

# C++ PoP – Sections Electricité et Microtechnique

Printemps 2025 : *The great Linked-Crossing challenge*

© R. Boulic & collaborateurs

Pourrez-vous traverser l'arène avant la fin du compte à rebours ?

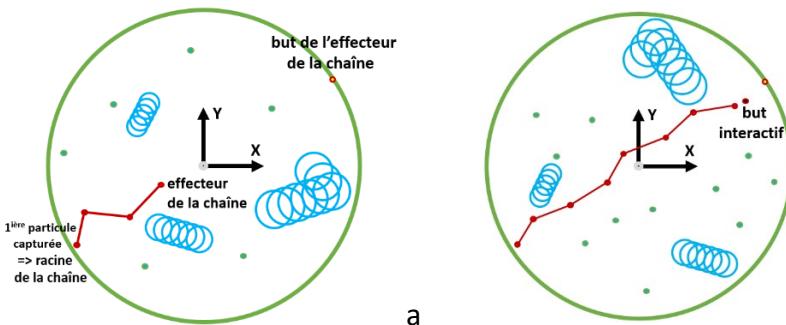


Fig 1 pitch : une arène circulaire contient des entités utiles (les *particules* pour construire une chaîne articulée) ou dangereuses (les *faiseurs*<sup>1</sup> qui détruisent la chaîne à leur contact). Le but est de construire le plus rapidement possible une chaîne reliant les deux bords opposés de l'arène.

## 1. Introduction

Ce projet est indépendant de celui du semestre dernier. Le lien reste néanmoins la mise en œuvre des grands principes (*abstraction*, *ré-utilisation*), les conventions de présentation du code et les connaissances accumulées jusqu'à maintenant dans ce cours. Le but du projet est surtout de se familiariser avec deux autres grands principes, celui de *séparation des fonctionnalités* (*separation of concerns*) et celui d'*encapsulation* qui deviennent nécessaires pour structurer un projet important en *modules* indépendants.

Nous mettrons l'accent sur le lien entre *module* et *structure de données*, et sur la robustesse des modules aux erreurs. Par ailleurs *l'ordre de complexité* des algorithmes sera testé avec des fichiers tests plus exigeants que les autres.

Vous pouvez faire plus que ce qui est demandé dans la donnée mais n'obtiendrez aucun bonus ; notre but est d'éviter que vous passiez plus de temps que nécessaire pour faire ce projet au détriment d'autres matières. Dans tous les cas, vous êtes obligé de faire ce qui est demandé *selon les indications de la donnée et des documents des rendus*. Vos éventuelles touches personnelles ne doivent pas interférer avec les présentes instructions.

Le projet étant réalisé par groupes de deux personnes, il peut comporter un oral final approfondi individuel noté pour lequel nous demandons à chaque membre du groupe de comprendre le fonctionnement de l'ensemble du projet. Une performance faible à l'oral implique une baisse individuelle des notes des rendus.

La suite de la donnée indique les **variables** en ***italique gras*** et les **constantes** globales en **gras** (la valeur des constantes est visible dans les annexes de ce document ; des fichiers .h seront fournis. On utilisera la *double précision* pour des calculs en virgule flottante.

**But** : le challenge de ce jeu est de construire et de guider une chaîne articulée d'une manière très efficace afin de relier deux points le plus rapidement possible avec cette chaîne articulée (le score est mesuré par la valeur d'un compte à rebours). Tous les éléments existent à l'intérieur d'un espace circulaire (l'arène). Les particules et les *faiseurs*<sup>1</sup> se déplacent en ligne droite et rebondissent sur les bords de l'arène. Si leur nombre maximum n'est pas encore atteint les particules se divisent à intervalle régulier tout en réduisant leur vitesse à chaque division, sinon elles disparaissent. Les *faiseurs* détruisent la chaîne s'ils touchent une de ses articulations. Des règles bien précises permettent de capturer des particules pour construire une chaîne et pour guider la chaîne vers le point opposé à son point de départ.

---

<sup>1</sup> Le mot *faiseur* provient du roman de science-fiction *Dune* de Frank Herbert ; il désigne un énorme et dangereux vers se déplaçant dans un désert.

## 2. Modélisation des composantes du jeu

Le programme va gérer deux activités complémentaires et quasi-simultanées grâce à la *programmation par événements* :

1. Les actions sur la souris permettent de **construire** ou de **définir le but** de la chaîne articulée.
2. Un timer active des mises à jour à intervalle régulier. Cela concerne le **déplacement** des particules et des faiseurs et le **guidage** de la chaîne articulée. Chaque mise à jour correspond à l'écoulement d'une unité de temps du jeu. Un compte à rebours est initialisé par la lecture de fichier (valeur maximum **score\_max**) ; il est décrémenté d'une unité à chaque mise à jour. Le jeu se termine avec échec quand il atteint la valeur nulle. En cas de fin avec succès, sa valeur est votre **score** : plus elle est grande, mieux c'est.

Nous demandons de structurer la mise à jour selon les étapes suivantes (détails dans les sections suivantes) :

```
// UNE mise à jour du jeu (tant que le score est strictement positif, sinon fin du jeu avec échec)
// entrée modifiée : ensemble des entités

    Décrémentation du score

    Pour chaque particule
        Incrémentation du compteur de la particule
        Si compteur = time_to_split
            Si nb_particule = nb_particule_max
                Destruction de la particule
            Sinon
                Décomposition en 2 particules
            Déplacement éventuel de la (ou des nouvelles) particule(s)

    Pour chaque faiseur
        Déplacement éventuel s'il n'y a pas de collision avec un autre faiseur
        Si Détection de collision avec une articulation de la chaîne
            Destruction de la chaîne

    Si le jeu est en mode GUIDAGE
        Effectuer UNE SEULE exécution de l'algorithme de Guidage
        Si Détection de collision d'une articulation avec un faiseur
            Destruction de la chaîne
        Si le but final se trouve dans la région de capture de l'effecteur
            Fin du jeu avec succès !
```

**Ebauche de pseudocode 1** : chaque mise à jour temporelle des entités du jeu contient différentes catégories d'actions qui doivent être effectuées dans l'ordre indiqué par cette ébauche de pseudocode..

