V 1.01

Projet Informatique – Sections Electricité et Microtechnique

Printemps 2025 : *Linked-Crossing* © R. Boulic & collaborators

Rendu2 (27 avril 23h59)

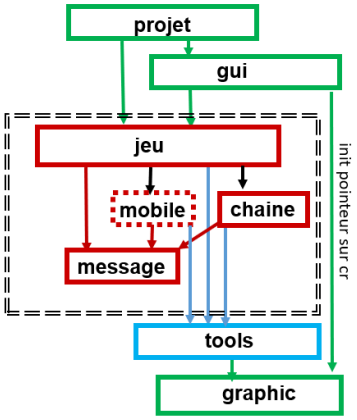
Table des matières :

Objectif de ce document : *Ce document utilise l'approche introduite avec la série théorique sur les méthodes de développement de projet qu'il est important d'avoir faite avant d'aller plus loin. En plus de préciser ce qui doit être fait, ce document identifie des ACTIONS à considérer pour réaliser le rendu de manière rigoureuse. Ces ACTIONS sont équivalentes à celles indiquées pour le projet d'automne ; elles ne sont pas notées, elles servent à vous organiser. Vous pouvez adopter une approche différente du moment que vous respectez l'architecture minimale du projet (donnée générale Fig 11b).*

1. Buts du rendu2 : mise au point de l'interface graphique et lecture/écriture

Ce rendu construit les structures de données et affiche l'état initial avec GTKmm (sections 5 et 6 de la donnée générale).

Un rapport devra décrire les choix de structures de donnée et la répartition des responsabilités des modules et des classes : qui fait quelle tâche ? Où sont les ensembles de données ?



Donnée générale Fig 11b

Lancement et comportement attendu:

Le programme sera lancé selon la syntaxe suivante :

```
./projet t01.txt
```

Le programme construit l'interface graphique et ouvre immédiatement le fichier fourni sur la ligne de commande en lecture.

En cas de succès de la lecture, le programme affiche l'état initial du jeu (dessin) et attend des commandes sur les widgets ou les touches du clavier.

En cas d'échec, c'est-à-dire dès la première erreur détectée à la lecture du fichier, le message d'erreur est affiché (c'est suffisant de l'afficher dans le terminal comme pour le rendu1). De plus les structures de données sont effacées, l'affichage du dessin est totalement vide avec un fond blanc, puis **le programme attend** qu'on utilise l'interface graphique pour ouvrir un autre fichier. On ne doit PAS quitter le programme quand on détecte une erreur. La sauvegarde de l'état courant du jeu dans un fichier sera aussi testée puis on fera des re-lectures des fichiers sauvegardés.

Du point de vue de l'affichage des dessins du jeu, on testera que le changement de la taille de la fenêtre graphique n'introduit pas de distorsion : les cercles restent des cercles.

Le test du jeu sera limité à la partie mise à jour des `particules` et des `faiseurs` du **pseudocode1** de la section 2 de la donnée générale. Cela implique de gérer leur déplacement ainsi que leur decomposition ou leur destruction (particules) ou leur collision (inter-faiseurs). Cela sera contrôlé avec les boutons **start** et/ou **step**.

La sauvegarde de l'état du jeu après creation de plusieurs particules sera aussi testée.

2. Architecture du rendu2 (Fig 11b donnée générale)

Dès le rendu2 il est demandé de mettre en place *une hiérarchie* de classes pour gérer les différents types d'entités dérivées de la superclasse Mobile. Nous vous demandons de fournir **un rapport de maximum 1.5 page** (qui sera complété par 0.5 page d'activité individuelle/cf section 4) décrivant votre organisation du code du Modèle, en particulier :

- **La hiérarchie de classe des** entités du module mobile : Quels attributs/méthodes sont gérés par la superclasse et par les classes dérivées. Optionnel (libre à vous): si vous utilisez le polymorphisme, quelles méthodes sont virtuelles afin de le mettre en œuvre ?

- **La structuration des données des autres entités du Modèle :**
  - Quels sont les attributs de la classe Jeu ?
  - Où et comment sont mémorisés les différents ensembles que doit gérer le jeu ?

- **Brève description des types mis en œuvre dans tools**.

2.1 Module projet

Comme pour le rendu1, Le module **projet** contient la fonction **main()** en charge d'analyser si la ligne de commande est correcte. La nouveauté par rapport au rendu1 est la déclaration de l'interface graphique GTKmm depuis **main()**. La classe responsable de l'interface graphique est définie dans le module **gui** dont le code sera partiellement fourni.

