TD/TP 4

C++ pratique 2
Les techniques courantes

Objectifs, méthodes

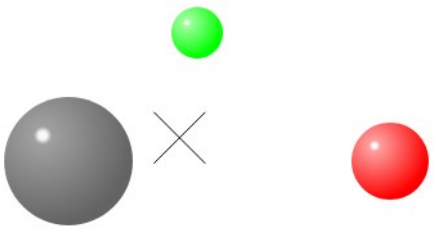
Encore du code ! En cours, plus de techniques du C++ ont été présentées que nous utiliserons lors de ce TP. Dans la perspective de pouvoir reprendre le fil conducteur "maillage triangulé" dès que nous saurons faire nos propres classes (TD/TP 6) on va commencer à générer des graphismes sous forme de fichiers en format vectoriel. L'image sera affichée dans un navigateur, il faudra recharger pour voir une nouvelle image, ce qui présente l'inconvénient de ne pas être interactif mais l'avantage d'être portable et de ne pas avoir à configurer une compilation avec une bibliothèque etc... Un code complet et portable est fourni pour faciliter la génération de ces fichiers images.

Les concepts du cours couverts par ce TP seront

- **`struct simple`** comme initiation aux objets (sans méthodes, sans encapsulation...)
- **`références et pointeurs`** pour transmettre des paramètres et stocker des objets alloués
- **`const`** pour éviter les déboires
- **`paramètres par défaut`** sont utilisés par la classe graphique fournie
- **`surcharge de fonction`** sera applicable à des fonction que vous écrirez
- **`surcharge d'opérateur`** un exemple est prévu
- **`new/delete`** pour ajouter/enlever dynamiquement des objets d'une collection

Cap sur les barycentres !

Les exos de ce TD/TP sont axés sur la réalisation d'une application de gestion d'objets sphériques massifs dans le plan (2D) et du calcul de leur barycentre : moyennes pondérées de leurs abscisses et ordonnées, autrement appelé **centre de gravité**. On voudra en particulier pouvoir illustrer graphiquement le centre de gravité de systèmes de corps astronomiques (planètes, satellites, exoplanètes, étoiles à neutrons...) réels ou imaginaires.



L'aspect graphique est fourni. Le travail qui vous est demandé sera de mettre en place une struct Sphere regroupant des attributs de position x et y, de masse, de rayon et de couleur. Ces 2 derniers attributs ne joueront aucun rôle dans les calculs de barycentre mais seront nécessaires pour donner un certain volume à nos « masses ponctuelles ». Vous associerez à cette struct un certain nombre de sous-programmes utiles à l'exploration de la thématique, en particulier une opération de somme qui fera la somme pondérée des objets qui lui sont confiés et qui donnera le barycentre sous forme d'une Sphere (les coordonnées du centre de gravité seront les coordonnées de cette Sphere résultat de la somme). Figure ci-dessus : 3 corps massifs et leur centre de gravité.

Enfin vous proposerez un menu interactif en console permettant d'ajouter de nouveaux objets (allocation dynamique avec new) à une collection (vecteur) de pointeurs sur objets. Il sera possible à tout moment de refaire un nouveau fichier graphique pour illustrer la progression de la collection. Enfin il sera possible d'enlever des objets de la collection (libération avec delete).

1. Télécharger et compiler le projet de départ, découvrir le format SVG

Téléchargez ce projet de départ qui contient la « bibliothèque » de dessin vectoriel :

https://fercoq.bitbucket.io/cpp/tdtp/tdtp4/barycentres_exo.zip

Dézippez, ouvrez le projet (.cbp) si vous utilisez Code::Blocks ou faites un projet qui inclut les 3 fichiers sources sinon, compilez, exécutez. Vous devez obtenir dans le répertoire d'exécution [1] (*macOS : voir note de bas de page*) un nouveau fichier **output.svg** qui a été généré par l'exécution. Vous pouvez visualiser ce graphisme en l'ouvrant dans un navigateur : tous les navigateurs récents (Firefox, Chrome, Edge…) sont compatibles avec le format SVG. Selon votre système il suffit de double-cliquer sur le fichier output.svg pour automatiquement l'ouvrir dans une application compatible (par exemple Firefox) ou si ça ne donne rien aller dans votre navigateur et ouvrir avec menu Fichier -> Ouvrir un fichier...

