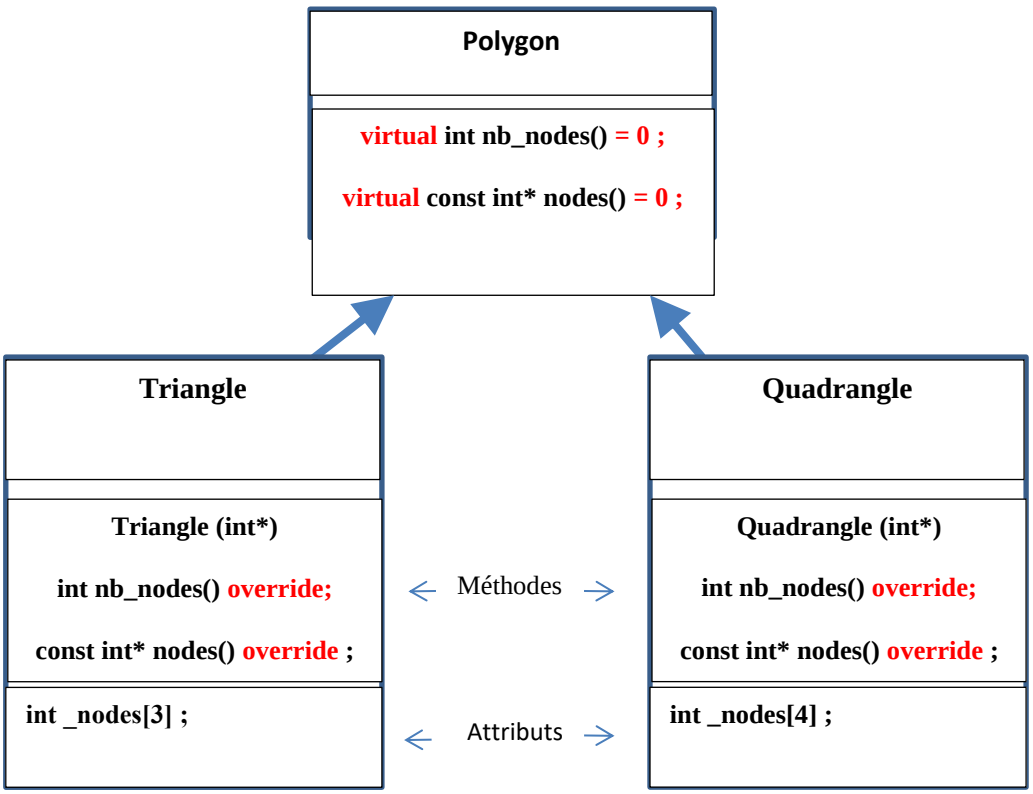

Fig. 1 : hiérarchie de classe

1) Création de la hiérarchie de classe

Il s'agit ici de créer 3 classes :

- Classe Polygon : classe *abstraite* (non instantiable) qui définit uniquement une interface pour les éléments surfaciques. Créer uniquement le fichier Polygon.h
  Elle ne possède pas d'attributs, pas de constructeurs, et possède au moins une méthode virtuelle pure (signature commençant par le mot clé « virtual » et se terminant par « =0 » pour dire que la classe n'implémente pas cette fonction mais oblige les classe héritantes à l'implémenter).

- Classe Triangle : classe *concrète* (i.e. possède au moins un constructeur et définit le destructeur) qui hérite de la classe Polygon. Créer les fichiers Triangle.h et Triangle.cpp.
  - Elle définit un constructeur qui prend un pointeur d'entier en paramètre. L'implémentation consiste à renseigner l'attribut « _nodes » à partir de ce pointeur.
  - Définir la méthode virtuelle nb_nodes qui retourne le nombre de sommets.
  - Définir la méthode nodes() qui retourne un pointeur vers l'attribut _nodes.
- Classe Quadrangle : même chose que la classe Triangle, sauf qu'on a 4 sommets.

