CNAM 2006-2007

LASER GAME 3D

LoD³

Nicolas AGIUS
Fabien DECUGIS

Juin 2007

Sommaire

Cahier des charges

Analyse et conception

LoD³ - Manuel d'utilisation

Annexes

LASER GAME 3D

Cahier des charges

1. Présentation générale

Le projet à réaliser consiste à créer un jeu en trois dimensions basé sur le principe du "Laser Game", c'est-à-dire un jeu où des joueurs s'affrontent à l'aide d'arme laser dans un labyrinthe à l'ambiance souvent sombre.

Le but du jeu réside à rester en vie pour pouvoir atteindre un objectif. Le jeu se déroule en réseau, chaque joueur agit sur sa propre console qui doit transmettre les informations sur l'autre console, et inversement.

Le nom du jeu est LoD³, à prononcer "LOD au cube". Il s'agit de l'acronyme de Laser of Double Dummy Death.

2. Spécifications

2.1. Les principales fonctionnalités du jeu

2.1.1. Type de jeu

Le jeu est un FPS (First Person Shoot) de type "Catch Flag" dans lequel s'affrontent deux équipes, un rouge et une bleue.

2.1.2. Environnement du jeu

Les deux équipes évoluent dans un labyrinthe (un seul niveau, pas d'étage ni de marche). Le labyrinthe ne dispose pas de plafond, pour permettre un éclairage de type solaire. Cet éclairage est de faible intensité (ambiance sombre) placé à la verticale du labyrinthe (pas d'ombre).

Le labyrinthe sera constitué de 4 éléments différents :

- Espace vide (passage)
- Mur (non franchissable)
- Base de l'équipe (lieu de *respawn*), un pour chaque équipe
- Socle du drapeau, un pour chaque équipe

Le plan du labyrinthe n'est pas disponible, le joueur doit lui-même mémoriser les chemins par lesquels il passe.

Dans un premier temps, le labyrinthe sera identique à chaque partie.

Un chronomètre affiche en temps réel la durée écoulée de la partie. Il n'y a pas de limite de temps pour gagner la partie.

2.1.3. But du jeu

Le but du jeu est d'attraper le drapeau de l'équipe adverse et de le rapporter à la base de son équipe.

2.1.4. Règles du jeu

Pour capturer le drapeau, le joueur doit simplement entrer en collision avec l'objet le représentant. Ce dernier disparaît de son socle. Le joueur ne peut prendre que le drapeau de l'équipe adverse.

Chaque joueur dispose d'une arme laser pour tirer sur l'adversaire. Lorsque qu'un joueur est touché par le laser adverse, il meurt. Le joueur est "mort" et ne peut plus joué pendant une durée de 5 secondes. Il est désactivé. Pendant ce laps de temps, il ne voit pas ce qu'il se passe dans le jeu. Ensuite il réapparaît à l'emplacement de la base de son équipe et peut recommencer à jouer.
Il n'y a pas de limite sur le nombre de tir possible avec l'arme laser.

Une fois pris, le drapeau ne peut pas être reposé, sauf dans le cas où le joueur est touché par le laser adverse. A ce moment-là, le joueur meurt dans les mêmes conditions évoquées ci-dessus, et le drapeau, quant à lui, retourne sur son socle d'origine.

La partie est gagnée lorsqu'une équipe a rapporté le drapeau adverse sur son socle. Un récapitulatif des scores de l'ensemble des joueurs est alors affiché (temps de la partie et nombre de fois mort pour chaque joueur).

Une équipe est composée d'un seul joueur pour ce premier projet. L'ajout d'autres joueurs dans une équipe est prévu dans les fonctionnalités optionnelles.

2.1.5. Les Commandes

Les commandes au clavier sont similaires à celles du célèbre jeu FreeDoom (http://freedoom.sourceforge.net/). Celles-ci sont :

- [CTRL] : tirer
- [UP] : avancer
- [DOWN] : reculer
- [LEFT] : tourner a gauche
- [RIGHT] : tourner a droite
- [SHIFT] : courir
- [ALT] + [LEFT] : se déplacer sur la gauche
- [ALT] + [RIGHT] : se déplacer sur la droite
- [R] : réinitialisation (suicide)
- [ESC] : quitter le jeu

La souris n'est pas utilisée.
Le saut n'est pas possible.

2.2. Charte graphique

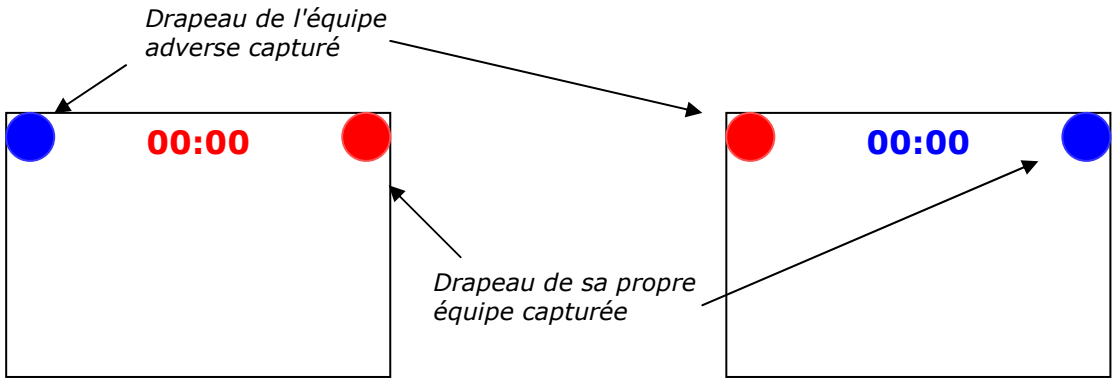
2.2.1. Interface

La vision de l'environnement du jeu est à la première personne, on ne voit donc pas son propre personnage.