### L'arène de jeu :

L'arène est représentée par un cercle centré en (0,0) de rayon **r\_max**. Les entités doivent être entièrement comprises dans ce domaine. La figure 1 montre le système de coordonnée **Monde** du jeu avec **X comme axe horizontal, orienté positivement vers la droite, et Y comme axe vertical, orienté positivement vers le haut**.

Les entités mobiles du jeu doivent toutes être contenues dans l'arène. C'est pourquoi une réflexion sur le bord intérieur de l'arène est effectuée lorsque le mouvement d'une entité particule ou faiseur les fait sortir de cette arène comme détaillé dans la section suivante.

### 2.1 Les entités mobiles du jeu

En plus de l'arène, deux entités mobiles autonomes existent dans cette simulation : particule et faiseur. Elles possèdent quelques propriétés communes :

- La **position** d'un point de référence (x, y) qui doit appartenir à l'espace de l'arène de rayon **r\_max**
- son rayon (nul pour une particule, ou compris entre [**r\_min\_faiseur**, **r\_max\_faiseur**] pour un faiseur)
- L'orientation  $\alpha$  du déplacement exprimée en radian dans l'intervalle  $[-\pi, \pi]$
- La norme du déplacement dans l'intervalle [0, **d\_max**]

### 2.1.1 Particule et faiseur : points communs des déplacements et de la collision avec l'arène

Les entités particule et faiseur se déplacent en permanence en ligne droite (Fig2 : passage de l'instant 0 à l'instant 1) sauf quand le déplacement les fait sortir de l'arène (Fig2 : calcul pour l'instant 2 en rouge). Dans ce cas le déplacement en ligne droite est annulé et remplacé par un déplacement effectué dans la direction produite par la réflexion vis-à-vis du bord de l'arène (Fig2 : modification de la direction du déplacement pour l'instant 2 en noir ; remarquer l'égalité de l'angle de chaque côté de la droite qui relie le centre du cercle avec la position de l'entité à l'instant 1).

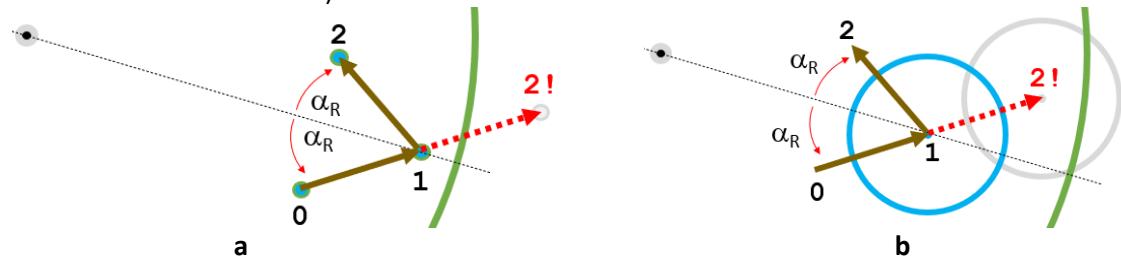


Fig 2 : description de deux mises à jour de la position d'une particule (a) ou d'un faiseur (b) ; de 0 à 1 en ligne droite ; de 1 à 2 avec une réflexion de la direction de déplacement car la position 2 en rouge les fait passer totalement ou partiellement en dehors de l'arène.

La superposition particule-faiseur ne pose pas de problème ; cela n'a aucune conséquence.

### 2.1.2 Spécificité de l'entité Particule : split ou destruction

Chaque particule possède un compteur initialement à 0 qui est incrémenté à chaque mise à jour du jeu. Lorsqu'il atteint la valeur `time_to_split`, et que le nombre de particules est égal à `nb_particule_max`, alors la particule est détruite, sinon elle se sépare en deux nouvelles particules (Fig3). Le compteur des nouvelles particules est ré-initialisé à zéro. Leur orientation est donnée par l'ancienne orientation à laquelle on ajoute/soustrait la quantité `delta_split` tandis que leur déplacement est donné par l'ancien déplacement multiplié par `coef_split` (< 1.).

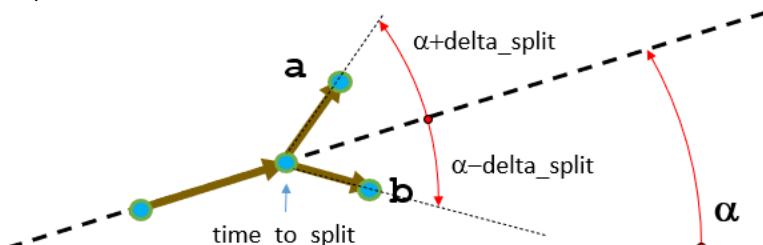


Fig 3 : lorsque le compteur de la particule atteint `time_to_split` et que le nombre total de particules est strictement inférieur à `nb_particule_max`, elle est remplacée par 2 nouvelles particules **a** et **b** qui sont déplacées avec une direction différente (+ ou - `delta_split`) et un déplacement multiplié par `coef_split`

### 2.1.3 Spécificité de l'entité Faiseur

Chaque faiseur comporte un nombre d'éléments circulaires (au moins 1) ; ce nombre reste constant. Le centre de chaque élément est placé sur une des positions précédentes du premier élément (appelé sa tête). La fig 4a montre en orange la possible prochaine position de la tête d'un faiseur de 8 éléments pour une mise à jour. Si elle est validée tous ses éléments se déplacent de la même quantité (Fig 4b). Cependant en cas de collision avec l'arène, le mouvement de la tête détermine la nouvelle direction qui sera prise (Fig 4c). Par ailleurs, si le déplacement de la tête entre en collision avec un élément d'un autre faiseur alors le faiseur reste immobile (Fig 4d).