2) Mise en action du polymorphisme
   Créer un fichier exo14.cpp, y définir une fonction main dans laquelle :

a) Un objet *t* de type Triangle est créé (choisir n'importe quoi pour les données du constructeur)
b) Idem avec un objet de type Quadrangle : *q*
c) Définir 2 pointeurs *p1* et *p2* de type Polygon, affecter les adresses respectives des objets t et q.
d) Appeler la fonction nb_nodes() via les pointeurs p1 et p2 et additionner les 2 valeurs. Vérifier que l'on obtient bien 7.

⇨ *C'est ce que l'on appelle le polymorphisme : un pointeur de type Polygon* peut pointer en réalité vers un objet d'une classe dérivée : depuis ce pointeur, on ne peut appeler que des méthodes déclarées dans Polygon, mais si l'une d'elle est surdéfinie dans la classe dérivée, c'est cette dernière qui est appelée.*

3) Chainage des destructeurs

a) Ajouter un affichage de sortie dans chacun des destructeurs (de Polygon, Triangle et Quadrangle). Par exemple pour Triangle :

```
std ::cout << «Triangle destructor call » << std ::endl ;
```

b) Compilez et exécutez le programme.

- Quel est l'ordre d'appel des destructeurs entre classe de base et classe dérivée ?
- L'objet Triangle est instancié avant l'objet Quadrangle. Qu'en est-il de l'ordre de leur destruction ? Quelle règle générale pouvez-vous en tirer quant à l'ordre de création par rapport à l'ordre de destruction des objets ?

c) Instancier maintenant dynamiquement (avec new) un autre objet t2 de type triangle et affectez son adresse à un pointeur *pPoly* de Polygon. Appelez tout de suite après l'opérateur delete sur pPoly. Est-ce que la destruction de t2 vous semble correcte ? Pourquoi ? Comment y remédier ?

Exercice 15 (« elements factory » : création des objets Triangles et Quadrangle à partir des tableaux Array<int>)

Cet exercice nécessite d'avoir défini correctement la classe template Array (exo 13). Une version de ce fichier est téléchargeable depuis le github.

Télécharger le fichier catT3Q4.mesh depuis le github. Ce fichier contient une version du modèle de chat avec des éléments mixtes, triangles et quadrangles.

Lorsqu'on effectue l'instruction suivante :

```
medith::read("catT3Q4.mesh", crd, cntE2, cntT3, cntQ4, cntTH4, cntHX8);
```

- cntE2 contient en sortie tous les segments contenus dans le fichier
- cntT3 contient en sortie tous les triangles contenus dans le fichier
- cntQ4 contient en sortie tous les quadrangles contenus dans le fichier
- cntTH4 contient en sortie tous les tétraèdres contenus dans le fichier
- cntHX8 contient en sortie tous les hexaèdres contenus dans le fichier

Avec le fichier « catT3Q4.mesh », seuls les containers cntT3 et cntQ4 contiendront des données.

L'objectif de cet exercice est de convertir ces données dans un formalisme objet et de gérer le cycle de vie de ces objets, c'est-à-dire de fournir les fonctions de création et de destructions des objets triangles et quadrangles correspondants (cf. note 1 et 2). Afin de vérifier l'exactitude de votre implémentation, on fera comme à l'exercice précédent un calcul simple sur les sommets grâce au polymorphisme (cf. note 3)).

Définir le main suivant dans un fichier exo15.cpp et créer les 3 fonctions (en gras) utilisées et remplacer le commentaire en vert par une procédure de vérification (cf. note 4):

```
int main()
{
  Array<int> cntE2(2), cntT3(3), cntQ4(4), cntTH4(4), cntHX8(8);
  Array<double> crd(3);

  medith::read("catT3Q4.mesh", crd, cntE2, cntT3, cntQ4, cntTH4, cntHX8);

  std::vector<Polygon*> pgs;
  create_polygons(cntT3, pgs);
  create_polygons(cntQ4, pgs);

  int total_nodes = count_nodes(pgs);

  // VERIFIER ICI LA VALEUR DE total_nodes

  destroy_polygons(pgs);

  return 0;
}
```

Notes :

1) create_polygons
   Une fonction qui parcourt un container *cnt* de type Array<int> et qui créé un objet (opérateur new) de type Triangle ou de type Quadrangle selon le stride de cnt et qui stocke en sortie un pointeur de type Polygon sur l'objet créé, qu'il soit un triangle ou un quadrangle.

