SMILE :-)

Conception

module dynamique

SOMMAIRE

1. Introduction

Dans le cadre du projet OPALE, et plus particulièrement en ce qui concerne le projet SMILE, l'ensemble des informations concernant la conception se trouve dans le présent document. Cependant pour des raisons de clarté et de facilité d'accès, les documents de conceptions, tout comme les dossiers de spécifications, sont répartis en 3 documents distincts... Vous avez entre les mains le dossier de conception concernant **le module dynamique**.

Ce document a été réalisé avec Openoffice[1], et les diagrammes avec dia[2]. Les numéros entre crochet font référence à un lien que vous trouverez en dernière page.

Ce document décrit l'architecture du module dynamique (alias SMILE) à différents niveaux.

- La première partie décrit son contexte, sa composition, et donne un diagramme de classe simplifié.
- La deuxième partie présente quelques fonctions.

2. Conception préliminaire

1. Le système dans son contexte

Le système pourra se déployer sur un réseau. En outre pour des raisons de performance il sera préférable de dédier un serveur pour le module Dynamique qui demandera beaucoup de ressources processeur.
Pour développer la première version du système nous avons choisis de faire communiquer les postes par le protocole tcp/ip en utilisant les sockets. De plus nous avons conçut les interfaces de manière à pouvoir utiliser un autre moyen de transport de l'information (IPC ou RPC par exemple). Mais quel que soit le choix du protocole, il faudra que l'échange puisse se faire sur un réseau hétérogène (Windows/Linux).

