Projet Informatique – Sections Electricité et Microtechnique

Printemps 2023 : *Mission propre en Ordre* © R. Boulic & collaborators

Rendu1 (2 avril 23h59)

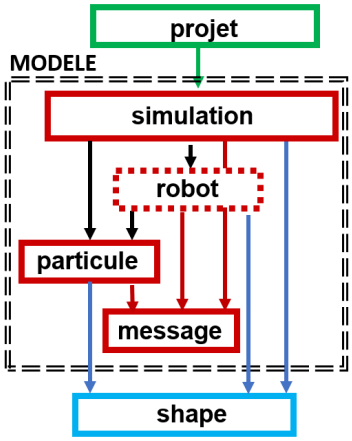
Objectif de ce document : *Ce document utilise l'approche introduite avec la série théorique sur les méthodes de développement de projet qu'il est important d'avoir faite avant d'aller plus loin.*

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 Fig 9a).

1. Buts du rendu1 : mise au point de shape et lecture de fichier

Le *premier* objectif est de disposer d'un module **shape** dont chaque fonction est validée par des test extensifs (avec du *scaffolding et un test unitaire de ce module*). Le but est de disposer d'un outil stable pour les rendus suivants. Cela implique de définir en priorité les structures de données, assez simples, de **shape** puis les fonctions que ce module va offrir dans son interface. Nous demandons d'y créer le type **S2d** (Donnée section 7.3.3) et les structures de données pour les carrés et cercles.

C'est seulement après la validation du module **shape** que vous pourrez aborder le second objectif d'ébauche du **Modèle** que vous voyez dans la boite en pointillés au dessus du module **shape** à droite. Ce Modèle doit contenir l'ensemble des modules et des dépendances visibles dans la figure ci-contre.



Donnée Fig 9a

N'utilise PAS GTKmm

Nous imposons que les modules du **Modèle** mettent en œuvre des **classes** en respectant le *principe d'encapsulation*, ce qui veut dire que les *attributs* seront **private** et qu'il faudra utiliser des *méthodes* pour y accéder. A part cette contrainte stricte, nous acceptons pour ce premier rendu que votre choix de structure de donnée soit une ébauche qui pourra être remise en question pour les rendus suivants.

Pour le rendu1, le module de plus haut niveau **projet** contient seulement la fonction **main** : sa tâche est de récupérer le *nom de fichier de test* transmis sur la ligne de commande au moment du lancement de l'exécutable. Ce nom de fichier doit être immédiatement transmis à une fonction ou méthode du module **simulation** qui agit comme point d'entrée du **Modèle**.

Les responsabilités des modules du Modèle (Donnée section 7.2) sont complétées ici :

- **simulation** : c'est le SEUL point d'entrée du Modèle vis-à-vis de l'extérieur (module **projet** pour le rendu1)
  - il gère au plus haut niveau d'abstraction les tâches de (*une*) mise à jour de la simulation, dessin, lecture et écriture de fichier. Pour le **rendu1**, on demande seulement de mettre au point l'action de **lecture** d'un fichier pour initialiser une simulation en détectant des erreurs prédéfinies.
  - en vertu du principe d'abstraction, ce module délègue l'exécution des sous-problèmes de ces tâches auprès des modules qui gèrent les entités de **particule** et de **robot** (cf Fig9a).

- **robot** : pour le rendu final il est demandé de mettre en place *une hiérarchie de classes* pour gérer les différents types de robots. Cependant, pour le **rendu1**, une approche simplifiée sera acceptée avec un attribut de *type* dans une classe unique **robot**.

- **particule** : toutes les actions liées à ce type d'entité sont gérée dans une classe (lecture et stockage, sauvegarde, test de collision, , désintégration…).

- Le module **message** est *fourni* pour l'affichage de messages standardisés pour la tâche de lecture du Modèle. Il ne faut pas le modifier.

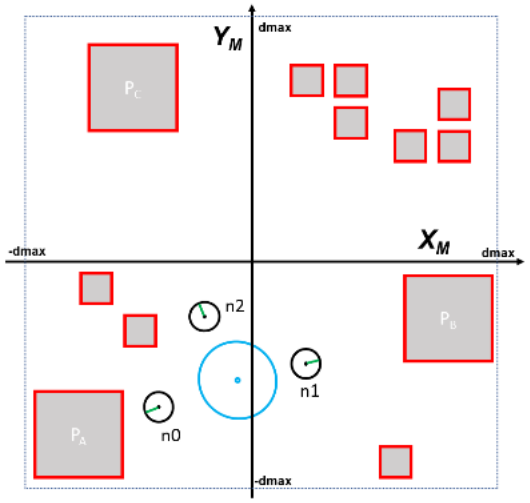
La section suivante précise ce qui est demandé pour le module **shape** et comment le tester.