2.2 Sous-système Modèle

Le module **jeu** reste le point d'entrée obligatoire pour des appels depuis le module **projet** ou depuis le module **gui** responsable de de l'interface graphique. Pour le rendu2, il faut les compléments suivants:

- Adapter la **lecture** de fichier pour:
  - ré-initialiser le jeu avant la commencer la lecture de fichier afin de pouvoir lire un nouveau fichier sans être perturbé par l'état des attributs/variables d'état de la lecture précédente.
  - renvoyer un booléen traduisant le succès (true) ou l'échec (false) de la lecture pour que le programme continue de fonctionner après la lecture avec succès et en cas de détection d'erreur.
  - Supprimer les structures de donnée lues en cas d'échec de la lecture.
- Créer une fonction/ méthode de:
  - **sauvegarde** du jeu qui va créer un fichier formaté selon les indications de la section 4 de la donnée générale. On lui donnera un nom de fichier en parametre.
  - **execution** d'une seule mise à jour du jeu pour les particules et les faiseurs seulement (donnée générale pseudocode1).
  - **dessin** de l'état courant du jeu (section 5 donnée générale)

En vertu du principe d'abstraction le module **jeu** délègue aux modules de **mobile** la partie qui concerne chaque entité pour réaliser les actions qui les concernent pour la sauvegarde, le dessin et la mise à jour des 2 entités dérivées de la superclasse Mobile.

2.2.1 ACTION : test d'une suite de plusieurs lectures/sauvegardes (sans le module gui)

Dans cette étape on se concentre sur la partie lecture/écriture de fichier format sans aucun code lié à l'interface graphique (comme pour le rendu1).

L'idée est d'ajouter une nouvelle fonction/méthode de jeu, appelons-la **test**, qui est appelée par projet avec le nom de fichier transmis sur la ligne de commande. Ensuite, dans cette fonction, on demande l'exécution d'une suite d'appels dont le premier la **lecture** du fichier transmis en parametre. On peut enchainer directement avec un appel de **sauvegarde** ou sur une autre **lecture** de fichier (éventuellement avec un nom prédéfini par vous), etc... Si le booléen renvoyé par la **lecture** est true on doit s'attendre à obtenir le même fichier en faisant une **sauvegarde** du jeu (on n'exige pas d'avoir la même présentation en terme d'espaces et de passages à la

ligne). Si le booléen renvoyé par la lecture est false, il faut supprimer les structures de données lues et le bouton de sauvegarde de fichier est désactivé.

2.3 Module gui

Le module **gui** crée l'interface avec les éléments visibles ci-dessous et la surface dédiée au dessin. Il est partiellement fourni ; vous devez le compléter en le connectant au Modèle et au module graphic.

2.3.1 Partie dédiée au dialogue : les widgets (Fig 7 donnée générale)

Les commandes suivantes devront être complétées en les connectant à votre Modèle et au module graphic:

- « **exit** »: quitter le programme
- « **open** »: lire un fichier avec GTKmm pour initialiser un jeu avec detection d'erreur
- « **save** »: si une instance de Jeu existe (en cas de succès à la lecture de fichier) alors on peut sauvegarder l'état courant du jeu dans un fichier dont le nom est fourni à GTKmm.
- « **restart** »: bouton pour ré-initialiser avec le dernier fichier lu ou attendre si ce fichier n'est pas encore défini.

- Affichage des informations suivantes:
  - score
  - nombre d'entités de type particule
  - nombre d'entités de type faiseur
  - nombre d'articulations

  L'image ci-contre montre l'affichage de la partie gauche du GUI

  Les 2 premières informations devront être actualisées lorsque le jeu est lancé avec **start, step**. L'évolution du jeu au niveau du rendu2 peut aussi conduire à la disparition de particules dès que leur nombre maximum est atteint. Si elle existe, la chaîne reste immobile pour le rendu2.

Le test des autres boutons sera limité au comportement suivant:

- « **start** »: le label du bouton devient "**stop**" et un **timer** est lancé qui produit l'affichage d'un compteur entier qui progresse d'une unité à chaque execution du signal handler du timer. Chacune de ces exécutions permet de simuler l'appel d'une mise à jour du jeu. Si on re-clique sur le bouton (qui maintenant affiche "**stop**") alors le timer s'arrête et le label redevient "**start**".

- « **step** »: l'action de ce bouton est seulement prise en compte quand le jeu n'est pas en cours d'exécution (c'est à dire quand on voit le label "start" au-dessus du bouton "step"). Dans ce contexte, un clic sur ce bouton produit UNE SEULE mise à jour du jeu. Cela est simulé en faisant afficher une seule incrémentation du compteur utilisé par le timer (cf bouton start/stop).

Le radiobutton avec les deux choix "Construction"/"Guidage" est destine au contrôle de la chaîne pour le rendu3. Sa valeur par défaut est "Construction" ; on doit pouvoir changer son état mais sans consequence sur le déroulement du jeu.