De plus, le pistolet laser n'apparaît pas à l'écran. Il n'y a que la trace du tir du laser qui est visible.

Dans le déroulement du jeu, lorsqu'un joueur capture un drapeau, un voyant (de la couleur du drapeau) est affiché à l'écran. Ce voyant est aussi affiché sur l'écran de l'adversaire, l'avertissant ainsi que son drapeau a été capturé.

Enfin, un chronomètre est affiché en haut au centre de l'écran afin d'indiquer le temps passé pour la partie en cours. Schématiquement, l'interface des joueurs se présente ainsi :



Interface joueur équipe rouge *Interface joueur équipe bleue*

2.3. Fonctionnalités optionnelles

Selon l'avancement du projet, il est prévu d'ajouter si possible plusieurs fonctionnalités optionnelles au jeu. Nous avons regroupé ces fonctionnalités par lots.

2.3.1. Lot n°1

- Chargement de modèle 3D pour les personnages et les drapeaux
- Mise en place de texture sur les objets graphiques (mur, sol)
- Gestion du son

2.3.2. Lot n°2

- Affichage de l'arme et d'une partie du personnage (main)
- Barre de vie progressive

2.3.3. Lot n°3

- Augmenter le nombre de joueurs dans chaque équipe
- Prise en charge de la souris pour la visée du tir
- Changement de labyrinthe a chaque partie
- Partie limitée dans le temps

2.4. Principales contraintes

L'application doit fonctionner sur toutes les plateformes i386 (Linux, Windows..) pour lesquelles Java est disponible.

Pour des raisons évidentes de jouabilité, il est demandé aux systèmes d'avoir une puissance minimale ainsi qu'une quantité de mémoire minimale suffisante (> Pentium IV ou équivalent, mémoire >512Mo RAM).

Pour obtenir une bonne fluidité des mouvements, une carte vidéo gérant l'accélération 3D matérielle est nécessaire sur chaque poste.

L'application présente une structure multipostes connectés en réseau local de type Ethernet - TCP/IP, avec un joueur par poste. La connexion de poste distant à l'aide de liens bas débit (Internet) n'est pas prévue.

3. Management

3.1. L'équipe en charge du projet

L'équipe en charge du projet est composée de M. Agius Nicolas et de M. Décugis Fabien.

La charge de travail sera équivalente pour les deux membres de l'équipe. Ils travailleront toujours en parallèle afin d'optimiser la réalisation du projet. Ceci permettra d'avancer plus efficacement le projet car chacun connaîtra le travail de l'autre.

Ainsi, ils auront à leur charge l'étude et l'analyse du projet. Ensemble, en combinant et en confrontant leurs idées, ils réaliseront les différents diagrammes UML qu'un projet de cette ampleur demande.

A l'issu de cette tâche, une formation et un apprentissage pour chacun seront à prévoir (formation et apprentissage de java 3D entre autre).

La partie développement sera divisée en trois axes. Les deux premiers axes seront développés en parallèle par chacun des membres de l'équipe. Le premier axe concerne le développement au niveau du serveur avec la mise en place des scènes 3D. Cette partie sera développée par M. Agius Nicolas. Le second axe se porte au niveau du client. Cette partie sera développée par M. Décugis Fabien. Enfin, le dernier axe traite de l'intégration client-serveur. Elle sera développée en commun par M. Agius Nicolas et M. Décugis Fabien.

Les tests unitaires seront réalisés par les auteurs du code. Il en sera de même pour la correction du code.

Les tests fonctionnels et la validation ne seront pas, quant à eux, réalisés par les auteurs du code. C'est l'autre membre de l'équipe, celle qui n'a pas écrit le code qui devra faire ces tests et valider le code. La correction reste, quant à elle, à la charge de l'auteur du code.

Enfin, la rédaction et la présentation du projet seront faite en commun.

3.2. Organigramme des tâches

Cf. annexe n° 1.

3.3. Diagramme de GANTT

Cf. annexe n° 2.

LASER GAME 3D

Analyse et conception

Introduction

Afin de mener à bien le développement de cette application Java, nous avons utilisé la méthode d'analyse UML (Unified Method Language). Cet outil de modélisation objet, est l'outil d'analyse idéal pour développer en Java, langage objet par excellence. Une conception orientée objet a donc été choisie de façon naturelle.