2. But du module shape

Ce module définit le type **S2d** pour modéliser un point ou un vecteur dans le plan (Donnée section 7.3.3). Il s'en sert pour définir les types pour modéliser un carré et un cercle dans un espace continu en virgule flottante double précision et selon les convention d'axes visibles à droite.

Ce module sert à gérer des carrés et des cercles et offrir des fonctions de:
.Test de superposition de deux carrés
.Test de superposition de deux cercles
.Test de superposition d'un carré et d'un cercle

Ces fonctions renvoient un booléen **true** en cas de détection de superposition.



Rendu1 Fig1 : conventions d'axes de coordonnées X et Y

Seule la constante **epsil_zero** est définie par ce module pour gérer deux variantes de détection de superposition des entités. Les autres constantes sont définies par le Modèle et ne doivent pas apparaître dans **shape**.

3. lecture de fichiers de configuration

3.1 Module projet

Le module **projet** contient la fonction **main()** en charge d'analyser si la ligne de commande est bien conforme à la syntaxe suivante : `./projet t01.txt`

Où `t01.txt` est un fichier de configuration (Donnée **Section 4**).

Nous ne testerons pas l'absence de l'argument. Pour le rendu1, votre programme doit se terminer en cas d'absence d'argument. Si l'argument est présent nous supposons qu'il correspond à un fichier de test présent dans le répertoire courant. Ce nom de fichier doit être transmis à une fonction ou méthode de **lecture** du module **simulation.**

Deux stratégies sont autorisées concernant *l'instance* de la classe **Simulation** qui pilote le Modèle :

- Soit elle existe dans le module **projet** et une méthode **lecture** est appelée sur cette instance.

- Soit elle est cachée dans le module **simulation** (car elle est unique) et une fonction **lecture** du module simulation est appelée sans avoir besoin de préciser en paramètre sur quelle instance elle travaille.

3.2 Liste des erreurs à détecter au niveau du Modèle (Donnée section 4.2):

Le programme cherche à initialiser l'état du **Modèle** avec les données lues dans le fichier de configuration. Pour le Rendu1 il **s'arrête** dès la ***première*** erreur trouvée dans le fichier en appelant la fonction mise à disposition pour l'affichage du message d'erreur puis on quitte le programme en appelant **exit(EXIT_FAILURE)**.

Le programme **s'arrête** aussi après la lecture du fichier s'il n'y a aucune erreur ; dans ce cas, il faut appeler la fonction **success()** du module message qui affiche un message indiquant le succès de la lecture puis on quitte en appelant **exit(0**).

Voici la liste des verifications à effectuer au niveau du Modèle

a) Le coté d'une particule doit être supérieur ou égal à **d_particule_min**
b) Le robot spatial et les particules doivent être entièrement à l'intérieur du domaine **[-dmax, dmax]**
c) Les particules ne doivent pas se superposer entre elles
d) Les robots ne doivent pas se superposer entre eux (excepté pour le robot spatial pour lequel les superpositions avec les autres robots sont autorisées)
e) L'attribut **k_update_panne** d'un neutraliseur doit être inférieur ou égal à **nbUpdate**
f) Absence de superposition entre un robot et une particule

Nous ne testerons pas vos projets sur la longueur des lignes de fichier ni sur d'autres éventuelles incohérences.

Les formules des tests de collisions sont décrites dans la donnée en section 2.2. Elles sont utilisées pour tester les superpositions éventuelles au moment de la lecture de fichier MAIS la constante **epsil_zero** doit être ignorée (c'est à dire nulle) pour ce contexte de lecture de fichier (rendu1). Cela veut dire que vos fonctions du module **shape** qui font les tests de superposition doivent prendre en compte un paramètre supplémentaire qui active ou désactive l'utilisation de epsil_zero dans les formules.

3.3 Utilisation du module message :

Le module **message** est fourni dans un fichier archive sur moodle ; il ne doit pas être modifié. Son interface **message.h** détaille l'ensemble des fonctions à appeler. Comme pour la détection d'erreur au niveau de **shape**, il faudra utiliser les messages de cette façon :

```
if(une détection d'erreur est vraie)
{
      cout << message::appel_de_la_fonction(paramètres éventuels);
     std ::exit(EXIT_FAILURE) ; // Rendu1
}
```

A partir du rendu2, il ne faudra pas quitter le programme mais renvoyer un booléen d'échec pour les cas d'échec et un booléen de succès quand la lecture de fichier et toutes les validations ont été effectuées avec succès. Dans ce cas, c'est la fonction **success()** qu'il faut appeler.

Le message affiché par ces fonctions se termine par un passage à la ligne. Il ne faut pas en ajouter un.

3.4 Méthode de travail

Plusieurs approches sont possibles ; utilisez le document *méthodes de développement de projet* comme guide. Au niveau de l'exécution, nous fournissons un nombre limité de fichiers de tests sur lesquels votre programme sera évalué. Le succès de ces tests ne peut pas garantir l'absence de bugs pour d'autres fichiers de tests. Donc, commencez à *organiser votre propre batterie de tests* indépendamment de ce que nous mettons à disposition.