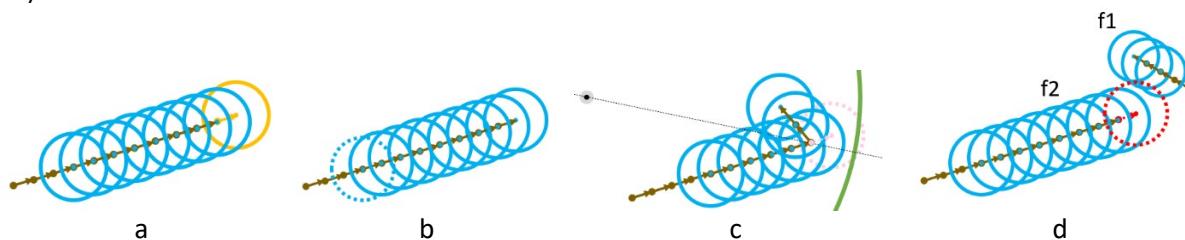
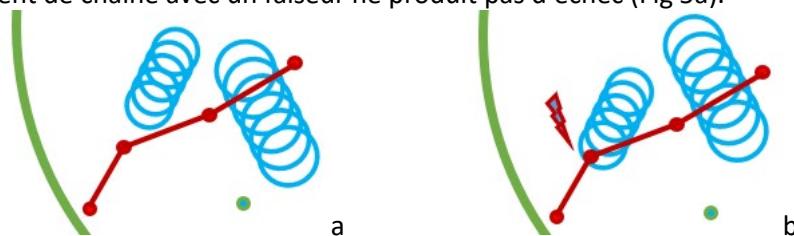


Fig 4 : mise à jour normale d'un faiseur de 8 éléments (a->b) ; collision avec l'arène (c) ou un autre faiseur (d)

## 2.2 Gestion de l'entité Chaîne

### 2.2.1 Définition des conditions de succès du jeu

Une chaîne articulée est définie par une suite de points, appelés des *articulations*, dont le premier est appelé la *racine* de la chaîne et le dernier est appelé l'*effecteur* de la chaîne (Fig 1). Le jeu s'arrête avec **succès** lorsque l'effecteur a atteint le but situé à l'opposé de la racine sur le bord de l'arène. La chaîne est détruite dès qu'une des articulations est incluse dans un des éléments d'un faiseur (Fig 5b). Par souci de simplification, l'intersection d'un segment de chaîne avec un faiseur ne produit pas d'échec (Fig 5a).



### 2.2.2 Modes de jeu : CONSTRUCTION ou GUIDAGE de la chaîne articulée

Le jeu peut se trouver dans l'un des 2 modes suivants :

- **CONSTRUCTION**: l'entité de chaîne articulée n'existe pas en début de partie ; un unique exemplaire est progressivement construit par la personne qui joue une partie. Cela est effectué en capturant des particules à l'aide de la souris.
- **GUIDAGE**: dans ce mode, la souris sert à définir un but intermédiaire à atteindre par l'effecteur. Chaque mise à jour du jeu inclut alors une (seule) exécution de l'algorithme de Guidage (pseudocode1).

On peut facilement passer d'un mode à l'autre à tout moment du jeu en cliquant respectivement sur le bouton gauche (Construction) ou sur le bouton droit (Guidage) de la souris, ou avec un RadioButton (section 7).

### 2.2.3 Construction de l'entité Chaine

Au tout début du mode Construction la position de la souris aide à déterminer la position de la région circulaire initiale de *capture* de rayon *r\_capture*. Cette région sert à capturer une particule pour la convertir en une articulation de la chaîne articulée. Pour cela, à chaque mouvement de la souris, on projette sa position (point noir entouré de jaune dans Fig6a) sur le bord de l'arène. Ensuite la région de capture est toujours centrée sur la dernière articulation de la chaîne. La capture est confirmée par la pression du bouton gauche de la souris. La capture n'est PAS possible s'il y a plusieurs particules présentes simultanément dans la région de capture.

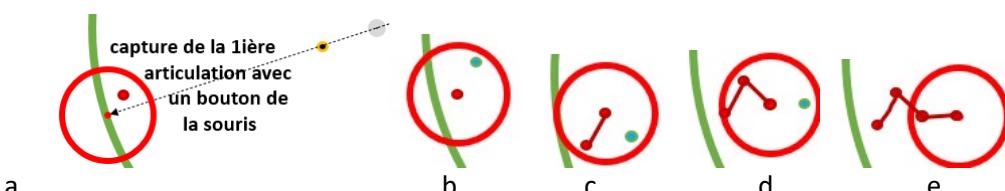


Fig 6 : région de capture initiale (a) ; ensuite la région est toujours centrée sur la dernière articulation de la chaîne articulée (b-e). la chaîne est dessinée avec une ligne reliant ses articulations successives.

La *racine* de la chaîne est la première articulation capturée. Sa position reste fixe pour le reste de l'existence de cette chaîne. Quand on ajoute une articulation supplémentaire cette articulation s'appelle l'*effecteur* de la chaîne ; on peut le guider avec l'algorithme décrit à la section suivante.

### 2.2.4 Guidage de l'entité Chaine

Le Guidage de la chaîne permet d'augmenter ses chances de succès de plusieurs manières. Il permet de :

- 1) rapprocher l'effecteur d'un endroit où passera une particule, ce qui permettra de rallonger la chaîne,
- 2) rapprocher l'effecteur du *but final*, défini comme le point sur le bord de l'arène qui est symétrique à la racine (Fig 7a), en guidant l'effecteur à l'aide de *buts intermédiaires* (Fig 7b).
- 3) changer la forme de la chaîne pour éviter qu'une articulation ne soit sur le chemin d'un faiseur car l'inclusion d'une des articulations de la chaîne dans l'un des éléments d'un faiseur détruit la chaîne (Fig 5b).

Ces 3 possibilités peuvent être mises en œuvre à l'aide de l'algorithme de Guidage détaillé avec la Fig 8. Seul cet algorithme de Guidage est autorisé. De plus, une seule exécution de l'algorithme est autorisée par mise à jour du jeu (Pseudocode 1). Le jeu se termine avec succès quand le but final se trouve dans la région de capture.

Lorsque le jeu est en cours dans le mode de Guidage, chaque modification de la position de la souris produit une mise à jour du but intermédiaire. Ce but est situé au maximum à une distance  $r\_capture$  de l'effecteur (Fig 7b). Dans tous les cas l'algorithme cherche à attirer la position de l'effecteur vers ce but intermédiaire.