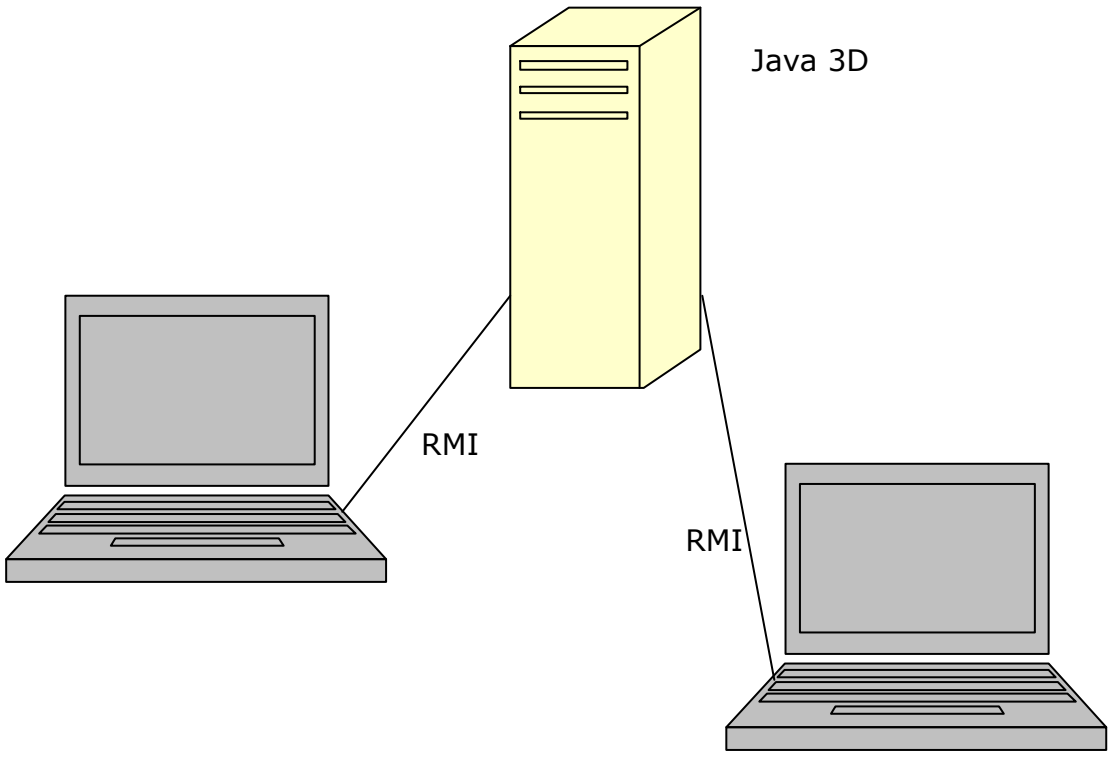
1. Analyses

1.1. Architecture générale

Nous avons dans notre système deux principaux acteurs : le client et le serveur. A partir de là, plusieurs solutions sont envisageables.

1.1.1. Première solution

La première solution consiste à mettre en place deux clients (un pour chaque joueur) et de les relier entre eux par un serveur.

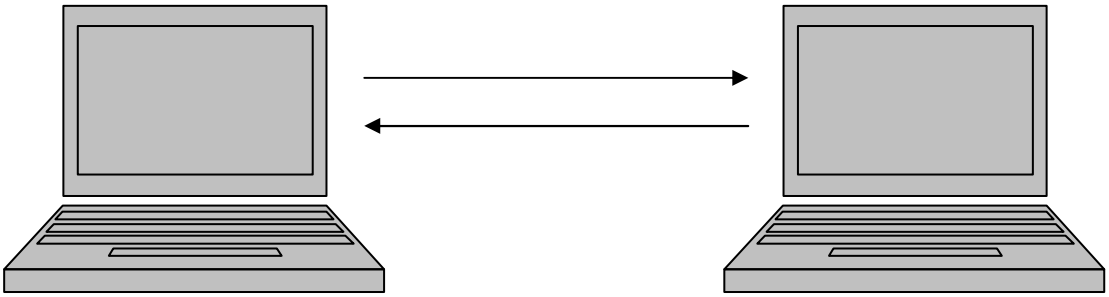


Le serveur gèrera les scènes 3D ainsi que tout l'environnement du jeu. Les clients gèreront uniquement les commandes et l'affichage.

Ce système devrait permettre d'optimiser la synchronisation puisque tout sera dès lors centralisé sur le serveur, les clients n'étant que de simple "terminaux".

1.1.2. Seconde solution

La seconde solution consiste à faire des clients, le serveur de l'autre client. Nous avons, ici, un système similaire au peer-to-peer.



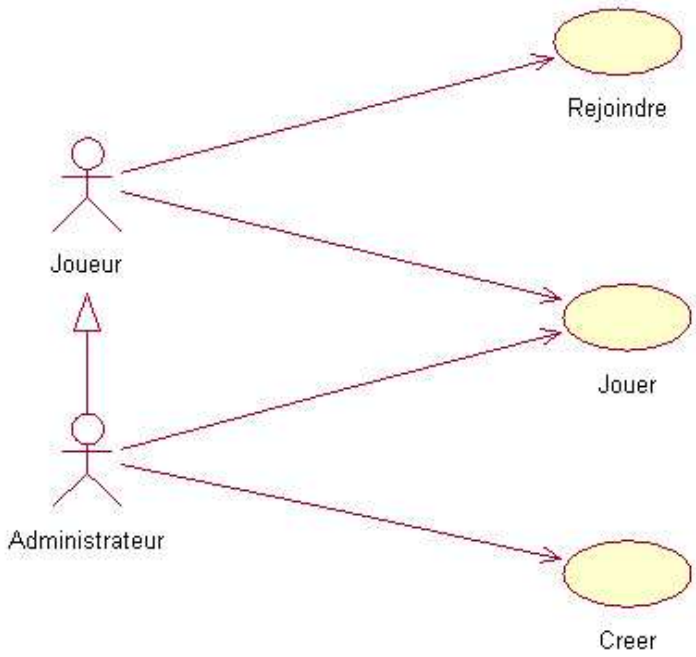
Avec cette méthode, c'est le client, à la fois serveur, qui va tout gérer et transmettre à son homologue les flux d'informations nécessaires pour l'exécution du système.