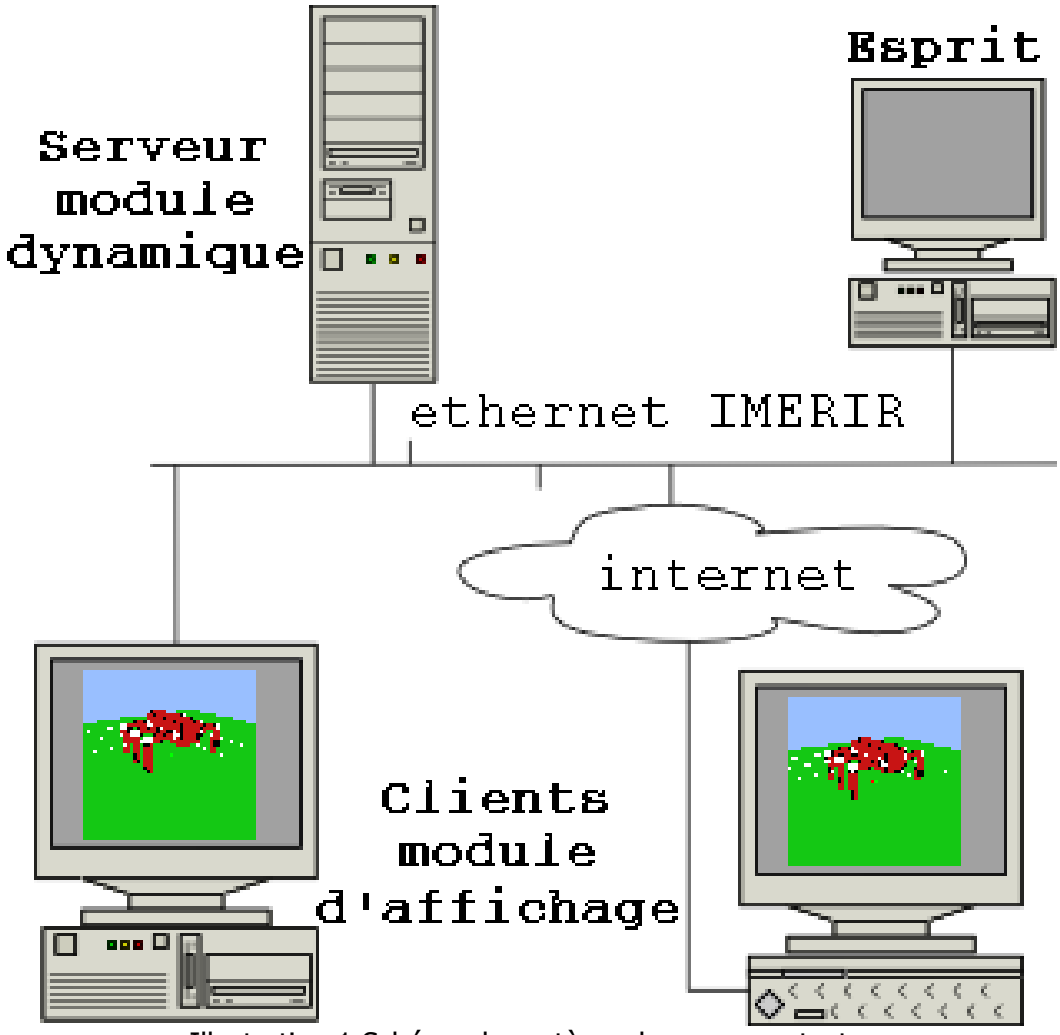

Illustration 1 Schéma du système dans son contexte

2. Les composants du système

Le système que nous développons cette année se décompose en trois parties:

- un module Opale
- un module Dynamique
- un module Affichage

Le schéma suivant illustre la répartition de chaque module et les relations qui les font communiquer.