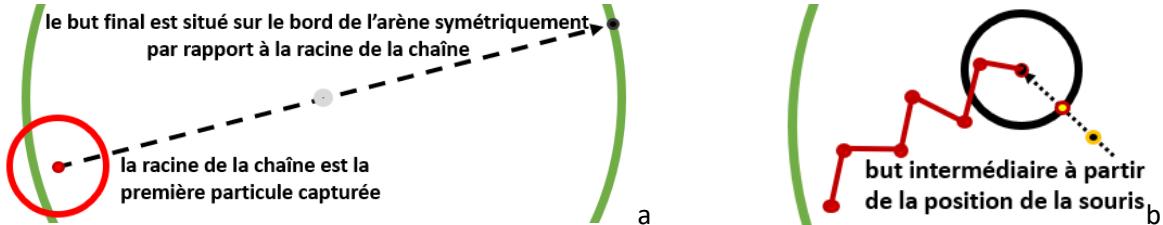


Fig 7 : le but final est symétrique à la racine (a), le but intermédiaire est limité à la région de capture à une distance  $r\_capture$  de l'effecteur (b)

Contrainte de l'algorithme: la **longueur**  $l_k$  entre l'articulation  $k$  et l'articulation  $k-1$  doit rester constante

L'algorithme fonctionne dès qu'un effecteur distinct de la racine existe. Il comporte deux itérations :

- 1) **effecteur vers racine** (Fig 8a) : elle place l'effecteur sur le but intermédiaire et met à jour la position  $p'$  des articulations en préservant les longueurs inter-articulation  $l_k$ . Connaissant la position  $p'_{k-1}$  on construit le vecteur  $p'_{k-1}p_{k-1}$  pour placer le point  $p'_{k-1}$  à la distance  $l_k$  de  $p'_{k-1}$ .
- 2) **racine vers effecteur** (Fig 8b) : elle utilise les positions  $p'$  de la première itération. Elle commence en replaçant la racine à sa position fixe  $p_0$  et met à jour la position  $p''$  des articulations en préservant aussi  $l_k$ . Connaissant  $p''_{k-1}$  on construit le vecteur  $p''_{k-1}p'_{k-1}$  pour placer le point  $p''_{k-1}$  à la distance  $l_k$  de  $p''_{k-1}$ .

**Attention aux divisions par zéro !** Il faut annuler le déplacement en cas de détection de ce problème.

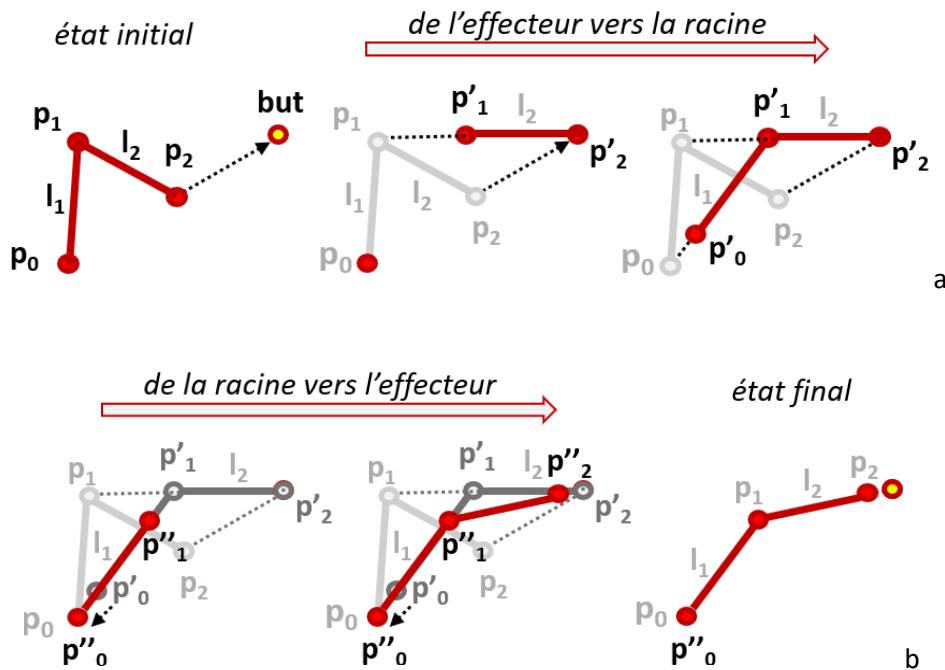


Fig 8 : itération de **l'effec teur vers la racine** (a) et de **la racine vers l'effec teur** (b).

### 3 Actions à réaliser par le programme en C++

#### 3.1 Exécution du jeu en continu vs en pause

L'exécution du jeu en **continu** est une boucle infinie réalisée avec un timer de GTKmm.

Celle-ci peut être mise en **pause** pour pouvoir effectuer ces actions :

- Une seule **mise à jour du jeu** à la fois à l'aide d'un bouton Step.
- La **Lecture** d'un fichier pour initialiser l'état du jeu à l'aide d'un bouton Open.
- L' **Ecriture** d'un fichier décrivant l'état actuel du jeu à l'aide d'un bouton Save.
- La **relecture** du dernier fichier à l'aide d'un bouton Restart.
- la *mise au point des actions des boutons de la souris* pour construire la chaîne ou définir un but intermédiaire. Le clic d'un bouton de la souris doit aussi *produire une seule mise à jour du jeu*. Cela dit, le score n'est pas valide si le jeu se termine avec succès en tirant parti de cette possibilité...

Après **chaque action de lecture et chaque mise à jour de la simulation**, une demande de rafraîchissement de l'affichage de l'état courant du jeu doit être effectuée. L'affichage est fait dans une fenêtre graphique (section 5) avec l'aide d'une interface dédiée (section 6). La section 7 précise l'architecture modulaire du projet et comment la simulation et l'affichage sont gérés à l'aide de la programmation par événements. La section 8 précise la répartition des tâches entre les 3 rendus pour structurer votre travail.

#### 3.2 Implémentation C++ (*compilation avec l'option std=c++17*)

On exige l'usage de la fonction **atan2** de **<cmath>** pour obtenir une orientation à partir d'un vecteur (x,y).

#### 3.3 Tolérance sur les tests en virgule flottante

**tests d'égalité** : (ex : norme nulle) ils doivent être remplacés par un test mettant en œuvre la tolérance **epsil\_zero** sur l'écart entre la valeur théorique (ex : nulle) et la valeur obtenue par le calcul ; le test donne *true* si la valeur absolue de cet écart est inférieure à **epsil\_zero**.