1.1.3. Solution retenue

L'avantage de la seconde solution réside principalement dans la rapidité de l'envoi des informations d'un client à l'autre puisque les informations ne transitent pas par un serveur. Cependant, dans le cas de cette seconde solution, nous pourrions être confrontés à un problème de synchronisation car nous avons deux environnements autonomes (un sur chaque client).

Aussi, face à ce problème, nous privilégierons la première solution qui, de plus, offre une approche réseau plus simple.

1.2. Diagrammes des cas d'utilisation UML



1.2.1. Description de "Créer"

Permet à l'administrateur de créer une nouvelle partie.

1.2.1.1. Flots principaux

- L'administrateur ouvre le dialogue de connexion
- Il saisit l'adresse IP du serveur et son identifiant (pseudo)
- Il valide les données pour se connecter au serveur
- Il saisit les paramètres de la partie (nombre maximal de joueurs)
- Il valide les données pour créer la partie
- Attente des joueurs : le système affiche les informations sur les joueurs rejoignant la partie
- Il lance la partie

1.2.1.2. Flots alternatifs

- L'administrateur demande sa déconnexion
- Le système stoppe la partie, tous les joueurs sont déconnectés
- Le processus client est fermé

1.2.1.3. Flots exceptionnels

- Le serveur ne répond pas : le système affiche le message d'erreur correspondant

1.2.1.4. Pré-conditions

- Le processus serveur et un processus client doivent être démarrés

1.2.1.5. Post-conditions

- L'administrateur est ajouté à la liste des joueurs de la partie créée
- La partie créée est en attente de joueurs supplémentaires

1.2.1.6. Acteurs

- Administrateur

1.2.2. Description de "Jouer"

Permet aux joueurs de jouer la partie.

1.2.2.1. Flots principal

- Le joueur se déplace

1.2.2.2. Flots alternatifs

- Le joueur tire, s'il touche un joueur adverse, celui-ci est désactivé pendant un certain temps
- Le joueur capture le drapeau, tous les joueurs sont notifiés
- Le joueur rapporte le drapeau : son équipe gagne ; la partie est terminée, un récapitulatif des scores est affiché
- Le joueur demande sa déconnection
- Le système le retire de la liste des joueurs de la partie et avertit les autres joueurs de son départ
- Le processus client est fermé.
- S'il y a moins de deux joueurs, la partie est terminée

1.2.2.3. Flots exceptionnels

- Le serveur ne répond pas : le système affiche le message d'erreur correspondant

1.2.2.4. Pré-conditions

- Le usecase "Créer" doit être terminé (partie lancée)

1.2.2.5. Post-conditions

- La partie est terminée

1.2.2.6. Acteurs

- Joueur
- Administrateur

1.2.3. Description de "Rejoindre"

Permet à un joueur de rejoindre une partie créée

1.2.3.1. Flots principal

- Le joueur ouvre le dialogue de connexion
- Il saisit l'adresse IP du serveur et son identifiant (pseudo)
- Il valide les données pour se connecter au serveur
- Il attend que la partie démarre (cf. usecase "Créer")

1.2.3.2. Flots alternatifs

- Le joueur demande sa déconnection
- Le système le retire de la liste des joueurs de la partie et avertit les autres joueurs de son départ
- Le processus client est fermé.

1.2.3.3. Flots exceptionnels

- Le serveur ne répond pas : le système affiche le message d'erreur correspondant

1.2.3.4. Pré-conditions

- L'étape de validation des données pour créer la partie du usecase "Créer" doit être réalisée (partie en attente de joueurs)
- Le processus client doit être démarré

1.2.3.5. Post-conditions

- Le joueur est ajouté à la liste des joueurs de la partie, l'équipe du joueur est choisie automatiquement par le serveur

