TD/TP 6

Classes, associations

Objectifs, méthodes

On a vu lors du cours 6 que les associations qu'on représente entre classes sur les diagrammes UML se traduisent dans le code C++ le plus souvent (mais pas tout le temps) par des pointeurs. Nous allons sur les 2 séances (3H) de ce TP implémenter **une partie** du cahier des charges du fil conducteur Maillage. Je vous renvoie donc aux énoncés assez détaillés du TD/TP 1 et 2 et au travail de conception que vous aviez alors réalisé (voir Maillage 2D triangulé du TD/TP 1, pages 5-6).

Ce TD/TP sera moins détaillé dans la démarche à suivre que les TD/TPs précédents. Quelques conseils seront donnés mais pas sous forme d'une séquence de développement précise. **Les objectifs à atteindre seront donnés par rapport aux résultats concrets attendus lors de l'exécution du programme, sous forme de captures écran/console.** Un outil d'injection de saisies (saisies simulées) vous permettra de tester votre application dans des situations complexes sans avoir à rentrer vous même 100 fois les mêmes séquences au clavier.
Cet outil ne sera utile avec les fichiers de tests fournis que si votre programme respecte scrupuleusement l'ordre des saisies et les numéros d'actions du menu.

Comme le thème du fil conducteur est la génération de maillages 2D triangulés nous allons continuer d'utiliser la classe fournie Svgfile et la génération de fichiers en format vectoriel avec un navigateur pour visualiser le résultat. Une version légèrement complétée de Svgfile est donnée, elle permet maintenant de faire les triangles.

Outre la classe technique Svgfile que vous devrez apprendre à utiliser mais pas nécessairement comprendre (il n'est pas nécessaire d'entrer dans les détails de svgfile.cpp), le projet de départ fourni comportera une classe **Couleur** (rouge vert bleu), une classe **Coords** (qu'on a déjà étudiée !), et une struct **StyleDessin** qui regroupera les options de dessin pour le rendu final. Cette dernière ne fait pas partie du modèle, elle pourra servir seulement de paramètre aux méthodes de dessin. **Aucune de ces 3 classes initiales fournies n'est obligatoire**. En supposant que vous les utilisiez alors le volume de code à produire pour aller au bout du TP de 3H est d'environ 300 lignes de code, ce qui correspond à 50 lignes de l'heure pour chaque membre d'un **binôme**. C'est approximativement la production attendue pour du code de difficulté « moyenne ».

