Virtual Hadron Collider
Conception

Jakob Odersky Christian Vázquez

May 29, 2011

Introduction

Lors du projet nous avons suivi en général les indications données. Quelques fois, nous avons tout de même suivi nos propres idées qui n'étaient pas compatibles avec les propositions officielles. Dans ce texte sont décris en bref nos propres idées et leur implication sur la conception du projet. Pour une documentation plus complète est détaillée, il est conseillé de voir les commentaires du code ou la documentation générée par les commentaires du code.

Structure

Le dosier du projet, nommé 'vhc' (pour 'Virtual Hadron Collider'), est structuré de la manière suivante:

```
vhc
|-- bin
|    |-- gui
|    |-- main
|    `-- test
|-- doc
|-- Doxyfile
|-- JOURNAL.txt
|-- Makefile
|-- CONCEPTION.pdf
|-- REPONSES.pdf
|-- README.txt
`-- src
     |-- gui
     |-- main
     `-- test
```

vhc

Le répertoire principale (dit 'de base') du projet est est 'vhc'. Dans celui-ci se trouvent les fichiers expliqués dans la description officielle du projet. Notamment:

- JOURNAL
- REPONSES
- README
- Makefile

Il contient de plus le fichier 'Doxyfile' utilisé pour générer de la documentation du code source.

src

Ce répertoire contient tout le code source, donc les fichiers les plus importants du projet! Le code source est lui-meme reparti dans les sous-répertoires suivantes:

- main
  Contient le code source principale, c'est à dire tout les fichiers sources du simulateur.
  Exemples: Vector3D.cc, Vector3D.h, Particle.cc, etc...
- gui
  Contient le code relatif à l'interface graphique.
- test
  Contient le code source des tests. Les fichiers tests contenant une fonction 'main' devraient se terminer avec 'Test'.
  Exemples: Vector3DTest.cc, etc...

doc

Contient de la documentation générée automatiquement par un outil comme 'doxygen'.

bin

Contient les fichiers binaires (i.e. executables, objets, librairies etc...) compilés du code source. La structure de ce répertoire est identique a celle de 'src', c'est-à-dire que les tests seront compilés dans 'bin/test/' et les sources principaux dans 'bin/main/'.

Makefiles

Afin d'automatiser le processus de compilation, un Makefile est présent dans le répertoire de base. A cause de la compléxité du répertoire source, le Makefile est récursif. Cela signifie que ce Makefile ne fait que de déléguer les commandes à deux makefiles contenus dans les répertoires src/main et src/test. Ainsi, lorsqu'on ajoute/supprime des fichiers des répertoires précédents, il suffit de modifier le Makefile contenu dans le répertoire respectif. En général, il ne faut pas modifer le Makefile de base.

Voici les commandes make de base:

- clean - supprime les fichiers générés
- build - compile les sources principaux
- test - lance tous les tests
- gui - compile et lance l'interface graphique
- doc - genere la documentation avec doxygen

Pour plus d'informations voir les commantaires des Makefiles.

Vecteurs

En commencant par la base. Nous avons choisi de garder un vecteur immuable. Il n'a donc pas de méthodes non-const. Ceci parait naturel pour un vecteur et facilite le raisonement sur la vie d'une de ses instances.

Pour des raisons d'efficacité, nous avons tout de même implémenté un vecteur muable. Celui-ci hérite d'un vecteur et défini en plus des méthodes non-const (tel que +=).

Eléments

La hierarchie des éléments ressemble à celle proposée par la description officielle du projet:

Tout en haut il y a un élément abstrait *Element*. De celui-ci héritent les différents éléments géométriques: un droit *StraightElement*, un courbé *CurvedElement* et un élément composé *CompositeElement*. Les éléments concrets sont donc définis en héritant d'un de ces trois éléments.

Pour considérer les champs électriques et magnétiques, nous avons simplement déclaré des méthodes virtuelle *getElectricFieldAt(const Vector3D& position)* et *getMagneticFieldAt(const Vector3D& position)* dans la classe de base *Element* qui renvoyent zéro pour tout position. Les élements concrets peuvent donc redéfinir ces méthodes.

Un autre aspect assez important est la détermination de la contenance d'un particule dans un élément. Comme nous avons décidés que nos éléments peuvent être décentrés (pas de centre commun avec l'accélérateur) la méthode de détermination de passage d'une particule, telle que proposée ne fonctionne plus. Nous avons donc décidé de créer un système de positionement des particules dans les élements. Une particule peut être testée si elle est après, avant ou a côté d'un élément. Ceci facilite aussi la possibilité des particules de tourner à contre-sens: ces particules n'ont qu'a vérifier si elles se situent avant leurs éléments respectifs pour passer aux suivants.