Voici la signature de la fonction :

```
void create_polygons(const Array<int>& cnt, std::vector<Polygon*> & pgs) ;
```

2) destroy_polygons
   Une fonction de nettoyage qui parcourt la liste des polygones créés et qui les détruit (appel à l'opérateur delete pour chaque élément de la liste)

```
void destroy_polygons(std::vector<Polygon*> & pgs) ;
```

3) count_nodes
   Cette fonction parcourt la liste des polygones en entrée, appelle pour chaque polygone la fonction nb_nodes(), et retourne en sortie la valeur cumulée de ces appels.

4) Procédure de vérification :
   Il faudra s'assurer que la valeur retournée est bien la valeur attendue :

   Nombre de triangle x 3 + nombre de quadrangles x 4

Exercice 16 (collision d'une surface avec un plan)

Notations : les termes en gras désignent des vecteurs de IR^3.

L'objectif de cet exercice est de définir une fonction qui détecte tous les polygones qui intersectent un plan donné. Le plan est défini par un point P et un vecteur normal **n**.

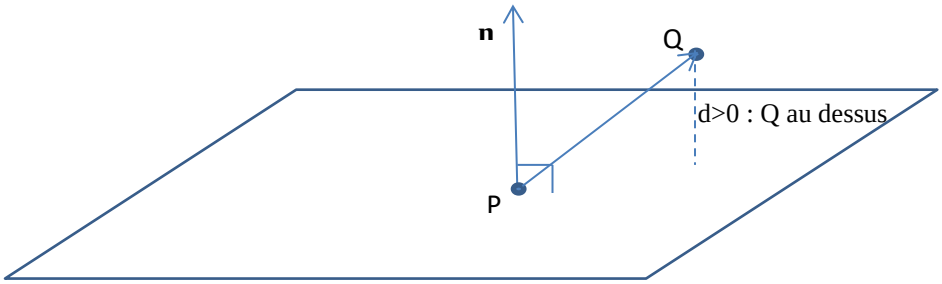

Fig. 2 : convention de localisation d'un point par rapport à un plan orienté.

Par convention, on considère que la normale pointe vers le demi -espace considéré « au-dessus » du plan. On appellera distance signée à un plan, la distance d'un point à ce plan, avec le signe « + » si il est situé « au-dessus », et « - » sinon.

1) Distance signée d'un point par rapport à un plan orienté
   Définir la fonction qui calcule la distance signée d'un point Q de IR^3 par rapport au plan (P,**n**) : d = **PQ**.**n**
   Définir cette fonction comme une méthode de Polygon :

   ```
   double signed_distance(const double* P, const double* n, const double* Q)
   ```

2) Collision d'un polygone avec un plan orienté
   Le critère de collision se base sur la distance signée des sommets du polygone par rapport au plan :
   Si toutes les distances signées sont de même signe, alors il n'y a pas de collision. Sinon, au moins un point se trouve de l'autre côté du plan par rapport aux autres sommets et il y a collision.

   Définir cette fonction comme une méthode de Polygon (on pourra l'implémenter directement dans le header) :

   ```
   bool collide_plane(const double* P, const double* n, const Array<double>& crd)
   ```

Note : Cette fonction retourne true si l'instance de polygone (triangle ou quadrangle) qui l'appelle intersecte le plan, false sinon.

3) Main et exécution.

a) Définir le main suivant dans un fichier exo16.cpp et y implémenter la fonction colliding_polygons :

```
void colliding_polygons(std::vector<Polygon*>& pgs, double* Pt, double* n, const
Array<double>& crd, std::vector<Polygon*>& opgs)
{
      // TODO :
      // Cette fonction doit parcourir tous les polygones et stocke dans le vecteur de sortie opgs les

     // adresses des polygones intersectant le plan (Pt, n)

}

int main()
{
  Array<int> cntE2(2), cntT3(3), cntQ4(4), cntTH4(4), cntHX8(8);
  Array<double> crd(3);

  medith::read("D:\\slandier\\DATA\\projects\\cours\\2019TP\\cases\\catT3Q4.mesh",
crd, cntE2, cntT3, cntQ4, cntTH4, cntHX8);

  std::vector<Polygon*> pgs;
  create_polygons(cntT3, pgs);
  create_polygons(cntQ4, pgs);

  std::vector<Polygon*> opgs;
  double P[] = { 0.053,1.847, 1.312 };
  double n[] = { 1., 0., 0. };

  colliding_polygons(pgs, P, n, crd, opgs);

  medith::write("xplane.mesh", crd, opgs);

  destroy_polygons(pgs);

  return 0;
}
```