En principe vous devriez voir quelques graphismes. Vous pouvez sans-doute zoomer/dé-zoomer avec Ctrl+molette de la souris ou zoom à 2 doigts sur périphérique tactile. A chaque fois que nous modifierons le programme, **une nouvelle exécution écrasera le fichier output.svg précédent mais le navigateur ne va pas automatiquement mettre à jour**. Cependant il n'est **pas nécessaire de fermer l'onglet et de ré-ouvrir le fichier output.svg à chaque fois**. Laissez le fichier ouvert dans le navigateur et **recharger** (*reload*, trouvez le raccourci clavier, Ctrl-R sous Firefox...)

Testez : essayer de voir la relation entre le code de test et ce qui est dessiné, ça ne devrait pas être trop compliqué à comprendre, trouver un paramètre, le modifier, relancer le programme, basculer sur l'onglet ouvert du dessin SVG et faire reload, en principe la modification est visible.

Pour info, le format SVG est un format normalisé de dessin vectoriel, c'est à dire des graphismes qui ne sont pas constitués de pixels mais de primitives graphiques avec une résolution « infinie ». Essayez de zoomer à l'extrême sur un disque à l'écran, on ne voit jamais les pixels ! Ce format présente l'avantage d'être lisible humainement, ce n'est pas un format binaire. Vous pouvez lire et même éditer un fichier SVG avec un éditeur technique. Par exemple sous Code::Blocks aller dans File -> Open... et ouvrir output.svg. Vous pouvez améliorer la lisibilité en indiquant à Code::Blocks que c'est un format de type XML : Edit -> Highlight mode -> XML. **On ne laissera pas le fichier SVG ouvert dans Code::Blocks** car il est modifié à chaque nouvelle exécution du projet et Code::Blocks nous demandera une confirmation de mise à jour à chaque fois. Cependant en cas de problème, par exemple l'image attendue ne s'affiche par correctement dans le navigateur, on pourra consulter ce fichier intermédiaire.

Pour une brève introduction au format SVG vous pouvez jeter un œil à la page wikipedia mais il n'est pas nécessaire d'en connaître les détails car la classe Svgfile fournit les primitives dont nous auront besoin. Il n'est pas non plus nécessaire d'aller éplucher les 150 lignes non commentées (faute de temps) de cette classe, pour l'utiliser inspirez-vous des exemples. Les noms des couleurs SVG possibles sont sur cette page. Pour représenter des sphères on préférera la version « en relief » avec les seules couleurs possible : **redball**, **greenball**, **blueball**, **yellowball**, **greyball**[2]

On utilise un repère cartésien infographique (ordonnées vers le bas, origine en haut à gauche) : vous pouvez voir ce repère en dé-commentant la ligne `svgout.addGrid();` *Tester. On peut vérifier ligne 33 de svgfile.h que la méthode addGrid utilise des paramètres par défaut, essayer d'appeler addGrid avec 1, 2 ou 3 paramètres (grille plus fine, couleur différente...)*

1 Pour Code::Blocks c'est le répertoire de projet. Avec d'autres systèmes ça peut être ailleurs. Pour macOS avec Xcode c'est remarquablement difficile à trouver, Google « xcode console execution directory » -> ici ou là

2 Pour les curieux : voir svgBallGradients à la fin de svgfile.cpp (mauvaise approche, patch en urgence)

2. Mise en place d'une struct Sphere, déclaration d'objets, affichages

En toute rigueur on devrait plutôt l'appeler SphereMaterielle mais on va s'en tenir à un identifiant court et expressif : chaque objet de type Sphere sera caractérisé par une abscisse **`x`**, une ordonnée **`y`**, une masse **`masse`**, un rayon **`rayon`**, et une couleur **`couleur`**, tous ces attributs étant des nombres à virgule double précision, type **`double`**, sauf le dernier qui est une **`std::string`** car les couleurs seront données sous forme textuelle. La suite de l'énoncé indiquera les données dans cet ordre : **`x, y, masse, rayon, couleur`**

Vous pouvez ajouter un fichier sphere.h et sphere.cpp à votre projet pour clairement séparer la déclaration de la struct et les prototypes des sous-programmes associés (sphere.h) et l'implémentation des sous-programmes (sphere.cpp) du code utilisateur (main.cpp pour l'instant). Ou vous pouvez faire ça plus tard et procéder uniquement dans main.cpp dans un 1er temps, si vous avez du mal à suivre le rythme.