Faisceaux

En créant un faisceau, l'utilisateur n'a pas besoin de se soucier dans quelle élément se trouve une particule. Il ne doit que l'ajouter à un accélérateur et il s'occupe du reste.

Cette simplicité est possible grâce a notre conception de faisceaux. Chaque faisceau à une méthode *initializeParticles()* qui initialise les particules du faisceau. Ainsi, lorsqu'on ajoute un faisceau à l'accélerateur, celui-ci appelle la méthode d'initialisation des faisceaux, positionne les particules dans leurs éléments respectifs et supprime tous ceux qui ne sont pas contenus dans un élément.

Accélérateur

Un accélérateur est l'objet principale d'un simulation. Il contient tous les éléments et faisceaux de la simulation. Afin de faciliter son utilisation, nous avons décidés q'un accélérateur contiendrait et sera responsable pour toutes ses composantes. Ainsi lorsqu'un élément ou une particule est ajoutée à un accélérateur, celle-ci est copiée dedans.

Pour garantir la validité d'un accélérateur, celui-ci est "fermé" avant une simulation. Concrètement cela veut dire qu'après avoir ajouter des éléments dans un accélérateur, une méthode est appelée pour vérifier sa continuité.

Forces inter-particules

Pour ajouter les forces interparticules à la simulation nous avons instauré la notion d'*interacteur*. Un interacteur (classe abstraite *Interactor*) est un objet qui gère les interactions entre particules. Chaque accélérateur en a un. Concrètement un interacteur possède une méthode *applyInteractions()* qui applique tous les forces interparticules entre les particules gérés. Or comme les particules d'un accélarateur varient sans cesse et sont souvant supprimés, il faudrait un système plus efficace pour connaître les particules à gérer que d'accéder directement aux

particules de l'accélérateur. Pour ceci, nous avons introduits le design pattern (motif de programmation?) des publishers/listeners.

Un *publisher* est une classe qui peut publier des évènements et un *listener* peut agir à la récéption d'un évènement. De plus, un listener doit se souscrire à un publisher pour être informé de ses évènements. Ainsi, dans notre projet, un faisceaux est un publisher et un interacteur un listener. Ceci implique que lorsqu'une particule est ajoutée à un faisceau, l'interacteur est informé et peut ajouter la particule à la collection des particules à gérer.

Cette conception donne de nombreux avantages. Premièrement, il est facile de changer de type de détecteur. On peut par exemple remplacer un détecteur force-brute avec un détecteur plus sophistiqué en une ligne de code. Deuxièmenent, il est très facile de concevoire ses propres détecteurs. Il suffit de créer une classe héritant de *Interactor*. Troisièmement, grâce au motif publisher/listener, on peut créer un interacteur qui choisit les particules à gérer. Ceci permet doc de facilement transformer un interacteur gérant tous les interactions particules en un interacteur ne gérant que les interactions particules dans un faisceau.

Interface graphique

Notre projet utilise OpenGL et Qt pour l'interface graphique. Pour ne pas rendre l'entièreté du projet dépendant de ces deux frameworks nous avons utilisé une autre méthode que celle de la "spécialisation graphique" proposée.

Nous avons créés des rendeur graphiques (*Renderer*). Ces rendeurs sont spécialisés pour dessiner des objets d'un certain type. Un rendeur de faisceaux, par exemple, peut dessiner des faisceaux. Or, pour des hiérarchies de classes nécéssitant un traitement différent pour chaque type concret d'instance (par exemple les éléments: un élément droit ne peut pas être déssiné comme un élément courbe!), cette méthode n'est pas très élégante et nécéssite une quantité énorme de code.

Pour contourner cela nous avons donc implémenté le motif de programmation des visiteurs. C'est "une manière de séparer un algorithme de la structure d'un objet"[1] et s'avère particulièrement utile pour des structures récursives, telles que les éléments composés. Concrètement cela signifie qu'un rendeur d'éléments est un visiteur d'éléments. Voir les commentaires du code pour plus de detailles.

Conclusion

En conclusion, on remarque que nos déviations du projet ne sont pas dues à cause d'un incompréhension où une difficulté à suivre ce qui était demandé, mais uniquement pour ajouter de la fonctionalité et de faciliter l'extension de notre projet par des tiers.

[1] (http://fr.wikipedia.org/wiki/Visiteur_%28motif_de_conception%29)