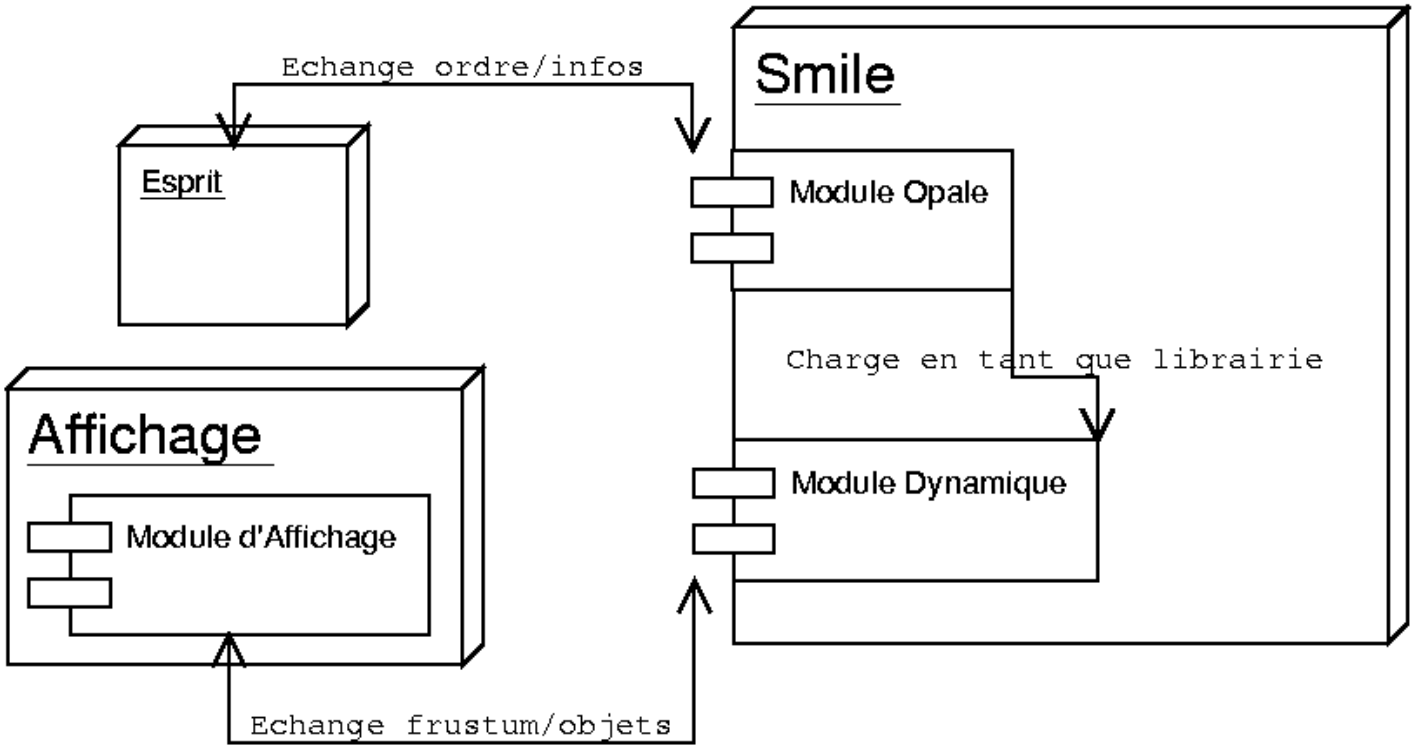

Illustration 2 Schéma des composants du système

3. Diagramme de classe du module dynamique

Nous présentons ici qu'un diagramme résumant les classes et leurs relations. Pour un diagramme de classe détaillé (avec les attributs et les méthodes) voyez en annexe de ce document.

Bien que les diagrammes soient à la norme UML, nous rappelons la légende ci-dessous.