Neutraliser le test donné initialement en enlevant l'appel à `svgTest`.

Coder la struct **`Sphere`** *(chapitre A du cours 4)*

Tester la struct en déclarant 2 objets terre et lune (pas de majuscules aux objets!) avec des données initiales et en affichant ***à la console*** *leurs attributs, directement dans le main :*
La terre aura les données initiales { 200, 400, 6, 63, "blueball" }
La lune aura les données initiales { 584, 400, 2.7, 27, "greyball"}

Mettre en place un sous-programme **`afficher`** *qui prend en paramètre un objet Sphere et affiche* ***à la console*** *la valeur de ses attributs. Utiliser ce sous-programme en l'appelant 2 fois depuis le main pour la terre et pour la lune. Tester.*

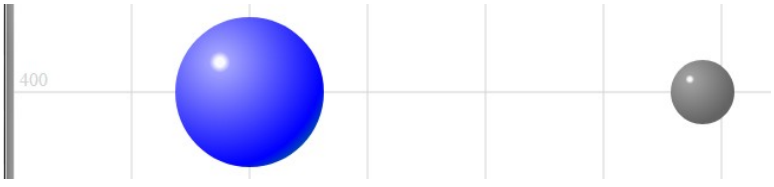
Avez vous bien utilisé un passage par référence pour éviter une copie inutile ? Avez vous bien utilisé la qualification const pour indiquer que les données référencées n'avaient pas vocation à être modifiées durant l'appel ? **Si vous êtes en galère avec le prototype à écrire ou que vous voulez vérifier voici un corrigé, sélectionner et copier le texte invisible entre chevrons ligne suivante**
<<< >>> (le **nom** du paramètre est **arbitraire**)

3. Dessin vectoriel des sphères

Comme dans le code donné en exemple initialement, créez un objet « dessin vectoriel » :

```
/// Sortie graphique avec repère
Svgfile svgout;
svgout.addGrid();
```

Puis en voyant comment s'utilisent les paramètres de `svgout.addDisk(...)` *ajoutez au dessin 2 disques : avec les paramètres de la terre et avec les paramètres de la lune vous devriez arriver à générer un graphisme qui donne ça (corrigé page suivante si vous bloquez)*



Corrigé <<<

>>>

Pour éviter d'avoir à répéter/substituer du code avec des objets différents on veut immédiatement un sous programme **`dessiner`** : *coder un sous-programme* **`dessiner`** *qui prend en paramètre une référence à un objet de type* **`Svgfile`** *et une référence à un objet de type* **`Sphere`** *et qui dessine (addDisk) la sphère dans le fichier SVG.* L'objet `Svgfile` reçu va-t-il être modifié (est-ce qu'il reçoit des informations) ? L'objet `Sphere` reçu va-t-il être modifié (est-ce qu'il reçoit des informations) ? En conclure quel(s) paramètre(s) doit éventuellement être qualifié de `const` ? Tester : on doit obtenir le même résultat que précédemment mais le main ne contient plus aucun appel direct à `svgout.addDisk` ...

Corrigé du prototype <<< >>>
(les **noms** des paramètres sont **arbitraires**)

4. Fonction de calcul de barycentre

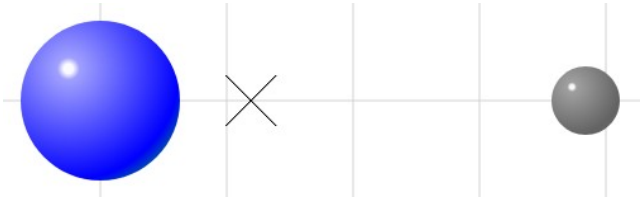
On va maintenant développer la fonction qui est au cœur de cette application : on va appeler **`sommer`** *une fonction qui reçoit 2 objets* `Sphere` *quelconques et qui renvoit un objet* `Sphere` *qui a comme position le centre de gravité des 2 objets reçus, comme masse la somme des 2 masses, et comme rayon 0 et couleur "black" (il semble difficile de faire une moyenne pondérée sur ces 2 critères*[3]*). Réfléchir au prototype d'une telle fonction compte tenu de son rôle.*