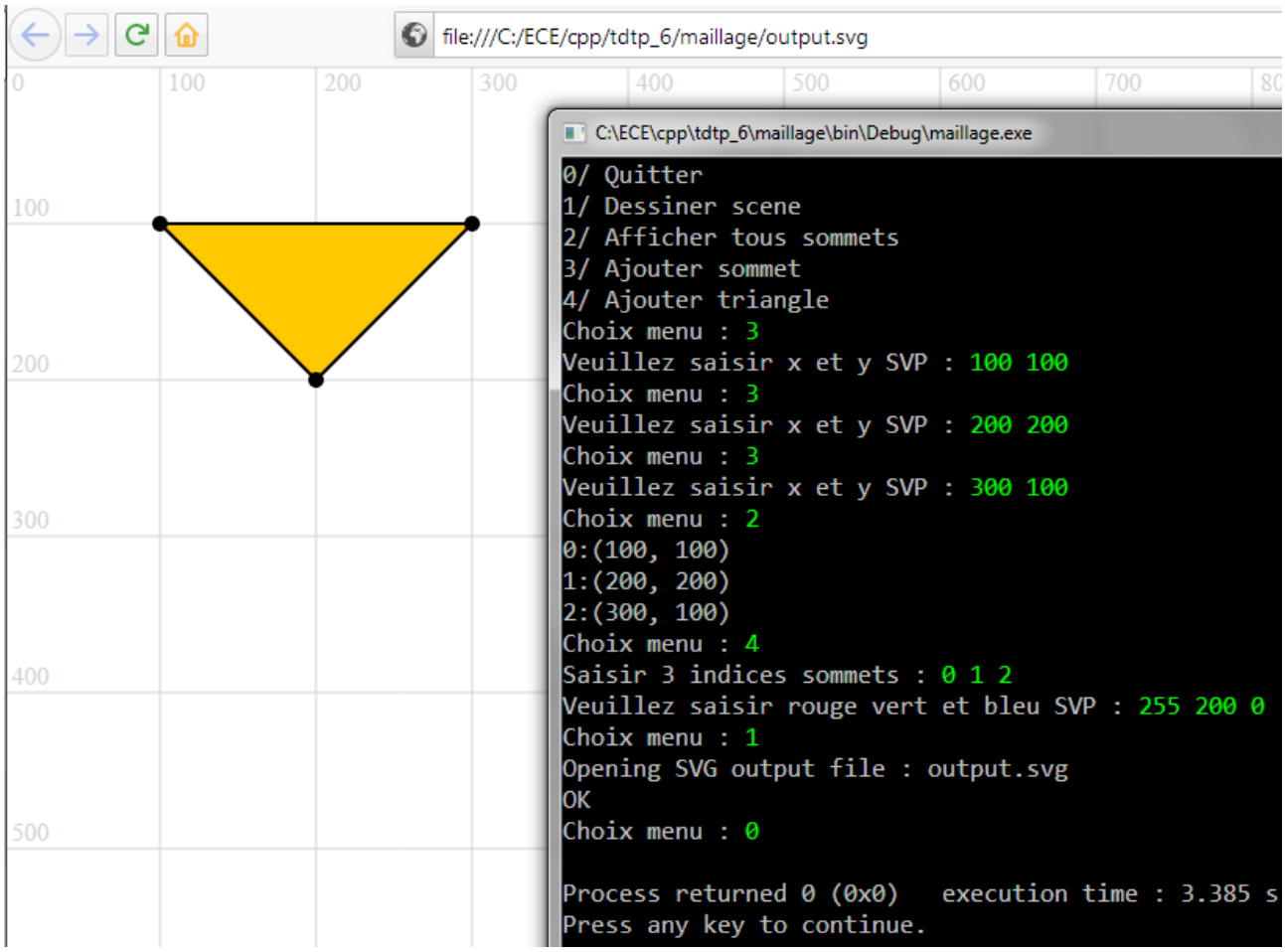
Les concepts du cours couverts par ce TP seront
- **`implémentation d'un modèle UML`**
- **`associations`**
- **`types valeurs, types entités`**
- **`classe gestionnaire de ressource`**
- **`new/delete`** pour ajouter/enlever dynamiquement des objets au système

1. Télécharger le projet de départ, le tester, comprendre main.cpp

https://fercoq.bitbucket.io/cpp/tdtp/tdtp6/maillage_exo.zip

2. Application interactive maillage avec ajouts sommets/triangles

A partir du code fourni, mettre en place la machinerie orientée objet nécessaire pour obtenir le résultat suivant : à droite le déroulement de la session interactive (en vert les saisies de l'utilisateur) et à gauche visualisation du fichier output.svg obtenu. ***Respectez scrupuleusements les numéros d'action du menu : ceci sera utile pour la suite.*** Quelques indications suivent...



Même si le CDC de cet exercice n'est pas aussi complexe que celui proposé initialement il est conseillé de se replonger dans les diagrammes d'objets et les diagrammes de classes que vous aviez fait à l'époque avant d'attaquer le clavier : c'est bien à ça que sert la phase initiale de conception !

Si vous n'aviez pas envisagé sur vos diagrammes de classe de l'époque une classe Maillage je vous suggère d'en envisager une maintenant. La classe maillage servira de classe englobant la gestion des Sommets et Triangles alloués dynamiquement. Je vous recommande de **bien séparer la classe Coords (fournie) et la classe Sommet** : la 1ère est de type valeur alors que la 2ème est de type entité. Il suffit d'avoir un attribut Coords dans la classe Sommet. Cet attribut peut s'appeler m_position par exemple.Cette distinction oblige à faire plus de code de liaison mais fournit une **meilleure abstraction** : identifiez d'une part toutes les classes de type valeur, toutes les classes de type entité d'autre part. **Les classes de type entité de cette application ne devraient même pas savoir que les coordonnées sont en 2D, elles pourraient être en 3D (x, y, z) ça ne changerait rien pour elles.** Sauf au niveau des méthodes de dessin où on n'a pas trop le choix puisque les méthodes d'ajout de primitives de la classe Svgfile nous obligent à donner séparément un x et un y. **Anticipez que les sommets devront pouvoir être sélectionnés.**

Il est évidemment souhaitable que les entrées soient blindées, au moins en intervalle (on a vu que blinder en type est compliqué!). Le namespace util:: fourni propose une nouvelle fonction **`videCin`** qui permet de vider le tampon de lecture du clavier si c'est nécessaire pour refaire proprement une nouvelle saisie. Voir un exemple d'utilisation dans la fonction saisirCanal, couleur.cpp, ligne 28. Blinder c'est indispensable mais ne passez pas trop de temps là dessus : essayez d'arriver déjà au résultat quand les données saisies sont correctes, et **en utilisant des approches orientées objets satisfaisantes (voir dernière page : Notes pour aller plus loin)**