#### 3.4 Traitement des erreurs d'arrondis causées par le formatage des fichiers

Les arrondis causés par le formatage des fichiers nous imposent d'effectuer des tests moins stricts quand on lit un fichier en comparaison de ces mêmes tests pendant le jeu. C'est pourquoi la valeur **epsil\_zero** doit être considérée comme **nulle** pour les tests d'inégalités suivants quand ils sont effectués à la lecture de fichier.

**tests d'inégalité** :

Posons qu'un cercle est représenté par un centre de position C et un rayon R :

Un point P est inclus dans un cercle (C, R) si : **distance(P,C) < R - epsil\_zero**

Un cercle (C1, R1) est inclus dans un cercle (C2, R2) si : **distance(C1,C2) < R2 - R1 - epsil\_zero**

Un cercle (C1, R1) intersecte un cercle (C2, R2) si : **distance(C1,C2) < R2 + R1 + epsil\_zero**

### 4. Sauvegarde et lecture de fichiers tests : format du fichier

Votre programme doit être capable d'initialiser l'état du jeu à partir d'un fichier texte. Il doit aussi pouvoir mémoriser la configuration actuelle dans un fichier texte également. Cela vous permettra de pouvoir créer vos propres scénarios de tests avec un éditeur de texte comme geany.

#### 4.1 Caractéristiques des fichiers tests

L'opération de lecture doit être indépendante des aspects suivants qui peuvent être différents d'un fichier à l'autre : présence de lignes vides commençant par **\n** ou **\r**, les commentaires commençant par **#** en début de ligne, et les espaces avant ou après les données. Les indentations visibles dans le format ci-dessous ne sont pas obligatoires non plus. Les fins de lignes peuvent contenir **\n** et/ou **\r** à cause du système d'exploitation sur lequel le fichier a été créé ; votre programme doit pouvoir traiter ces cas (cf série fichier).

Le fichier contient le **score** (entier positif) puis les données des particules puis les faiseurs puis la chaîne. Dans ce format **x** et **y** désignent la position d'une entité ; **a** une orientation en  $rd \in [-\pi, \pi]$ , **r** un rayon positif, **c** un compteur de mises à jour ( $0 \leq c < \text{time\_to\_split}$ ), **d** un déplacement dans  $[0, D_{\max}]$ , **nbe** un nombre d'éléments  $> 0$ , **mode** peut prendre seulement 2 valeurs : CONSTRUCTION ou GUIDAGE.... Des exemples de fichiers sont fournis.

format général du fichier : les indices commencent à zéro

```

# Nom du scenario de test
#
score

# nombre d'entité particule puis les données d'une entité par ligne
nbPart
    x0 y0 a0 d0 c0
    ...
    ...

# nombre d'entité faiseur puis les données d'une entité par ligne
nbFais
    x0 y0 a0 d0 r0 nbe0
    ...
    ...

# nombre d'articulations (nul veut dire « pas de chaîne »)
#      puis une ligne par articulation
nbArt
    x0 y0 // racine de la chaine s'il y en a une
    ...
mode

```

## 4.2 Vérifications à effectuer pendant la lecture et conséquences d'une détection d'erreur

La **lecture** doit vérifier :

- Score strictement positif et inférieur ou égal à **score\_max**
- Toutes les entités particules et faiseurs sont contenues dans l'arène
- Toutes les positions des articulations sont contenues dans l'arène
- Mobile : **d** compris dans [0, **d\_max**]
- Particule : nombre dans [0, **nb\_particule\_max**] ; compteur **c** dans [0, **time\_to\_split**[
- Faiseur : intervalle de validité du rayon **r** ; **nbe>0** et absence de collision d'éléments inter-faiseurs
- Chaine : racine distance intérieure et longueur entre articulations consécutives  $\leq r_{capture}$ , absence d'inclusion d'une articulation dans un élément de faiseur

Si plus de données que nécessaire sont fournies sur la ligne de fichier elles sont simplement ignorées sans générer d'erreur de lecture. Un commentaire peut aussi suivre les données et ne doit pas poser de problèmes. Les conséquences d'une détection d'erreur dépendent du rendu du projet (section 8):

## 5. Affichage et interaction dans la fenêtre graphique

A partir du rendu 2, l'exécution du programme ouvre une fenêtre GTKmm contenant l'interface graphique utilisateur (Fig 9) et le dessin du jeu dans un *canvas*. Le dessin doit montrer l'arène complète du jeu (Fig 1).

**Taille de la fenêtre d'affichage en pixels** : La taille initiale du *canvas* dédié au dessin du jeu est de **taille\_dessin** en largeur et en hauteur (annexe C). La taille de la fenêtre peut changer durant l'exécution du programme. Un changement de taille de fenêtre ne doit pas introduire de distorsion dans le dessin (un cercle reste un cercle quelle que soit la taille et la proportion de la fenêtre).

**Formes et couleurs** : Le module graphique de bas niveau (**graphic**) met à disposition une table de couleurs prédéfinie dont les index peuvent être indiqué en paramètre des fonctions de dessin (section 8). Les entités suivantes devront utiliser les formes et couleurs suivantes :

- On travaille en light mode = le fond du dessin est blanc.
- La bordure de l'arène du jeu est verte
- Une particule est un cercle de rayon **r\_viz** de bord vert avec l'intérieur de couleur cyan (Fig 9).
- Un faiseur est un cercle vide de couleur bleue (dans la Fig9 ils sont très proches=> illusion d'une capsule).
- Une chaine est un ensemble d'articulations (cercle vide de rayon **r\_viz**) de couleur rouge, reliées par des traits rouges. Le cercle de capture est également de couleur rouge. Le but est noir.

## 5.1 Interaction avec le clavier

Utiliser la touche clavier ‘s’ pour faire la même action que le bouton Start/Stop.

Utiliser la touche clavier ‘1’ pour faire la même action que le bouton Step.

Utiliser la touche clavier ‘r’ pour faire la même action que le bouton Restart.

## 5.2 Interaction avec la souris

Toute variation de déplacement de la souris doit conduire à interpréter sa position selon le mode courant. Cela est obtenu en souscrivant aux événements de type « mouse move ». Indépendamment de ces événements, un clic avec le bouton gauche de la souris a pour conséquence :

- d'entrer (ou de rester) dans le mode Construction
- s'il n'y a pas encore de chaîne, création de l'articulation racine selon la méthode de la figure 6a
- sinon, ajout d'une articulation supplémentaire comme illustré sur Fig 6b-e

et un clic avec le bouton droit de la souris a pour conséquence :

- d'entrer (ou de rester) dans le mode Guidage
- de définir la position d'un but intermédiaire selon la méthode de la figure 7b.