Corrigé du prototype <<< >>>
(les **noms** des paramètres sont **arbitraires**)

Implémenter la fonction. Pour rappel la formule d'un calcul de barycentre cg entre a et b est

$$x_{cg}=\frac{m_a x_a+m_b x_b}{m_a+m_b} \qquad y_{cg}=\frac{m_a y_a+m_b y_b}{m_a+m_b} \qquad m_{cg}=m_a+m_b$$

Tester avec le couple terre lune pour faire un objet `Sphere cgTerreLune`*. Afficher l'objet obtenu, est-il cohérent? Modifier le sous-programme dessiner de la question précédente : quand il reçoit un objet avec un rayon nul*[4] *il n'appelle pas* `svgout.addDisk(...)` *mais* `svgout.addCross(...)` *de telle sorte qu'en appelant dessiner avec l'objet cgTerreLune on obtient finalement le dessin vectoriel suivant* :



5. Fonction de calcul de barycentre à trois paramètres (FACULTATIF)

Comment feriez vous pour permettre une fonction **`sommer`** *à 3 paramètres ? Paramètre par défaut ? Version surchargée ? Et dans ce dernier cas comment ne pas recoder les calculs et se reporter sur l'existant en exploitant* **`sommer`** *à 2 paramètres ? Tester...*

3 On comprend que la situation n'est pas 100% satisfaisante : pour une partie des objets de type Sphere, ceux qu'on a obtenus par l'intermédiaire d'un appel à cette fonction de détermination de centre de gravité, il y a 2 attributs « neutralisés » ou « inutilisés ». On verra plus tard avec l'héritage comment résoudre ce problème.

4 Voir note ci-dessus !

6. Surcharge de l'opérateur +

On va goûter au doux plaisir de modifier un opérateur du langage en surchargeant operator+ : dans le code appelant on voudrait pouvoir écrire un truc qui claque comme

```
Sphere cgTerreLune = terre + lune;
```

au lieu d'un très banal appel de fonction à 2 paramètres

```
Sphere cgTerreLune = sommer(terre, lune);
```

En fait en regardant de près le *chapitre F du cours 4* on comprend que c'est exactement la même chose ! En particulier la surcharge (la version) de la fonction **`operator+`** adaptée à cette situation a exactement la même signature (même prototype) que la fonction **`sommer`** déjà écrite !

Dit autrement il suffit de dupliquer la fonction `sommer` de remplacer son nom par `operator+` et de remplacer `sommer(terre, lune)` par `terre+lune` dans le code appelant, et voilà. *Tester.* Ce genre de modification du langage est souvent qualifiée de sucre syntactique (syntactic sugar) : en effet ça n'apporte rien de fondamentalement nouveau en terme de mécanisme de programmation, c'est juste un jeu d'écriture. C'est intéressant quand les objets manipulés ont des propriétés algébriques cohérentes avec opérateurs surchargés, on pense par exemple aux nombres complexes, mais aussi aux nombres entiers en précision illimitée, aux vecteurs etc... Dans un autre registre c'est la possibilité plutôt naturelle d'interpréter + et += comme une concaténation dans le domaine des chaînes. Attention de ne pas abuser de la surcharge d'opérateur : le langage peut devenir illisible.

Ceci dit, on a obtenu notre operator+ pour pas cher en dupliquant sommer mais ça se payera plus tard... En effet on n'a pas respecté la maxime DRY : Don't Repeat Yourself. On ne doit pas copier-coller du code similaire (le moins possible). Parce que dès qu'il faudra « entretenir » un code cloné il faudra courir après les clones pour les patcher aussi : c'est long, source d'erreur, et en général inutile. Au fait, dans `sommer` vous avez bien pensé à vérifier que la somme des masses des 2 sphères est non nulle ? Non ? Parce qu'avec cette somme on fait une division... Et votre code trouvera fatalement un chercheur qui s'intéresse aux objets exotiques de masses négatives et qui voudra connaître le barycentre d'un système dont une somme partielle est de masse nulle. Vous pouvez lui offrir mieux qu'une division par 0 : le résultat étant de masse nulle il n'aura aucun poids dans les calculs ultérieurs à condition d'avoir évité la division par 0 : *ajouter un test dans `sommer` pour éviter les divisions par 0. Et modifier l'implémentation de `operator+` pour qu'il appelle la version mise à jour de `sommer` (plutôt que de patcher un code cloné).*