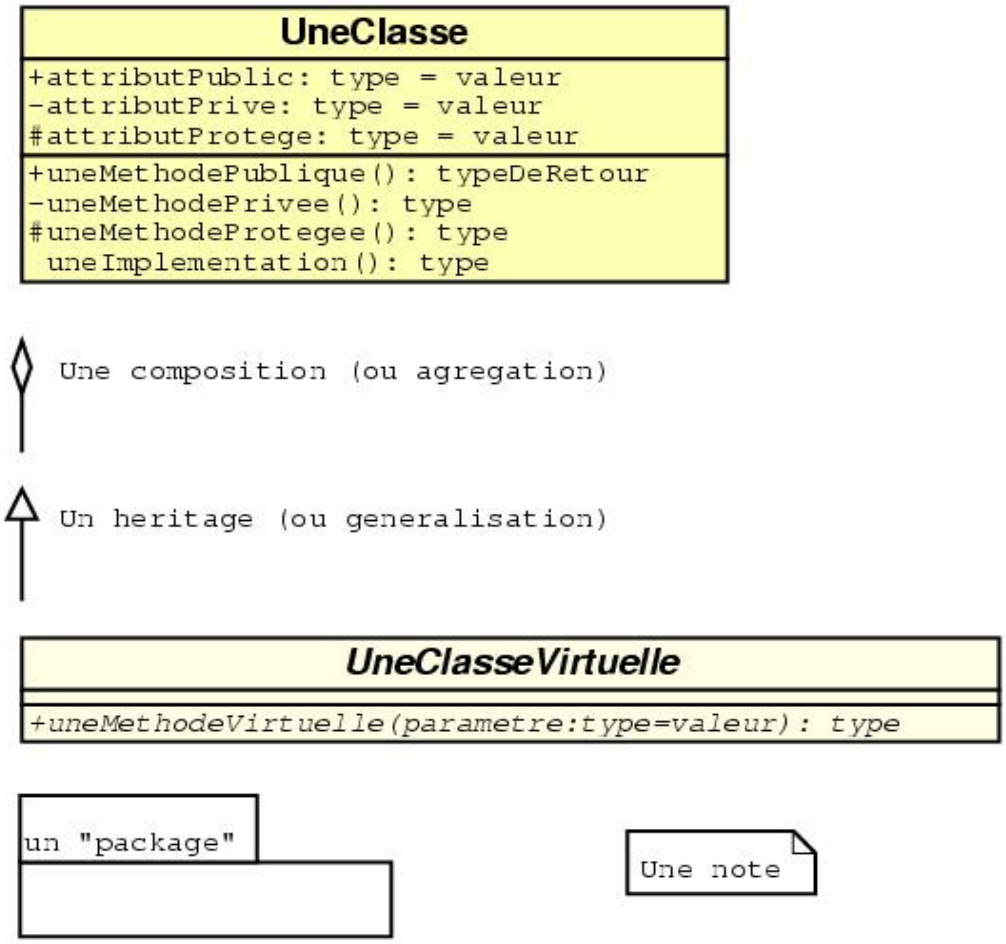

Illustration 3 Légende

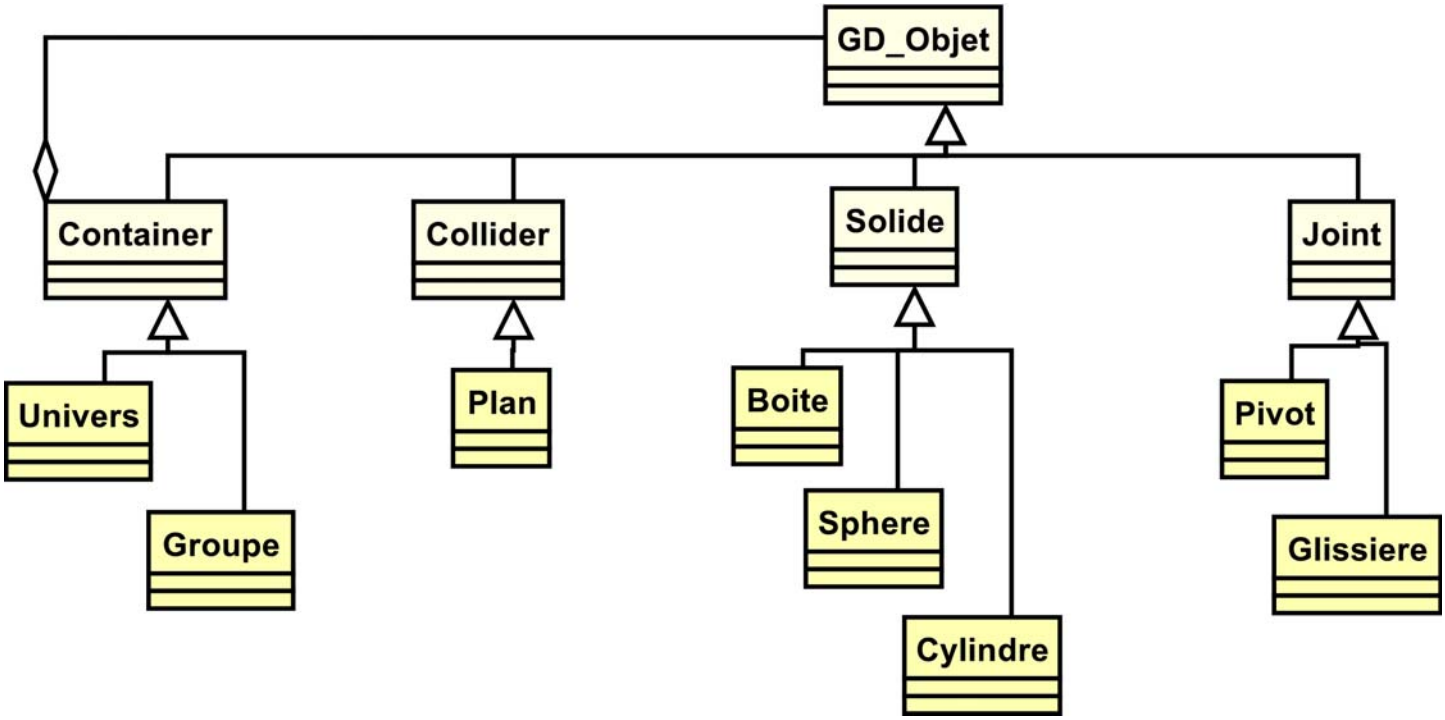

Illustration 4 Diagramme de classe.