Lorsque l'exécution est en pause (section 3.1) un clic de souris doit être suivi d'une seule exécution.

## 6. Interface utilisateur (GUI)

Nous utilisons une seule fenêtre graphique divisée en 2 parties =>

- Les boutons des actions ou affichage de données =>
- Le dessin de l'état courant du jeu dans un *canvas* =>

### L'interface utilisateur doit contenir:

Commandes générales (seulement dans l'état **pause** du jeu):

- **Exit** : quitte le programme
- **Open** : remplace la simulation par le contenu du fichier dont le nom est fourni. Les structures de données antérieures doivent être supprimées ; on obtient donc un écran blanc si une erreur est détectée à la lecture.
- **Save** : mémorise l'état actuel du jeu dans le fichier dont le nom est fourni.
- **Restart** : bouton pour ré-initialiser avec le dernier fichier lu

Jeu :

- **Start** : bouton pour commencer/stopper l'exécution en continu
- **Step** (si l'exécution est en **pause**) : calcule seulement une mise à jour
- Choix du **mode** avec radio-button : Construction (default) ou Guidage

Affichage de données générales :

- **Score**
- **nombre d'entités particule**
- **nombre d'entités faiseur**
- **nombre d'articulations**

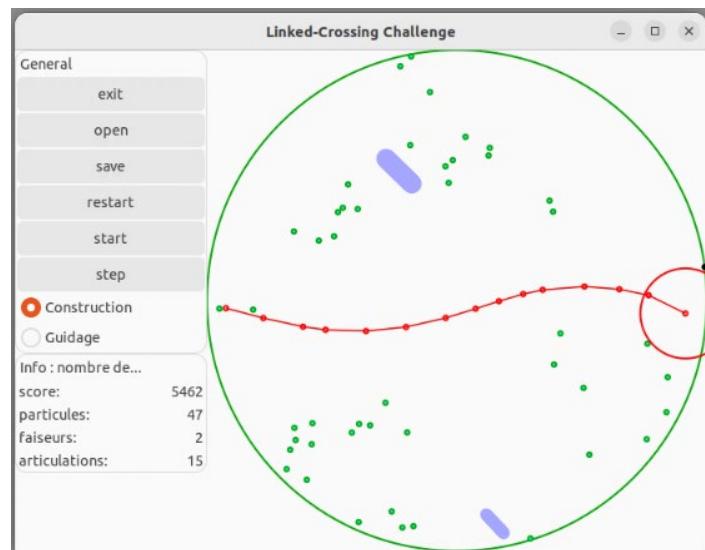


Fig 9 : GUI contenant les boutons et affichage de données (à gauche) et le dessin de l'état courant du jeu (à droite)

## 7. Architecture logicielle

### 7.1 Décomposition en sous-systèmes

L'architecture logicielle de la figure 10 décrit l'organisation minimum du projet en sous-systèmes avec leur responsabilité (Principe de Séparation des Fonctionnalités) :

- **Sous-système de Contrôle** : son but est de gérer le dialogue avec l'utilisateur (Fig 10b). Si une action de l'utilisateur impose un changement de l'état de la simulation, ce sous-système doit appeler une fonction du **sous-système du Modèle** qui est le seul responsable de gérer les structures de données du jeu (voir point suivant).
- Le sous-système de contrôle est mis en œuvre avec deux modules :
  - Le module **projet** qui contient la fonction **main** : son rôle est modeste car il est seulement responsable de traiter les éventuels arguments fournis sur la ligne de commande au lancement du programme. Pour le rendu1, le sous-système de **Contrôle** ne contient que le module **projet**.
  - le module **gui** est créé à partir du rendu2 pour gérer le dialogue utilisateur à l'aide de l'interface graphique mise en œuvre avec GTKmm. Nous fournirons une partie du code source de **gui**.
- **Sous-système du Modèle**: est responsable de gérer les structures de données du jeu. Il est mis en œuvre sur plusieurs niveaux d'abstractions selon les Principes d'Abstraction et de Ré-utilisation (section 7.2).
- **Utilitaire générique indépendant du Modèle** : un module **tools** gère des points, manipule des vecteurs, et effectue des tests sur des cercles (cf annexe B) ; c'est l'équivalent d'une bibliothèque mathématique.
- **Sous-Système de Visualisation** : le module **graphic** dessine l'état courant du jeu à l'aide des entités élémentaires gérées par le module **tools**. Le module **graphic** rassemble les dépendances vis-à-vis de GTKmm pour faire les dessins. On autorise l'inclusion de son interface dans l'interface **tools.h** pour que le Modèle puisse choisir les couleurs prédéfinies dans **graphic.h**.