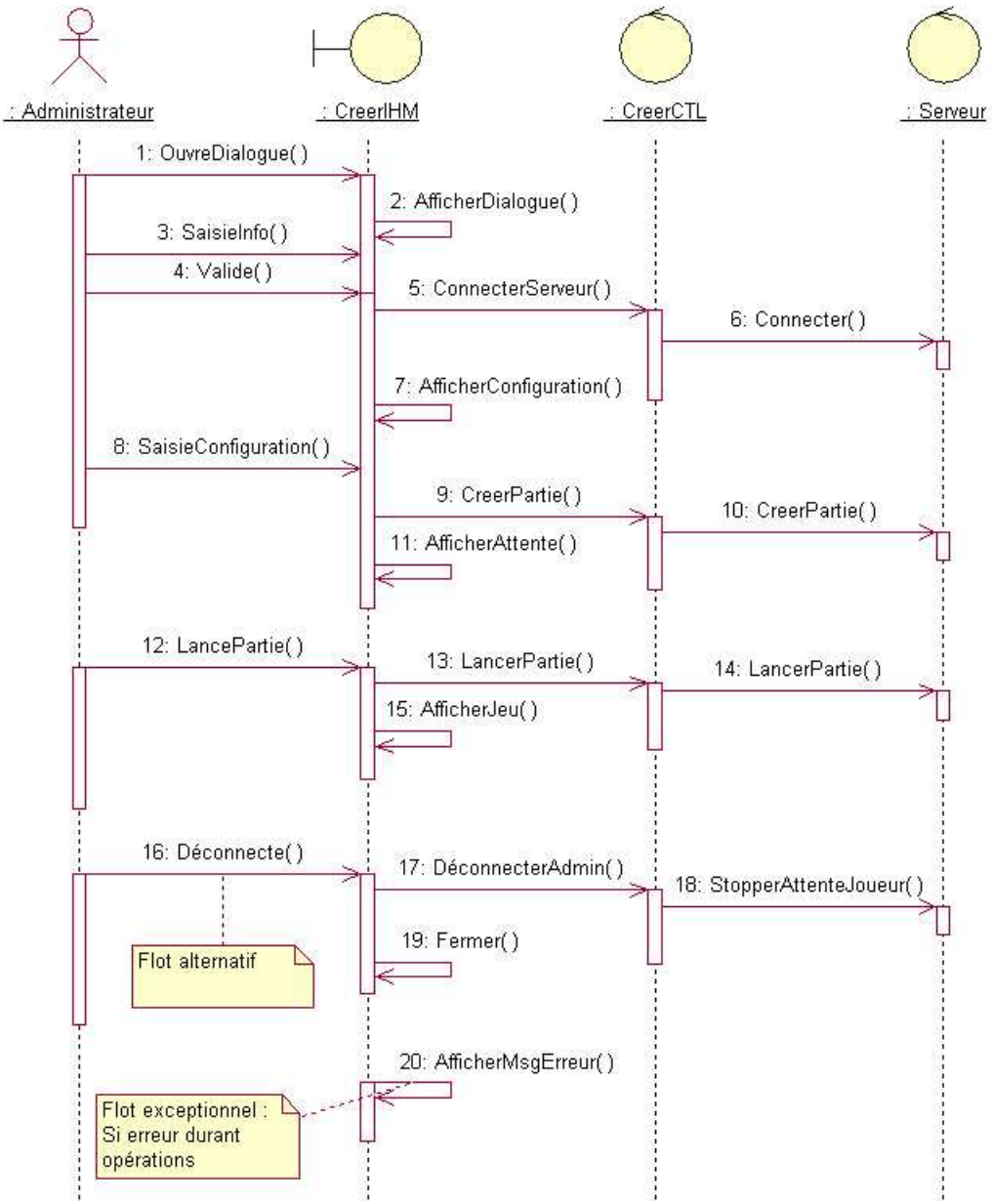
1.2.3.6. Acteurs

- Joueur

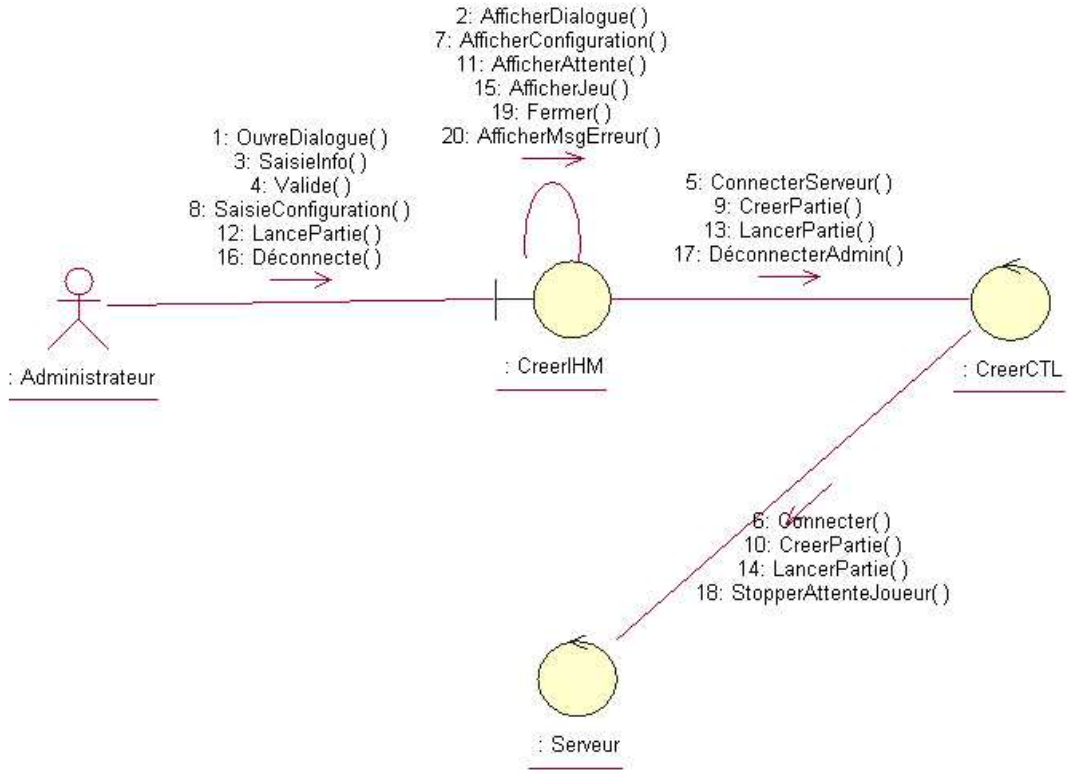
1.3. Les packages

1.3.1. Package "Créer"