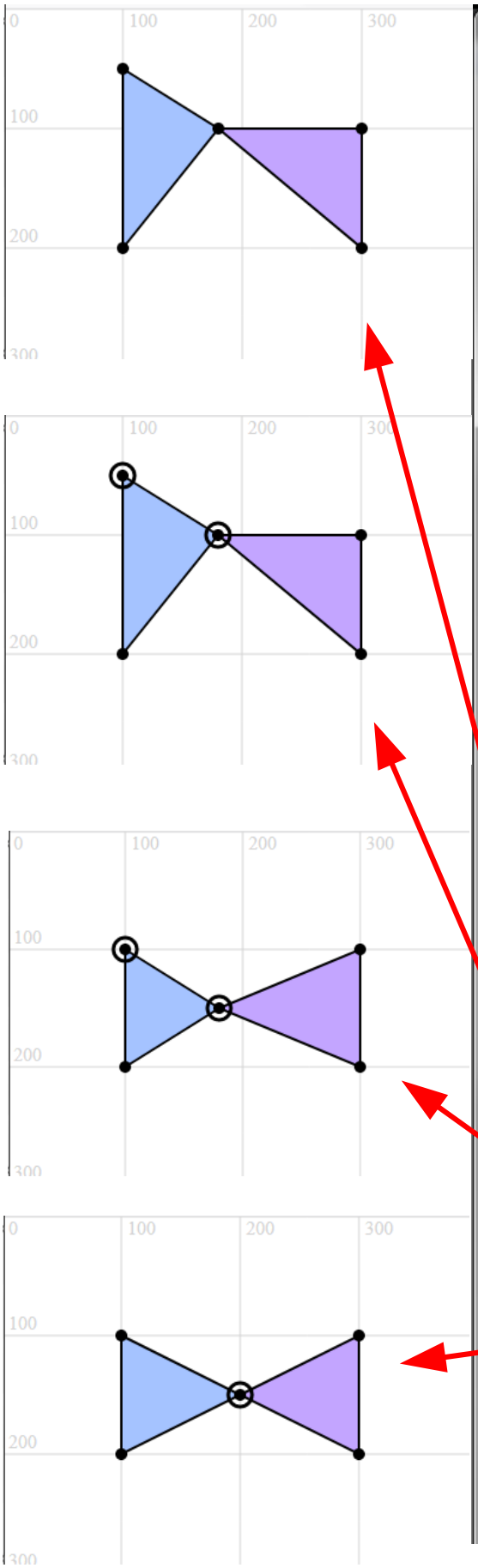
Pour tester votre application sans avoir à rentrer manuellement au clavier des données de test, le code fourni propose 2 fonctions supplémentaires dans le namespace util:: startAutoCin et stopAutoCin. Il n'est pas nécessaire de comprendre leur implémentation dans util.cpp pour pouvoir les utiliser (heureusement!). **Dès que vous aurez mis en place votre menu vous pourrez injecter une simulation d'entrées au clavier à partir d'un fichier (voir fichier test1.txt[1]) :**

```
util::startAutoCin("test1.txt", 50);

  int choix;
  do
  {
      std::cout << "Choix menu : ";
      std::cin >> choix;
      switch(choix)
      {
          case 0:
              break;

          case 1:
              …
              break;
          case 2:
              …
              break;
          … etc …

          default:
              std::cout << "Anomalie choix menu" << std::endl;
              break;
      }
  }
  while ( choix!=0 );

  util::stopAutoCin();
```

Vous devriez alors voir défiler automatiquement les commandes clavier comme si vous tapiez vous même. Sur Windows les entrées clavier simulées seront en vert. Pas de coloration pour les autres OS pour l'instant. Vous pouvez changer le rythme de défilement : voir commentaires dans util.h. Vous pouvez utiliser un autre fichier test : si test1.txt fonctionne correctement et donne bien le résultat attendu (page 2) alors vous pouvez ouvrir à la place de test1.txt **cocotte.txt** et obtenir un résultat graphique plus complexe. **Si vous êtes parmi les 1ers à obtenir la cocotte, faites valoriser par votre chargé de TP** (qui modulera le bonus en fonction de la qualité du code : vite et bien).