b) Validation de votre implémentation

Vérifiez que l'on obtient bien la sélection suivante en ouvrant le fichier xplane.mesh avec medit en effectuant la commande suivante dans un terminal :

/LOCAL/medit-3.0/medit-linux xplane.mesh

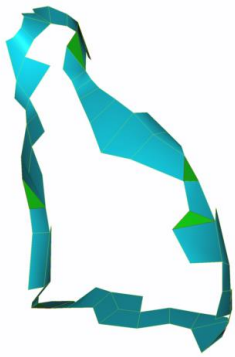

Fig. 3 : Sélection des 56 polygones intersectant un plan.

Exercice 17 (collision d'une surface avec un maillage volumique)

A partir du github, récupérez les maillages catT3.mesh et bgmTH4.mesh. Rapatriez aussi la dernière version de medit.hxx.

La configuration de départ est un maillage volumique fait de cellules tétraédriques (bgmTH4.mesh) dans lequel on immerge la surface du chat (catT3.mesh) :

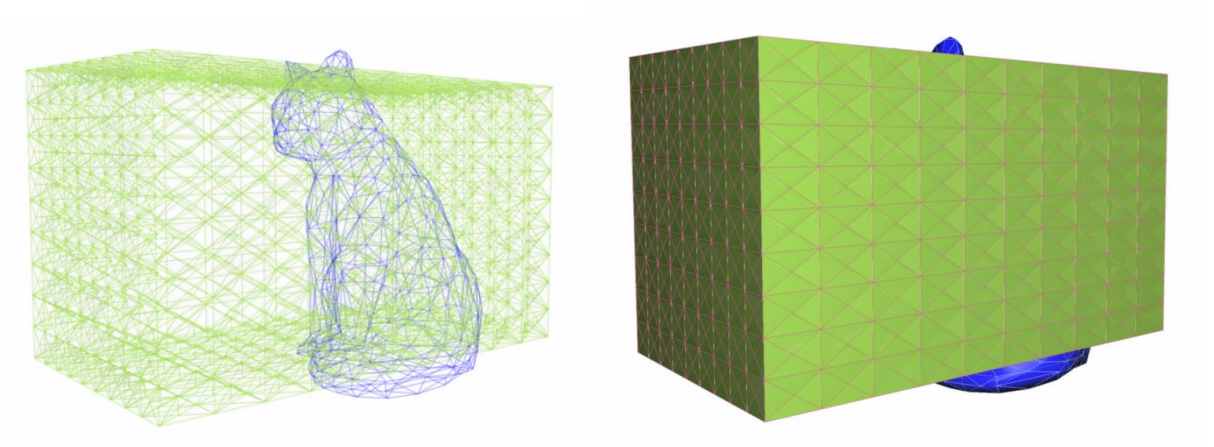

Fig. 4 : maillage tétraédrique immergeant un maillage surfacique

L'objectif est de capturer tous les tétraèdres qui contiennent un point du chat :

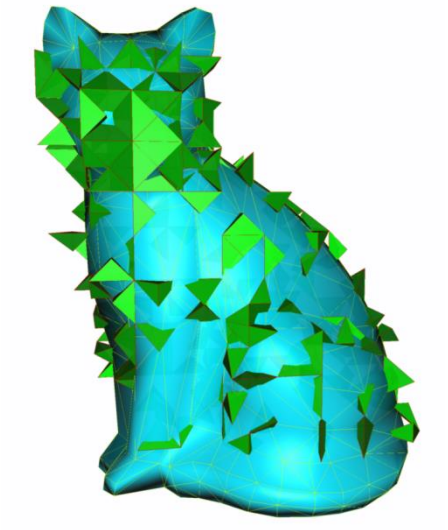

Fig. 5 : sélection des tétraèdres contenant un point de la surface.