3.4.1 ACTION : test unitaire de shape

Pour effectuer le test unitaire d'un module vous devez écrire un programme de test (*scaffolding*) qui effectue des appels de toutes les fonctions offertes par le module et ce programme compare le résultat renvoyé par chaque appel au résultat attendu (que vous avez calculé indépendamment par un autre moyen).

A partir des indications de ce rendu1 décidez le nom des fonctions et leur prototype et mettez les fonctions exportées dans **shape.h**. Une fois cela fait on peut inclure **shape.h** dans un programme de test pour tester chacune des fonctions de **shape.cc**. C'est votre responsabilité de trouver un nombre suffisant d'exemples pertinents pour s'assurer qu'on n'oublie pas de cas particuliers.

Il est essentiel d'effectuer ce test unitaire sur **shape** avant d'utiliser ses fonctions dans le Modèle.

3.4.2 ACTION : test du Makefile de l'architecture du rendu1

A partir du dessin de l'architecture du rendu1 (page 1, Fig9a de la Donnée) en déduire les dépendances et écrire le fichier Makefile. Testez-le avec des modules contenant le minimum pour être compilable et exécutable, c'est-à-dire les includes et, au plus haut niveau, une fonction main vide ou simplement avec affichage d'un message.

3.4.3 ACTION : tests du module simulation

Au stade du rendu1, seul le constructeur et la fonction/méthode de lecture de fichier sont nécessaires. Dans ce module l'opération de lecture met en place *l'automate de lecture* (cours Topic3) qui filtre les lignes inutiles du fichier et délègue l'analyse fine de lecture d'une ligne aux autres modules plus spécialisés (particule, robot) qui ont la responsabilité de faire les vérifications nécessaires. L'automate peut être testé avec des *stubs* de ces fonctions/méthodes plus spécialisées de lecture qui renvoient toujours true pour indiquer que le décodage de la ligne de fichier s'est bien passé.

A ce stade l'intégration avec le module projet est immédiate puisqu'il suffit de remplacer le stub de la fonction/méthode de lecture du fichier par un appel de celle mise au point pour le module simulation.

3.4.4 ACTION : tests du module particule

Déclarez un attribut de type carré offert par **shape** pour décrire l'espace occupé par une particule. Au stade du rendu1, seul le constructeur et la fonction/méthode de décodage d'une ligne lue dans un fichier sont nécessaires.

Il est cependant nécessaire de mémoriser les particules dans une structure de donnée pour pouvoir immédiatement tester si une nouvelle particule lue se superpose à une particule déjà lue. Cet ensemble peut, au choix, être caché dans le module particule (ex : un vector de particules déclaré dans l'espace de noms non nommé) ou être un attribut de la classe qui gère une simulation.

3.4.5 ACTION : tests du module robot

Les robots peuvent être représentées par une seule classe à l'aide d'un attribut de type mais cela n'est accepté que pour le rendu1 ; il faut proposer une hiérarchie de classes (2 niveaux suffisent) pour les rendus 2 et 3. Indépendamment de votre choix pour le rendu1, déclarez un attribut de type cercle offert par **shape** pour décrire le l'espace occupé par un robot. Ecrivez du code de scaffolding pour initialiser des instances des différentes types de robots et vérifier la valeur des attributs avec une méthode d'affichage dans le terminal.

Ensuite seulement écrivez la méthode qui décode la ligne du fichier pour chaque type de Robot. Testez cette méthode en transmettant une ligne avec la syntaxe du fichier et vérifiez avec la méthode d'affichage que le décodage s'est bien passé. Ensuite intégrez de proche en proche avec les niveaux supérieurs.

4. Forme du rendu1

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)

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

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 rendu1 doit contenir (**aucun répertoire**) :

- 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** en lançant la commande* **make** *après décompression du fichier **zip**.*

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 et que celui-ci fonctionne correctement.

Exécution sur la VM: votre projet sera évalué sur la VM à distance.

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

Debugging : Visual Studio Code offre un outil intéressant pour la recherche de bug (tutorial du sem1).

Gestion du code au sein d'un groupe :

- **Solution simple mais limitée** : créer un répertoire sur **gdrive.epfl.ch** qui est partagé seulement par les 2 membres du groupe. Cependant il n'y a pas d'éditeur de code en mode partagé.
- **Solution plus ambitieuse mais qui demande un apprentissage non négligeable** : créer un compte sur gitlab.epfl.ch . A partir de ce compte vous créez un projet ; .Attention : il FAUT **restreindre** l'accès du code aux seuls 2 membres du groupes sinon il est public par défaut. Ensuite il faut utiliser l'outil **git** avec **gitlab** comme expliqué dans cette video YouTube

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