1 Pour Linux et macOS utiliser les versions avec linux dans le nom, test1_linux.txt etc ...
Pour macOS copier ces fichiers dans le répertoire d'exécution (Products...)

3. *Application interactive maillage avec sélection / translation*



```
C:\ECE\cpp\tdtp_6\maillage\bin\Debug\maillage.exe
0/ Quitter
1/ Dessiner scene
2/ Afficher tous sommets
3/ Ajouter sommet
4/ Ajouter triangle
5/ Selectionner sommet
6/ Deselectionner sommet
7/ Translater selection
Choix menu : 3
Veuillez saisir x et y SVP : 100 50
Choix menu : 3
Veuillez saisir x et y SVP : 180 100
Choix menu : 3
Veuillez saisir x et y SVP : 100 200
Choix menu : 3
Veuillez saisir x et y SVP : 300 100
Choix menu : 3
Veuillez saisir x et y SVP : 300 200
Choix menu : 4
Saisir 3 indices sommets : 0 1 2
Veuillez saisir rouge vert et bleu SVP : 165 195 255
Choix menu : 4
Saisir 3 indices sommets : 1 3 4
Veuillez saisir rouge vert et bleu SVP : 195 165 255
Choix menu : 1
Opening SVG output file : output.svg
OK
Choix menu : 5
Saisir indice sommet a selectionner : 0
Choix menu : 5
Saisir indice sommet a selectionner : 1
Choix menu : 1
Opening SVG output file : output.svg
OK
Choix menu : 7
Vecteur translation : Veuillez saisir x et y SVP : 0 50
Choix menu : 1
Opening SVG output file : output.svg
OK
Choix menu : 6
Saisir indice sommet a deselectionner : 0
Choix menu : 7
Vecteur translation : Veuillez saisir x et y SVP : 20 0
Choix menu : 1
Opening SVG output file : output.svg
OK
Choix menu : 0

Process returned 0 (0x0)   execution time : 7.925 s
```

Cette session correspond au fichier test2.txt

Notes pour aller plus loin...

(si vous ne comprenez rien ici ne vous acharnez pas : faites à votre manière)

Pour rappel : il est en général maladroit de manipuler les objets d'un certain niveau comme des marionnettes depuis le niveau supérieur en réglant les détails. Par exemple ce n'est pas le rôle d'une méthode de la classe Maillage de saisir les données d'un Triangle, logiquement la saisie des données d'un triangle devrait se faire dans une méthode de la classe Triangle. Ceci peut poser problème si le composant a une vision trop locale et nécessite de connaître le contexte. Sur cet exemple : la saisie des sommets d'un triangle se fait par indice mais l'objet Triangle va (en principe) stocker des pointeurs sur 3 Sommets. Il faut passer d'un indice sommet au pointeur correspondant. À son niveau la classe Triangle ne dispose pas de cette information. Une solution normale (classique) à ce problème est alors de **passer le contexte en paramètre** par exemple de passer la classe composite Maillage en paramètre de l'appel à la méthode de saisie d'un Triangle, ce qui permet en retour à la méthode de saisie de Triangle d'appeler une méthode de la classe composite Maillage qui trouve l'adresse d'un sommet à partir de son indice.

maillage.cpp

```
void Maillage::ajouterTriangle()
{
    Triangle* t=new Triangle;
    t->saisir(*this);
    m_triangles.push_back(t);
}

Sommet* Maillage::trouverSommet() const
{
    size_t idx;

    std::cin >> idx;
    while (idx>= m_sommets.size())
    {
        std::cout << "Mauvais indice
sommet, recommencer : ";
        util::videCin();
        std::cin >> idx;
    }

    return m_sommets[idx];
}
```

triangle.cpp

```
void Triangle::saisir(const Maillage&
maillage)
{
    std::cout << "Saisir 3 indices
sommets : ";

    for (size_t i=0; i<3; ++i)
        m_sommet[i] =
maillage.trouverSommet();

    m_couleur.saisir();
}
```

Ceci implique que la classe Triangle connaisse la classe Maillage pour pouvoir déclarer un paramètre de type Maillage. Il faudrait faire #include "maillage.h" dans triangle.h mais comme on a déjà #include "triangle.h" dans maillage.h et que les inclusions circulaires sont interdites on va utiliser ce qu'on appelle une déclaration « en avant » (***forward declaration***) :

```
/// Fichier triangle.h
… PAS DE INCLUDE DE maillage.h

/// Forward declaration de la class Maillage
class Maillage;

class Triangle
{
    public :
        void saisir(const Maillage& maillage); // OK, Maillage est une classe !
```