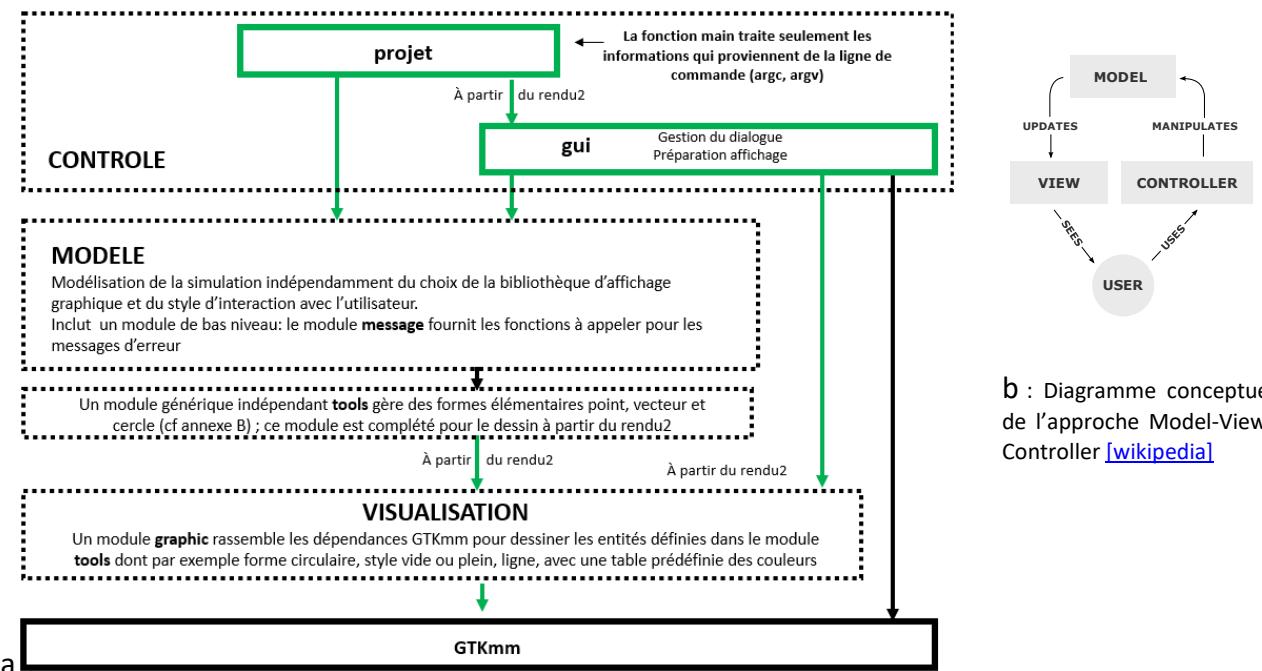


Fig 10 : Architecture logicielle minimale à respecter (a), inspirée par l'approche MVC (b)

### 7.2 Décomposition du sous-système MODELE en plusieurs modules

Cette partie du présent document fait partie de l'étape *d'Analyse* dans la mise au point d'un projet. En bref, le Modèle gère la simulation ; ce Modèle est organisé en plusieurs modules pour maîtriser la complexité du problème et faciliter sa mise au point. La Figure 11 présente l'organisation minimale à adopter en termes d'organisation des modules :

- **Au plus haut niveau**, le module **jeu** gère le déroulement du jeu et les autres actions (lecture, écriture de fichier). Ce module doit garantir la cohérence globale du Modèle. C'est pourquoi, en vertu du principe d'abstraction le module **jeu** est le seul module dont on peut appeler des fonctions en dehors du MODELE (Fig

11). Si le sous-système de Contrôle veut modifier l'état du jeu cela doit se faire par un appel d'une fonction de **jeu.h**. Par exemple la lecture du fichier doit se faire en appelant une fonction disponible dans **jeu.h**.

- **Niveau intermédiaire**: un module **mobile** doit définir les entités mobiles autonomes du jeu (particule et faiseur) avec une hiérarchie de classe (2 niveaux suffisent). Nous accepterons une version simplifiée, sans hiérarchie de classe, pour le **rendu1** seulement. De plus un module indépendant doit gérer la **chaine** articulée.

- **Au plus bas niveau** : nous mettons à disposition un ensemble de fonctions dans **message.h** . Ces fonctions doivent être appelées pour faire afficher les messages d'erreurs liées au Modèle et détectées à la lecture d'un fichier. Une fonction supplémentaire est fournie pour afficher un message quand la lecture est effectuée avec succès. Il n'est pas autorisé de modifier le code source de ce module car il sera utilisé par notre autograder.

## 7.3 Module générique indépendant tools

### 7.3.1 Indépendance de tools

Ce module est équivalent à une bibliothèque mathématique destinées à être utilisées par de nombreux autres modules de plus haut niveau (principe de Ré-utilisation). L'idée fondamentale est qu'il doit aussi être conçu pour être utilisable par d'autres programmes très différents de notre projet. Pour cette raison **AUCUN des noms de types/concepts du niveau supérieur MODELE ne doit apparaître dans tools**. On y trouvera en particulier les fonctions effectuant les tests de collision utiles pour les niveaux supérieurs (cf Rendu1).

### 7.3.2 Relation « Possède-un » entre la hiérarchie de classes du module mobile et les types de tools

Les classes du MODELE devront utiliser les types mis à disposition dans l'interface de **tools** MAIS SEULEMENT en utilisant la relation « **possède-un** » plutôt que la relation « **est-un** ». C'est-à-dire que, par exemple, une entité faiseur possède un ou plusieurs éléments circulaires pour ses calculs géométriques et l'affichage mais, en vertu du principe de *séparation des fonctionnalités* le module **tools** n'a pas vocation à servir de classe parente pour l'ensemble des applications qui utilisent ce module.

### 7.3.3 Type concret S2d et autres types de tools

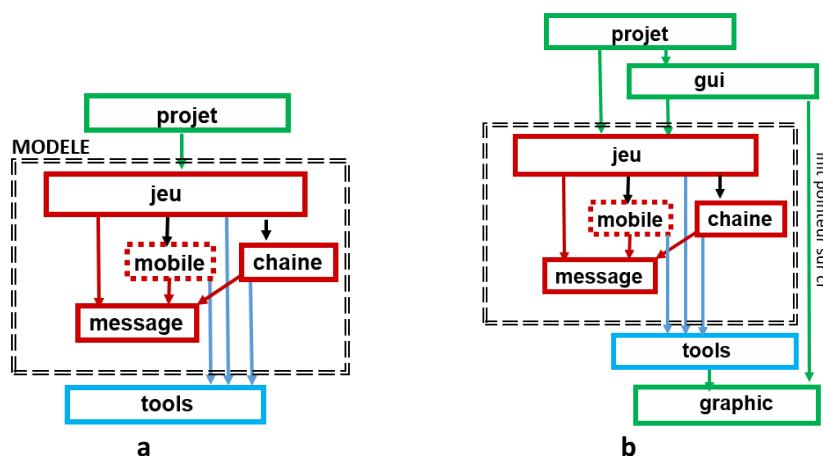
Nous demandons d'implémenter le type **S2d** qui permet de modéliser une position et/ou un vecteur soit avec cette approche :

```
struct S2d {double x=0.; double y=0.;}; //plus robuste aux bugs
```

ou avec cette approche :

```
typedef array<double,2> S2d;
enum {X,Y}// quand on veut accéder explicitement à une coordonnée
```

Pour l'entité cercle, nous considérons qu'elle est suffisamment simple et de bas-niveau pour vous autoriser à la créer à l'aide de **struct**. Alternativement, vous pouvez aussi la dériver d'une classe parente possédant un attribut **S2d** pour représenter la position d'un point.