Au fait : les données pour le système Terre Lune étaient fictives, il n'y a que dans les films de space opera que les astres sont aussi gros relativement à leur distance. D'autre part la masse lunaire est d'environ un centième de la masse terrestre, le centre de gravité du système est dans la Terre. Essayez les données suivantes plus proche du réel (aux échelles près).

```
Sphere terre{200,      400, 6,    6.3, "blueball"};
Sphere lune {200 + 384, 400, 0.07, 1.7, "greyball"};
```

source NASA

7. Collection de Spheres : allocation dynamique

On veut pouvoir ajouter/enlever des sphères à une collection, de façon interactive. Les objets sphères sont à la fois suffisamment légers et isolés (il n'y a pas d'autres objets qui pointent sur un objet Sphere) pour pouvoir être traités par valeur : on pourrait gérer une collection de sphères sous forme d'un vecteur de Sphere std::vector<Sphere> directement, sans allocation dynamique, de la même façon qu'on peut gérer interactivement une collection d'entier std::vector<int>.

Mais quand on commencera à avoir des objets plus gros (en octets) et surtout qui sont référencés (pointés) par d'autres cette approche posera problème, par exemple les vecteurs grossissent automatiquement (push_back...) mais ça implique de bouger en mémoire les données stockées, une donnée pointée devient alors invalide (et on ne peut pas en général remonter vers les « objets pointants » pour les mettre à jour). Il devient rapidement nécessaire de faire de l'allocation dynamique et de stocker les collections non pas sous la forme « objets valeurs » mais sous la forme « pointeurs sur entités stables ».