L'action des boutons **start** et **step** devra produire la mise à jour du score affiché à gauche et s'il y a des entités particules ou faiseur elles doivent être mises à jour selon le pseudocode1, le dessin doit montrer leur nouvel état et le compteur de particules doit être actualisé.

Les 3 touches de clavier '**s**', '**1**' et '**r**' (section 5.1 de la donnée générale) devront produire le même comportement que les boutons associés (respectivement, **start**, **step** et **restart**).

2.3.1.1 ACTION : test du module projet avec initialisation de l'interface graphique (sans dessin)

Dans cette étape on se contente d'établir le lien entre le module **projet** avec main() et le module **gui** fourni qui initialise l'apparence de l'interface graphique (ci-dessus) sans se connecter au Modèle ni à la partie qui s'occupe du dessin.

Commencer par la lecture de fichier en connectant le code fourni pour le bouton Open à la partie Modèle et en vérifiant le comportement du programme enc as de detection d'erreur : on vide les structures de données et on ne quitte pas le programme. Ensuite, enc as de lecture avec succès, on peut tester la sauvegarde de fichier et comparer le fichier sauvegardé avec le fichier lu (on accepte les differences de presentation).

A ce stade il faut tester des séquences de lecture/sauvegarde à partir de l'interface graphique, avec et sans erreur car c'est de cette manière que votre projet sera testé.

2.3.2 Conversion des coordonnées dans gui et dessin avec le module graphic

Le sous-système de visualisation apportera une aide précieuse pour la vérification du programme.
Les taches sont réparties comme suit:

- la méthode **on-draw**() de l'interface gérée par le module **gui** effectue la conversion de coordonnées au niveau de ce module **gui**. Ce module doit mémoriser les informations sur la fenêtre (width et height) et sur le cadrage du Modèle (valeurs Min et Max selon X et Y). On mettra en oeuvre l'approche proposée en cours avec les transformation **translate** et **scale**. Enfin cette méthode **on_draw** doit mettre en oeuvre *l'absence de distorsion* quand on change la taille de la fenêtre (cf cours semaine6). Dans sa version complète la méthode **on_draw**() demandera au Modèle de se dessiner en appellant une seule méthode offerte par le module **jeu.**

- La méthode de dessin du module **jeu** délègue autant que possible la tâche du dessin aux autres modules du Modèle selon les indications de la *section 6 de la donnée générale*. Ces modules demandent à **tools** de dessiner des cercles, lignes comme précisé ci-dessous. Comme indiqué dans la section 7 de la donnée, l'interface de tools contient aussi l'interface de graphic, ce qui permet de préciser des paramètres de style de dessin comme la couleur désirée.

- Le module indépendant **tools** doit être complété pour proposer une ou plusieurs fonction(s)/méthode(s) pour dessiner des cercles, ligne.

- **tools** doit transférer les demandes de dessin de bas niveau au module **graphic** qui est le seul à avoir le droit de faire des appels directs à GTKmm

- le module **graphic** effectue les appels à GTKmm pour définir la couleur et dessiner des lignes, cercles et carrés. Il est aussi partiellement fourni.

En cas de lecture de fichier avec erreur (ou de lancement du programme sans fournir de fichier de configuration pour le rendu3), le jeu est ré-initialisée à « vide ». La méthode de dessin de jeu ne dessinera rien.

2.3.2.1 ACTION : test du module projet avec initialisation de l'interface graphique (avec dessin)

Dans une première étape, on laisse le Modèle de coté et on se contente de vérifier que la méthode **on_draw**() est capable de mettre en place la conversion de coordonnée et l'absence de distorsion en appellant une fonction du module **graphic** qui dessine un cercle ayant la taille l'arène. Ce cercle doit rester un cercle quand on change la taille de la fenètre.

Une fois que ces aspects sont maîtrisés au niveau de **gui**, **graphic** et **tools**, vous pouvez commencer à écrire et tester la méthode de dessin du Modèle. Le niveau **jeu** délègue aux modules inférieurs la tâche de se dessiner et eux-mêmes délèguent cette tâche au module **tools** en lui passant les paramètres de style et d'indice de couleur pour représenter les différentes entités.