**Figure 11 :** (a) architecture minimale montrant les dépendances entre modules du sous-système *Modèle* pour la recherche d'erreur dans le fichier (rendu1); (b) modules et dépendances supplémentaires pour la mise en œuvre de l'interface graphique (rendus 2 et 3)

## 7.4 Module graphique de bas-niveau (graphic)

A partir du rendu2 le module **tools** va aussi offrir des fonctions de dessin pour les types qui y sont définis dont **S2d**. Cependant le module **tools** doit rester indépendant d'une librairie graphique particulière (principe de regroupement des dépendances). C'est pourquoi les dépendances vis-à-vis de la bibliothèque **GTKmm** doivent être rassemblées dans le module **graphic**. C'est dans ce module qu'on définit une table de couleurs prédéfinies et les fonctions de tracé des formes géométriques ; son interface **graphic.h** met à disposition des symboles pour définir une table de couleurs prédéfinies. On autorise **tools.h** d'inclure **graphic.h** pour avoir accès à ces symboles de style/couleur dans le MODELE.

## 8. Syntaxe d'appel et répartition du travail en 3 rendus notés

Chaque rendu sera précisément détaillé dans un document indépendant. Votre exécutable doit s'appeler **projet**. Selon le rendu le programme doit pouvoir traiter un argument optionnel sur la ligne de commande.

### **8.1 Rendu1 :** Son architecture est précisée par la Fig 11a.

Ce rendu sera toujours testé en indiquant un nom de fichier de test sur la ligne de commande selon la syntaxe suivante : **./projet t01.txt**

Le programme cherche à initialiser l'état du jeu en construisant une première version de vos structures de données. Le programme s'arrête dès la première erreur trouvée dans le fichier : il faut afficher dans le terminal le message d'erreur fourni avec le module **message** et quitter le programme. Le programme s'arrête aussi après la lecture du fichier s'il n'y a aucune erreur ; dans ce cas, il y a affichage d'un message indiquant le succès de la lecture.

Il sera possible de faire évoluer votre choix de structure de données entre le rendu1 et les suivants.

Le rendu1 ne doit PAS utiliser GTKmm.

### **8.2 Rendu2 :** Son architecture est précisée par la Fig 11b.

Ce rendu avec GTKmm sera toujours testé comme pour le rendu1, en indiquant un nom de fichier de test sur la ligne de commande selon la syntaxe suivante : **./projet t01.txt**

Ce rendu construit les structures de données et affiche l'état initial avec GTKmm (section 6). Ce rendu sera testé en effectuant plusieurs lecture/écriture/relecture avec le GUI pour vérifier que l'affichage est bien correct, que le programme gère bien les erreurs détectées à la lecture et qu'il ré-initialise correctement les structures de données à chaque lecture de fichier. En effet, il est demandé de détruire les structures de données existantes avant de commencer toute lecture.

Attention: dès le rendu2 il ne faut plus quitter le programme en cas de détection d'erreur. Dès la première erreur détectée à la lecture, il faut afficher dans le terminal le message d'erreur fourni, interrompre la lecture, détruire la structure de donnée en cours de construction, ce qui produit une arène vide, et attendre une nouvelle commande en provenance de l'interface graphique.

**L'exécution du jeu** devra gérer seulement le déplacement des particules et des faiseurs au cours du temps pour le rendu2. On ne fera aucune action sur la chaîne à part son dessin.

Un rapport devra décrire les choix de structures de donnée et préciser la méthode de travail du groupe.

**8.3 Rendu3 :** Ce rendu utilise toujours GTKmm (avec l'architecture de la Fig 11b). Si un nom de fichier est indiqué sur la ligne de commande il doit être ouvert pour initialiser l'interface graphique et le dessin, incluant l'affichage de la valeur initiale de l'état de la planète. Si aucun nom n'est fourni le programme initialise l'interface graphique et attend qu'on l'utilise pour demander l'ouverture d'un fichier.

Plusieurs scénarios de simulation seront testés pour illustrer les règles définies dans le présent document.

Un rapport final devra compléter celui du rendu2 et illustrer quelques simulations.

## **ANNEXE A : constantes globales du Modèle définies dans constantes.h**

Ces constantes sont appelées « globales » car elles pourraient être nécessaires dans plus d'un module du Modèle. L'utilisation de **constexpr** crée automatiquement une instance *locale* dans chaque fichier où constantes.h est inclus ; il n'y a donc pas de problème de définition multiple de ces entités.

Ces constantes sont associées au Modèle ; elle reflète la nature du problème spécifique résolu dans le sous-système du Modèle. Pour cette raison *il n'est pas autorisé d'inclure ce fichier de constantes dans le module utilitaire tools* qui ne doit rester très général/générique et donc n'avoir aucune dépendance vis-à-vis de concepts et de constantes de plus haut niveau. Si vous désirez mettre en œuvre vos propres constantes, les bonnes pratiques sont les suivantes :

- utilisez **constexpr** pour les définir
- définissez-les *le plus localement possible* ; inutile de les mettre dans l'interface d'un module (.h) si elles ne sont utilisées que dans son implémentation (.cc)

Selon nos conventions de programmation, un nom de constante définie avec **constexpr** suit la même règle qu'un nom de variable (E12). Le texte de la donnée les fait apparaître en **gras** dans le texte.

```
#include "tools.h"      // nécessaire pour utiliser epsil_zero et disposer des symboles de graphic.h
enum Mode {CONSTRUCTION,GUIDAGE} ;

constexpr double r_max(100.) ;
constexpr double r_min_faiseur(1.5) ;
constexpr double r_max_faiseur(5.) ;
constexpr double r_capture(18.) ;
constexpr double r_viz(0.9) ; //seulement pour le dessin, pas pour les calculs de collision

constexpr double d_max(r_max/40.) ; // déplacement max par mise à jour du jeu
constexpr unsigned time_to_split(500) ;
constexpr unsigned nb_particule_max(50) ;
constexpr double delta_split(0.5) ; // radian
constexpr double coef_split(0.8) ;

constexpr unsigned score_max(8000) ;
```

## **ANNEXE B : constantes génériques définies dans tools.h**

```
constexpr double epsil_zero(0.5) ;
```

## **ANNEXE C : constante destinée au sous-système de Contrôle**

```
constexpr unsigned taille_dessin(500);           en pixels
```