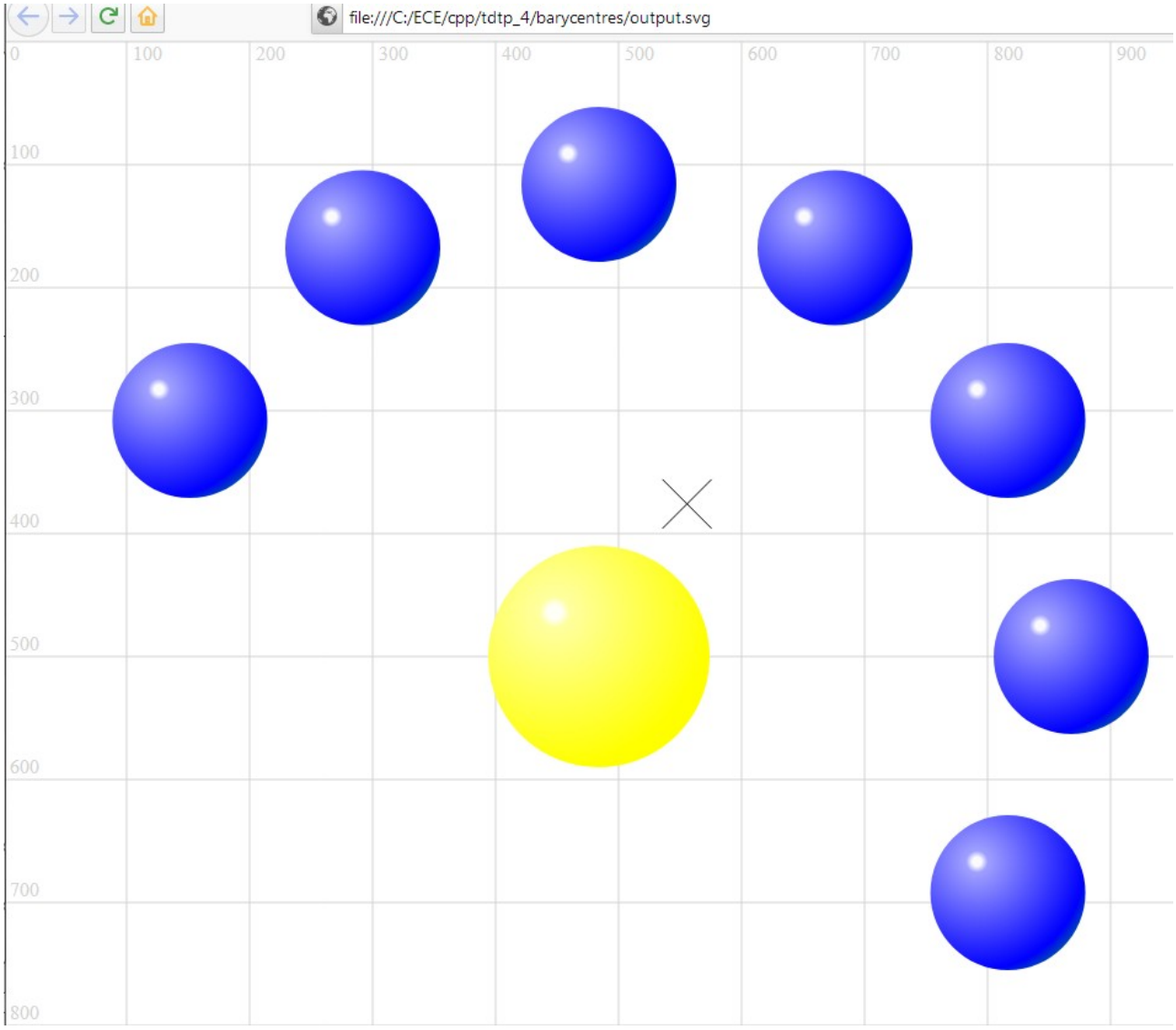
On aura donc un stockage de type std::vector<Sphere*>. Le but de cet exercice est de s'entraîner à traiter ce genre de collection. Le *slide 108 du cours 4* montre le principe général, avec des sous-programmes de gestion de la collection qui prendront en paramètre une **référence à un tableau de pointeurs sur les objets**. Les développeurs en Java et C# n'ont pas l'occasion d'apprécier ce genre de nuances. Les sous-programmes valables pour traiter seulement un ou deux objets pourront être **surchargés** pour accepter ce type de vecteur. On aura au final :

```
Sphere sommer(const Sphere& a, const Sphere& b);

Sphere sommer(const std::vector<Sphere*>& vec);

void dessiner(Svgfile& svgout, const Sphere& obj);

void dessiner(Svgfile& svgout, const std::vector<Sphere*>& vec);
```

Les versions qui prennent la collection en paramètre appelleront les versions qui en traitent un ou deux à la fois. On verra apparaître un appel à **`dessiner`** (un seul objet) dans le sous-programme **`dessiner`** (vecteur d'objets) mais il ne s'agit pas du tout d'une récurrence : ces deux sous-programmes ont le même nom mais il s'agit bien de 2 sous-programmes distincts.

L'utilisation d'un même nom pour une même opération portant sur des choses différentes offre l'avantage de ne pas encombrer la mémoire du client développeur avec des déclinaisons (dessinerCeci, dessinerCela) mais présente l'inconvénient d'une certaine opacité, voir parfois d'une confusion (dessiner quoi au fait ?). Plus important : noter que tous les paramètres passés par référence sont const à l'exception de l'objet fichier SVG qui a effectivement vocation à recevoir des informations pour réaliser la fonction de dessiner.

Mettre en place ces sous-programmes de gestion « collective »
Coder un main avec un menu interactif qui propose
- ajouter une sphère (saisir ses paramètres) **utilisation de new**
- dessiner la situation actuelle dans un nouveau fichier output.svg
  astuce : déclarer Svgfile localement dans un bloc {} à l'accolade fermante le fichier se ferme il est donc possible en basculant sur le navigateur et en faisant reload de voir la scène se construire progressivement sans quitter le programme en cours d'exécution
- enlever une sphère (saisir son numéro dans le vecteur ?) **utilisation de delete**
- quitter


file:///C:/ECE/cpp/tdtp_4/barycentres/output.svg
0
100
200
300
400
500
600
700
800
900