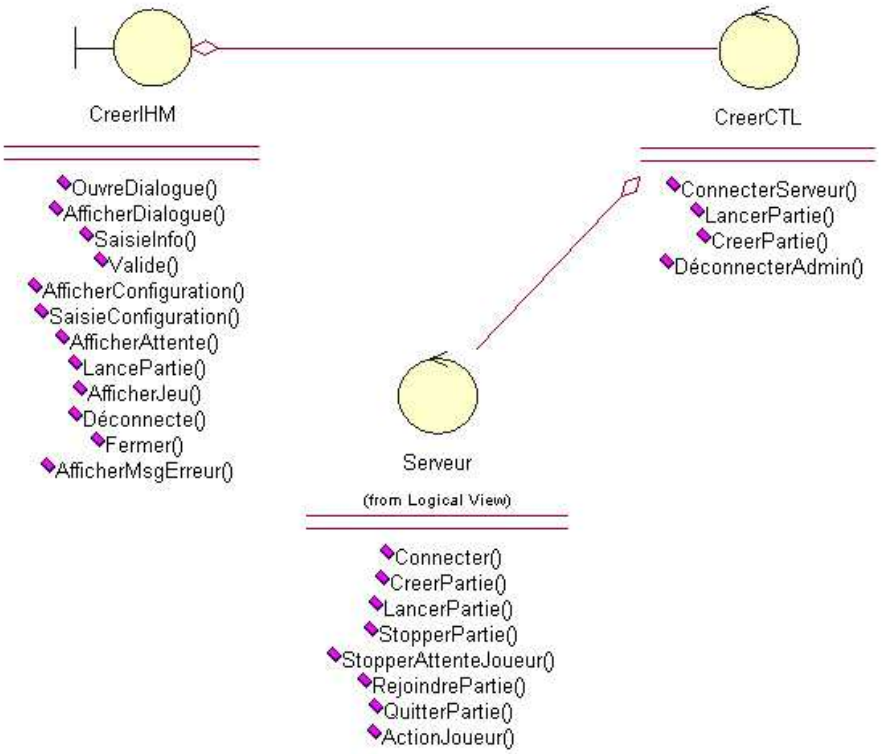
1.3.1.1. Diagramme de séquence



1.3.1.2. Diagramme de collaboration

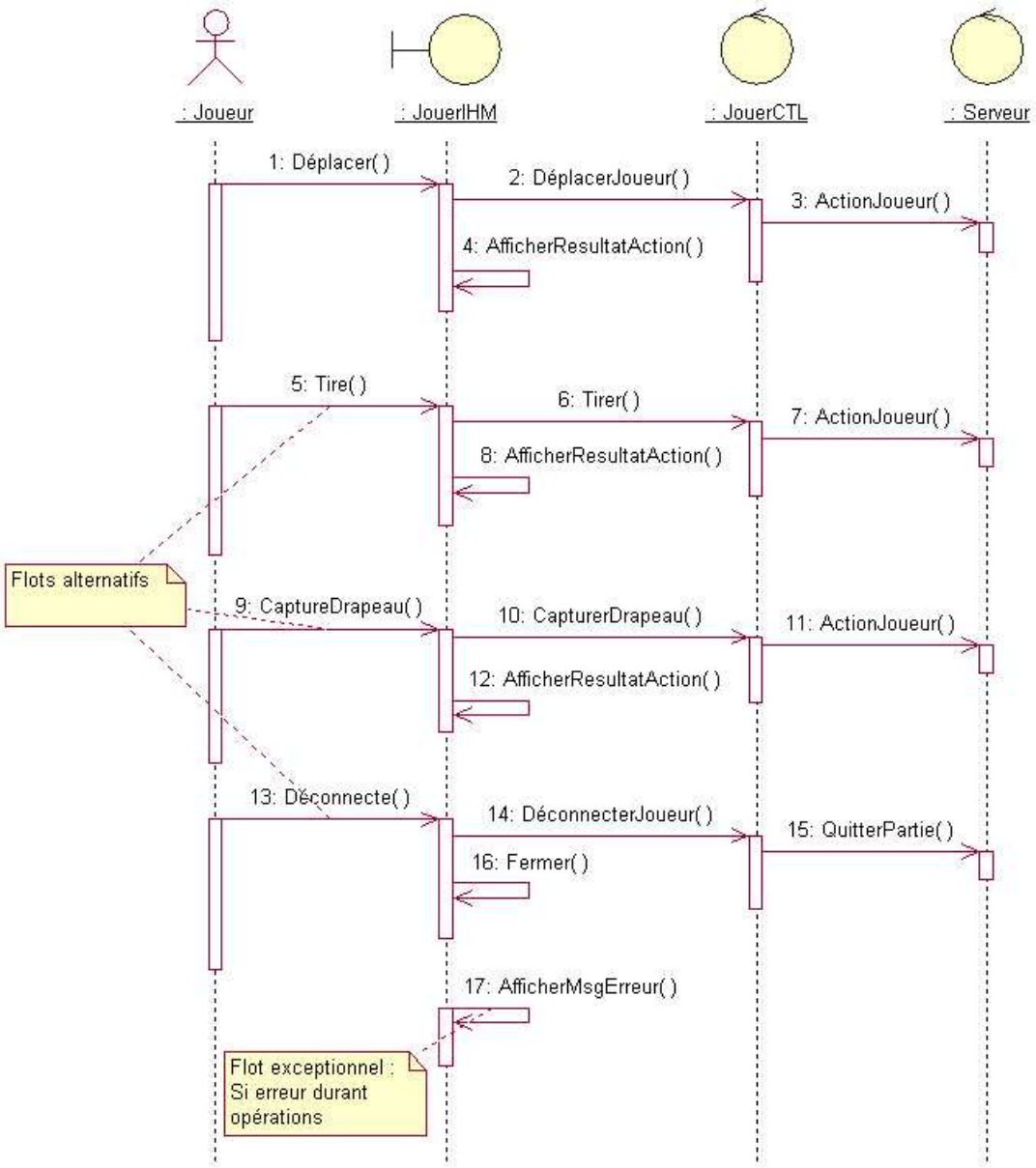


1.3.1.3. Diagramme de classe