2. Conception détaillée

1. Environnement de développement

Description de l'environnement de développement par ressources :

Systèmes d'exploitation:

Windows NT/2000/XP.

GNU/Linux.

Logiciels de développement:

Microsoft Visual C++

gcc

Bibliothèques :

Stl, ODE, OpenGL, Glut.

Matériels :

Les contraintes matérielles se situent sur les points ci-dessous :

Réseau éthernet / Internet (celui d'IMERIR)

Carte Graphique compatible OpenGL (pour le module graphique)

Ram (pour le serveur qui fera tourner le module dynamique)

Fréquence processeur élevée (pour le serveur qui fera tourner le module dynamique)

Humain :

l'équipe projet (5 personnes) + le tuteur.

Détail du choix de ODE :

Pour ce module, le cahier des charges proposait d'utiliser Vortex, une bibliothèque de simulation physique développée par CmLabs[3]. Malgré un compromis performance/prix réellement intéressant, notre budget nous a contraint à trouver une solution de remplacement moins coûteuse. Notre choix s'est donc porté sur O.D.E.[4], un moteur dynamique sous licence GPL. Bien que ce dernier soit encore en développement, il offre juste le minimum dont nous avons besoin pour créer SMILE. Enfin, O.D.E. propose une interface en c++ mais nous décidons d'utiliser l'interface en c que nous encapsulerons.

2. Description par classe

Classe GD_Objet :

Remarque : cette classe est à la base de toutes les autres.

Les constructeurs/destructeurs :
Ces méthodes initialisent les champs relatifs à la partie graphique (visibilité, couleur...) et à la partie dynamique (dimension, position, orientation), à des valeurs que le développeur jugera cohérentes.

Les accesseurs :
Ils permettrons d'accéder aux matrices de rotation, position, d'échelle qui permettent de décrire l'état dynamique de l'objet, ainsi qu'aux attributs relatifs à l'affichage (couleur par exemple).

Les autres méthodes :

`void moveOf(double x,double y,double z)`
Permet de déplacer l'objet par rapport à sa position actuelle.
mettre à jour la nouvelle position avec *setPosition()* en additionant l'ancienne position obtenue avec *getPosition()* et en y ajoutant les valeurs passées en parametres.

`void rotateOf(double angle,double x,double y,double z)`
Permet de tourner l'objet par rapport à son orientation actuelle.
mettre à jour la nouvelle orientation avec *setRotation()* en multipliant l'ancienne matrice d'orientation obtenue avec *getRotation()* à celle déduite des valeurs passées en parametres.

`bool isIn(frustum_t frustum)`
Retourne vrai si l'objet est à l'intérieur du frustum*. Cet algo est décrit dans [5]. Nous avons choisi de garder l'algo de base uniquement. Cet algo est donc largement améliorable. (AABB pour AxisAlignedBoundingBox)

générer la *aabb*
pour chaque plan qui compose le *frustum*
  prendre le vertex du aabb le plus proche du plan
  tester si le vertex est dans ou hors du frustum
  s'il est dehors retourner *false*
fpour
retourner *true*

void getProperty(property_t* property)
Permet d'obtenir les informations spécifiques à un objet. Par exemple pour une boite ce seront ses cotés...

void getProperty(property_t* property)
Cette methode utilise les methodes getType(), getCouleur([...]), getPosition([...]), getRotation([...]) et getEchelle([...]) afin de compiler le tout dans une structure qui pourra etre communiqué par réseaux ou servir à l'affichage local du GD_Objet.

Classes Container : Univers et Groupes

Les constructeurs/destructeurs .

Les constructeurs doivent initialiser les pointeurs à zéro. Ces classes prennent un autre Container en argument, celui-ci devant etre mis à jour (grace à la methode addGDObjet([...])) lors de la construction. Nous devons faire attention au fait que seul un Univers peut etre instancié sans container puisque il n'est contenu par aucune autre classe.

Les accesseurs .