3. Evaluation du rendu2 :

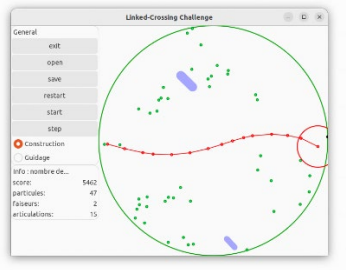
Le barème détaillé est visible à la suite de la donnée générale. Nous effectuerons une évaluation manuelle de votre programme SANS LE RELANCER :

- lecture avec succès, suivi d'une sauvegarde, suivi d'une lecture différente avec succès, suivi de la lecture du fichier sauvegardé, etc…
- lecture avec succès, suivi d'une sauvegarde, suivi de l'utilisation des autres boutons, suivi de la lecture du fichier sauvegardé, etc…
- lecture avec échec, lecture avec succès, lecture avec échec, etc…

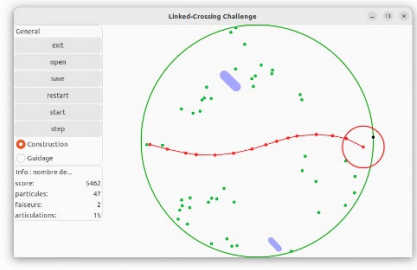
Dans le cas d'une lecture avec échec les structures de données doivent être détruites pour ne pas perturber la lecture suivante. L'affichage doit alors montrer un espace blanc à droite de la colonne des boutons.
Le fonctionnement attendu des boutons est décrit en section 2.3.1.

- **Affichage** :

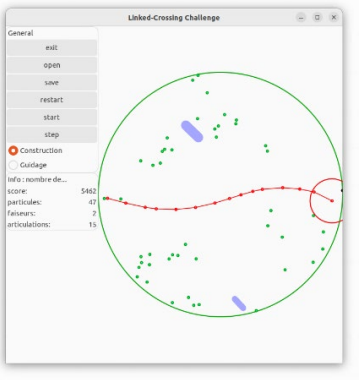
Les proportions, formes et couleurs des éléments du jeu doivent être respectées. L'absence de distorsion est à mettre en oeuvre dès ce rendu2 ; voici un exemple avec le fichier **t21.txt** pour lequel l'arène est centrée dans l'espace libre pour le dessin. On peut choisir un autre choix de cadrage du moment qu'on n'introduit pas de distorsion:



Fenêtre avec la taille initiale



Agrandissement horizontal



Agrandissement vertical

- **Déroulement partiel du jeu** :

On lancera l'exécution du jeu avec Start/Stop/Step. Les particules et faiseurs présents devront se déplacer conformément aux indications de la donnée générale et les particules devront se dupliquer ou se détruire. Les labels de la colonne de gauche devront être mis à jour pour le score et les nombres d'entités.

- si le fichier contient une chaine celle-ci sera dessinée et restera immobile quand on lancera l'exécution du jeu. De même l'interaction chaine-faiseur ne sera pas évaluée dans ce rendu2.

4. Forme du rendu2

Convention de style : il est demandé de respecter les conventions de programmation du cours.

- On autorise *une seule* fonction/méthode de plus de 40 lignes (max 80 lignes) car le module gui est fourni et respecte déjà la consigne de longueur des fonctions/méthodes.

Documentation : l'entête de vos fichiers source doit indiquer les noms des membres du groupe, etc.

Activité individuelle (max 0.5 page à la fin du rapport/pdf) de chaque membre selon ce message edstem décrivant la méthode de travail recommandée.

Rendu : pour chaque rendu **UN SEUL membre d'un groupe** (noté **SCIPER1** ci-dessous) doit téléverser un fichier **zip**[1] sur moodle (pas d'email). Le non-respect de cette consigne sera pénalisé de plusieurs points. Le nom de ce fichier **zip** a la forme :

SCIPER1_ SCIPER2.zip

Compléter le fichier fourni **mysciper.txt** en remplaçant 111111 par le numéro SCIPER de la personne qui télécharge le fichier archive et 222222 par le numéro SCIPER du second membre du groupe.

Le fichier archive du rendu2 doit contenir (**aucun répertoire**) :

- Rapport (fichier pdf)
- Fichier texte édité **mysciper.txt**
- Votre fichier **Makefile** produisant un executable **projet**
- Tout le code source (.cc et .h) nécessaire pour produire l'exécutable.

*On doit obtenir l'exécutable **projet** après décompression du fichier **zip** en lançant la commande* **make** *dans un terminal de la VM (sans utiliser VSCode).*

Auto-vérification : Après avoir téléversé le fichier **zip** de votre rendu sur moodle (upload), récupérez-le (download), décompressez-le et assurez-vous que la commande **make** produit bien l'exécutable lorsqu'elle est lancée dans un *terminal de la VM* (sans utiliser VSCode) et que celui-ci fonctionne correctement.

Exécution sur la VM: votre projet sera évalué sur la VM à distance (compilation avec l'option -std=c++17).

Backup : Il y a un backup automatique sur votre compte myNAS.

Debugging : Visual Studio Code offre un outil intéressant pour la recherche de bug (VSCode tuto).

[1] Nous exigeons le format zip pour le fichier archive