Travaux pratiques
2 Bibliothèque Geometry

Dans ce TP, vous allez continuer la création du logiciel de visualisation d'images 3D. Cette fois, vous allez créer une bibliothèque de calculs pour les objets géométriques en C++. Vous devez créer au moins un fichier en-tête (ex. `libgeometry.h`) contenant l'interface de votre bibliothèque. Votre bibliothèque peut aussi contenir d'autres fichier. Toutes les classes et fonctions de la bibliothèque doivent être crées dans un espace de nom (ex. `libgeometry`). Vous devez aussi penser à créer des routines de test pour votre bibliothèque. La bibliothèque doit au moins définir les classes, méthodes, fonctions et constantes décrits ci-dessous.

Classe Quaternion : Définie un quaternion dont les composants sont de type T, où T est un paramètre de la classe. La classe doit au moins définir les méthodes suivantes

- `Quaternion()` : construit un quaternion à partir des arguments. Le(s) constructeur(s) doit être au moins capable de construire un quaternion dont l'angle de rotation en dégrées et l'axe (classe Direction ci-dessous) sont passés en argument.
- `conjugate()` : retourne le conjugué du quaternion.
- `norm()` : retourne la norme d'un quaternion.
- `im()` : retourne la partie imaginaire du quaternion.
- `inverse()` : retourne l'inverse du quaternion.
- `re()` : retourne la partie réelle du quaternion.
- `to_norm()` : retourne une copie du quaternion normalisée.
- opérateur + :
  - addition de scalaire et quaternion.
  - addition de quaternion et scalaire.
  - addition de deux quaternions.
- opérateur += : addition et affectation.
- opérateur - :
  - négation
  - différence de scalaire et quaternion.
  - différence de quaternion et scalaire.
  - différence de deux quaternions.
- opérateur -= différence et affectation.
- opérateur * :
  - multiplication de scalaire et quaternion.
  - multiplication de quaternion et scalaire.
  - multiplication de deux quaternions.
- opérateur *= : multiplication et affectation.
- opérateur << : surcharge l'opérateur pour les outputs.

Classe Transformation : Définie une transformation (c.-à-d., une rotation, mise à l'échèle et translation). La classe doit au moins définir les méthodes suivantes :

- `Transformation()` : construit une transformation selon les arguments passés. Le(s) constructeur(s) doit être au moins capable de créer :
  - une rotation définie par le quaternion passé en argument.
  - une rotation dont l'axe (classe Direction ci-dessous) et l'angle en dégrée sont passés en arguments.
  - une translation dont les valeurs de chaque axe sont passés en argument.
  - une mise en échèle dont les valeurs de chaque axe sont passés en argument.
- `concat()` : retourne la concaténation de deux transformations.
- `to_quat()` : retourne le quaternion correspondant à une rotation.
- `transform()` :
  - retourne un nouveau point qui correspond à la transformation du point passée en argument.
  - retourne une nouvelle direction qui correspond à la transformation de la direction passée en argument.

— retourne une nouvelle sphère (classe Sphere) qui correspond à la transformation passée en argument.
— opérateur << : surcharge l'opérateur pour les outputs.

Classe Point : Définie un point dans un espace de N dimensions et où chaque composant est de type T. T et N sont des paramètres de la classe. la classe doit au moins définir les méthodes suivantes (évidement, pas besoin de redéfinir des méthodes qui sont éventuellement, hérités d'une autre classe déjà construite) :

— `at()` : retourne la i-ème coordonnée (i passé en argument) du point.
lève une exception en cas d'erreur d'indice.
— `behind()` : retourne `true` si le point est derrière le plan passé en argument (voir classe plane ci-dessous).
— `is_null()` : retourne `true` si le point contient une valeur invalide et `false` sinon.
notamment, si le point possède des valeurs `nan`.
— `length_to()` : retourne une direction (voire classe direction ci-dessous) qui représente le vecteur entre le point et un autre point passé en argument.
— `outside()` : retourne `true` si le point est à l'intérieur d'une sphère passée en argument (voir classe sphere ci-dessous).
— `rotate()` : retourne un nouveau point qui correspond à la rotation du point par le quaternion passé en argument.
— opérateur << : surcharge l'opérateur pour les outputs.
— opérateur `[]` : adresse la i-ème coordonnée (i passé en argument) du point. Doit également permettre l'affectation. Ne vérifie pas si i est valide.

Classe Direction : Définie une direction dans un espace de N dimensions, où N est un paramètre de la classe. Les coordonnées de la direction sont de type T, où T est aussi un paramètre de la classe. La classe doit au moins définir les méthodes suivantes (évidement, pas besoin de redéfinir des méthodes qui sont éventuellement, hérités d'une autre classe déjà construite) :

— `at()` : retourne le i-ème composant (i passé en argument) de la direction.
Lève une exception en cas d'erreur d'indice.
— `is_null()` : retourne `true` si la direction contient une valeur invalide et `false` sinon. Notamment, si la direction possède des valeurs `nan`.
— `is_unit()` : retourne `true` si la direction est unitaire et `false` sinon.
— `norm()` : retourne la norme de la direction.
— opérateur << : surcharge l'opérateur pour les outputs.
— opérateur `[]` : adresse le i-ème composant (i passé en argument) de la direction. Doit également permettre l'affectation. Ne vérifie pas si i est valide.

Classe LineSegment : Définie un segment de droite. La classe doit au moins définir les méthodes suivantes :

— `get_begin()` : retourne le point de départ du segment.
— `get_end()` : retourne le point de fin du segment.
— `inter_coef()` : retourne le coefficient d'intersection entre le segment et un plan passé en argument (classe Plane ci-dessous). Le coefficient d'intersection détermine si la droite dont le segment appartient se trouve sur le plan, devant ou derrière celui-ci.
— `inter()` : retourne le point d'intersection entre le segment et un plan passé en argument (classe Plan ci-dessous).
— `is_null()` : retourne `true` si le segment contient une valeur invalide et `false` sinon. Notamment, si le segment contient des valeurs `nan`.
— opérateur << : surcharge l'opérateur pour les outputs.

Classe Plane : Définie un plan. La classe doit au moins définir les méthodes suivantes :

— `is_null()` : retourne `true` si le segment contient une valeur invalide et `false` sinon. Notamment, si le segment contient des valeurs `nan`.
— opérateur << : surcharge l'opérateur pour les outputs.

Classe Sphere : Définie une sphère. La classe doit au moins définir les méthodes suivantes :

— `behind()` : retourne `true` si la sphère est derrière le plan passé en argument.

— `is_null()` : retourne `true` si le segment contient une valeur invalide et `false` sinon. Notamment, si le segment contient des valeurs `nan`.
— opérateur << : surcharge l'opérateur pour les outputs.

Classe Rectangle : Définie un rectangle. La classe doit au moins définir les méthodes suivantes :
— `is_null()` : retourne `true` si le segment contient une valeur invalide et `false` sinon. Notamment, si le segment contient des valeurs `nan`.
— opérateur << : surcharge l'opérateur pour les outputs.

Classe Triangle : Définie un triangle. La classe doit au moins définir les méthodes suivantes :
— `area()` : retourne l'aire du triangle.
— `is_null()` : retourne `true` si le segment contient une valeur invalide et `false` sinon. Notamment, si le segment contient des valeurs `nan`.
— `get_p0()` : retourne le sommet (point) 0.
— `get_p1()` : retourne le sommet (point) 1.
— `get_p2()` : retourne le sommet (point) 2.
— opérateur << : surcharge l'opérateur pour les outputs.