Les autres méthodes :

-

`void addGD_objet(GD_Objet* gdobjet)` `void remGD_objet(GD_Objet* gdobjet)`
Ajoute ou enlève un GD_Objet dans la liste.
`Ajouter/retire l'objet gdobjet à la liste gd_objets.`

`List<objet_t> getObjetIn(frustum* frustum)`
Retourne une liste des caracteristiques de chaques objet dans le frustum passé en parametre..
`Utilise la methode isIn(), puis getProperty() de chaque gd_objet de la liste.`

-

`void simLoops(double step, int repeat)`
Cette méthode est spécifique à la classe Univers Fait avancer de 'step * repeat' millisecondes la simulation

`repeter{`
`  faire avancer le monde d'un pas.`
`}`

`void nearCallback(void *data, dGeomID geom1, dGeomID geom2)`
Cette fonction est spécifique à la classe Univers Voir la documentation de ODE [6]

Classes Collider : Plan

Les constructeurs/destructeurs :

Il ne peut y avoir de constructeur par défaut. Les constructeurs prendrons impérativement en paramètre les propriétés qui caractérisent l'objet (container et dimmensions). Le programmeur pourra attribuer une valeur par défaut à ces derniers en veillant rigoureusement à ce que le pointeur vers le container ne soit pas nul.

Les accesseurs :

Il faudra implémenter les methodes virtuelles (position, orientation, echelle) de la classe GD_Objet.

Les autres méthodes :

Classes Solide : Boite, Sphère et Ccylindre

Les constructeurs/destructeurs :

Il ne peut y avoir de constructeur par défaut. Les constructeurs prendrons obligatoirement en paramètre les propriétés qui caractérisent l'objet (container, dimmensions et masse). Le programmeur pourra attribuer une valeur par défaut à ces derniers en veillant rigoureusement à ce que le pointeur vers le container ne soit pas nul.

Les accesseurs :

Il faudra implémenter les methodes virtuelles (position, orientation, echelle) déclarées dans la classe GD_Objet.

Les autres méthodes :

Classes Joint : Charnière

Les constructeurs/destructeurs :

Il ne peut y avoir de constructeur par défaut. Les constructeurs prendrons obligatoirement en paramètre les propriétés qui caractérisent l'objet (container et objets à lier). Le programmeur veillera rigoureusement à ce que le pointeur vers le container ne soit pas nul, et qu'au moins un objet sur deux soit spécifié.

Les accesseurs :

Il faudra implémenter les methodes virtuelles (position, orientation, echelle) déclarées dans la classe GD_Objet.

Les autres méthodes :

`void setForce(double force)`
Elle permet d'imprimer une force.
`faire appel aux fonction de ODE pour les joints.`

`void setVel(double force)`
Elle permet de fixer une vitesse à atteindre lorsqu'on applique une force.
`faire appel aux fonction de ODE pour les joints.`

-

`void setHiStop(double angle)` `void setLoStop(double angle)`
Ces méthodes sont spécifiques à la classe Joint Elles permettent de fixer les butées d'une charnière.
`faire appel aux fonction de ODE pour les joints.`

Voir la documentation de ODE[6] pour plus de précision.

4. Références :

[1] Site internet de OpenOffice :
http://fr.openoffice.org/

[2] Site internet de dia :
http://www.lysator.liu.se/~alla/dia

[2]: Cahier des charges du projet Opale
/Projets/Projets2A/Opale2A/documentation/cdch_v1.2.doc

[3] Site internet de CMLabs :
http://www.CM-Labs.com

[4] Site internet de ODE :
http://opende.sourceforge.net/

[5] Référence pour l'algorithme d'intersection d'un frustum avec un AABB :
/Projets/Projets2A/Opale2A/documentation/assarssonXXoptimized.pdf

[6] Documentation de ODE :
http://opende.sourceforge.net/ode-latest-userguide.html#ref54

5. Table des illustrations

6.Annexe

Nous fournissons ici un diagramme de classe reprenant tous les attributs et toutes les méthodes à implémenter, en détail.