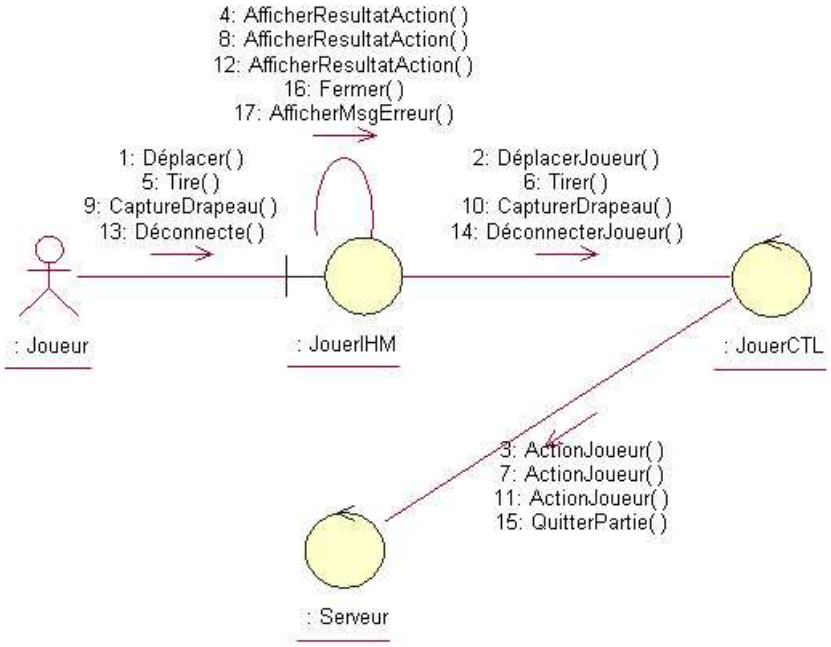


1.3.2. Package "Jouer"

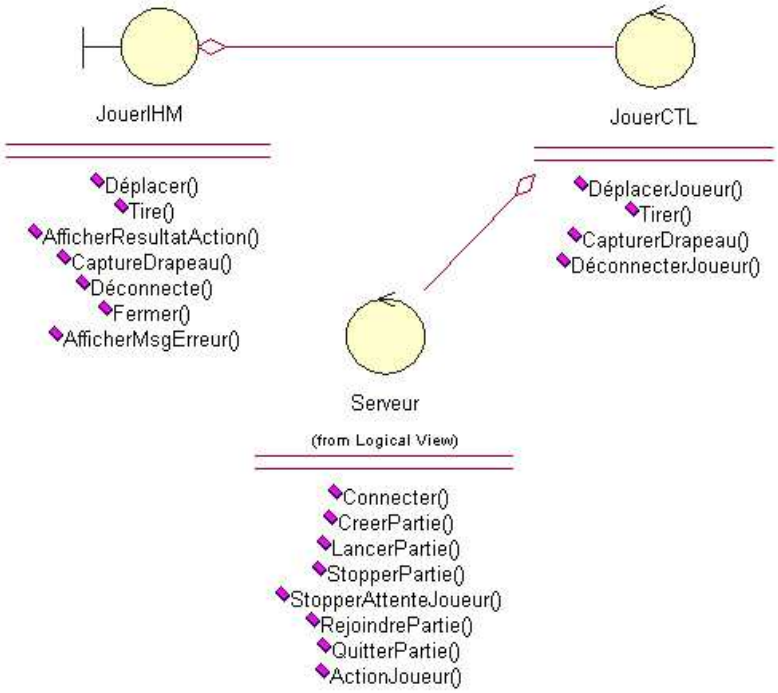
1.3.2.1. Diagramme de séquence