Lorsque l'on récupère un élément de cntTH4 (mêmes notations que dans l'exercice précédent), l'ordre des 4 nœuds (indice faisant référence à un point dans le tableau de coordonnées) suit celui de la figure 7.

Pour chaque point du chat, on va tester l'inclusion de ce point dans tous les tétraèdres[1]. Il y aura donc une double-boucle à mettre en place.

Test d'inclusion d'un point dans un tétraèdre : Un tétraèdre ayant 4 faces planes, le test se ramène à vérifier que le point se situe « en dessous » des 4 plans supportant les faces en considérant les 4 normales pointant vers l'extérieur de la cellule.

La normale à un plan se calcule comme le produit vectoriel de 2 vecteurs non-colinéaires de ce plan (cf. Fig. 6).

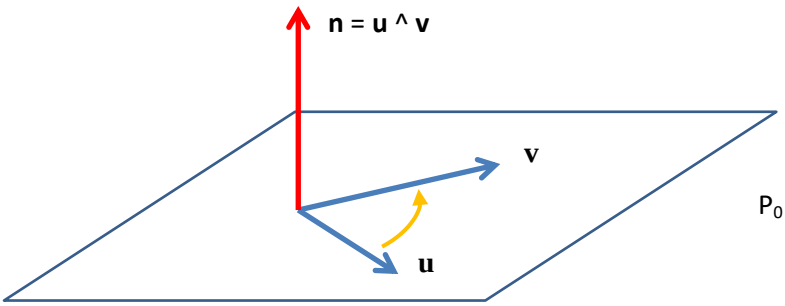

Fig. 6 : Calcul de la normale à un plan

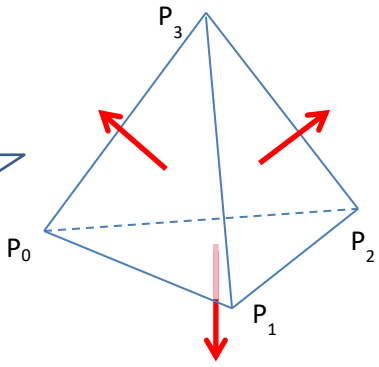

Fig. 7 : Arrangement des points d'une cellule

1) Définir une fonction membre de la classe Triangle permettant de calculer sa normale n :

```
void normal(const Array<double>& crd, double* n)
```

Vous utiliserez le code suivant pour calculer le produit vectoriel de 2 vecteurs :

```
// z = x ^ y
void crossProduct(const double* x, const double* y, double* z) {
   z[0] = x[1]*y[2] - x[2]*y[1];
   z[1] = x[2]*y[0] - x[0]*y[2];
   z[2] = x[0]*y[1] - x[1]*y[0];
}
```

2) Créer un fichier exo17.cpp, y définir une fonction main et la fonction permettant le test d'inclusion dont la signature est :

```
bool pt_is_inside(const double* Pi, const int* elt, const Array<double>& crd)
```

où Pi est le point à tester, elt une cellule et crd le tableau des coordonnées.

Note : La fonction retourne true si le point est dans la cellule, false sinon.

3) Test et validation

Ecrire dans la main la double boucle permettant de tester tous les points de la surface du chat avec toutes les cellules.

Notes :

[1] Cette façon de faire, dite « méthode de la force brute » est loin d'être efficace, mais fonctionne. Le maillage volumique étant petit, on peut se la permettre dans cet exemple didactique.

- on créera avant la boucle un vecteur de booléen de la taille du nombre de tétraèdres et initialisé à false. Puis, on mettra à jour ce tableau dans la boucle en mettant à true le tableau pour chaque cellule qui contient un point du chat.
- En sortie de boucle, il faut alors de créer un tableau de tétraèdres que l'on remplit avec les cellules marquées.
- Générer le fichier « selec.mesh » contenant la sélection.

Vérifiez que l'on obtient bien la sélection de la figure 5 en ouvrant le fichier selec.mesh avec medit. Il doit y avoir XXX tétraèdres.