1.3.2.2. Diagramme de collaboration

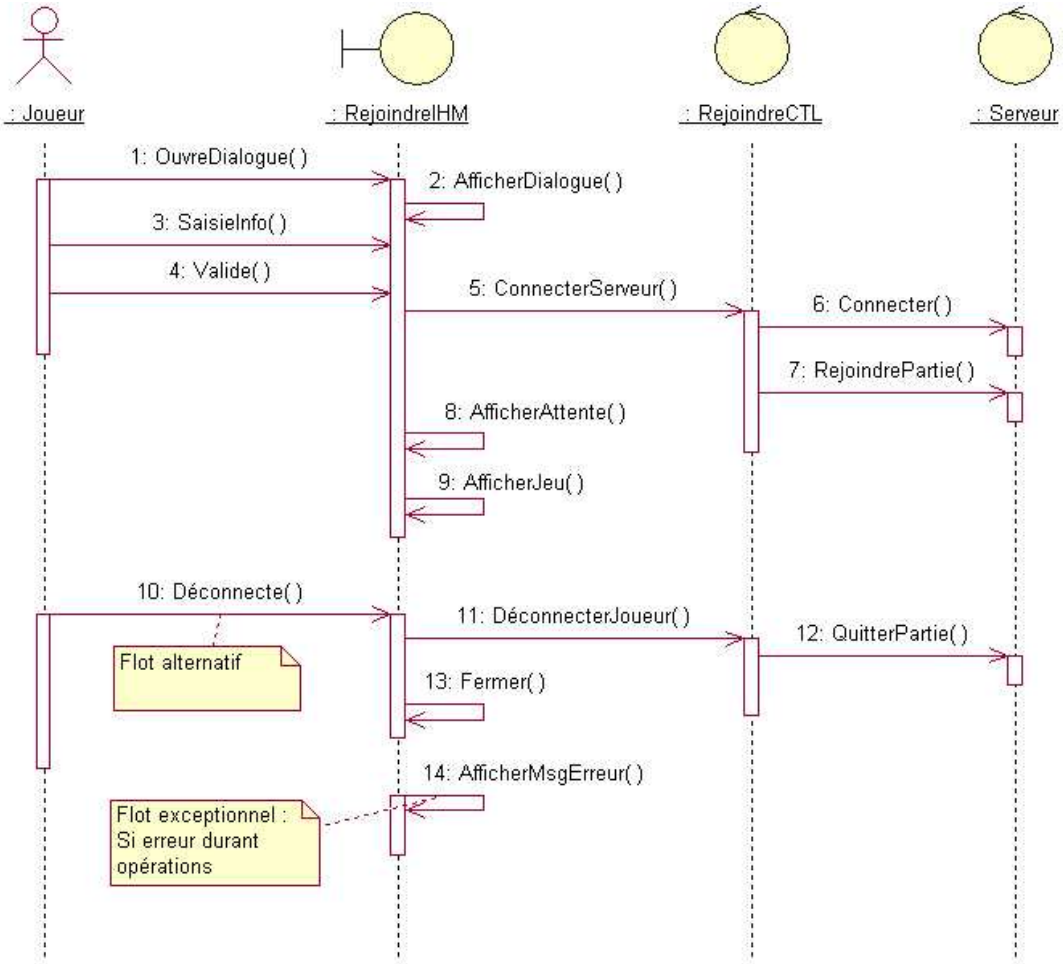


1.3.2.3. Diagramme de classe

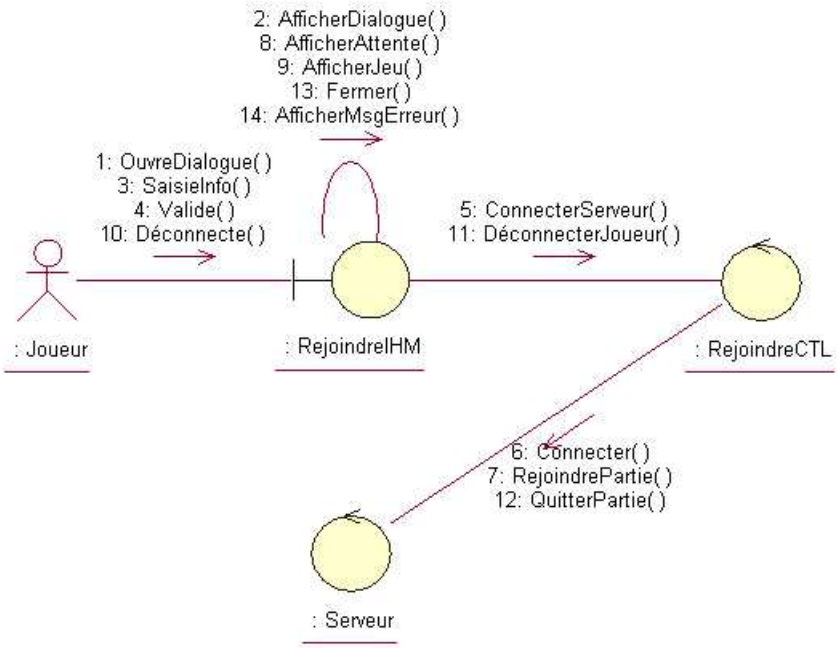


1.3.3. Package "Rejoindre"

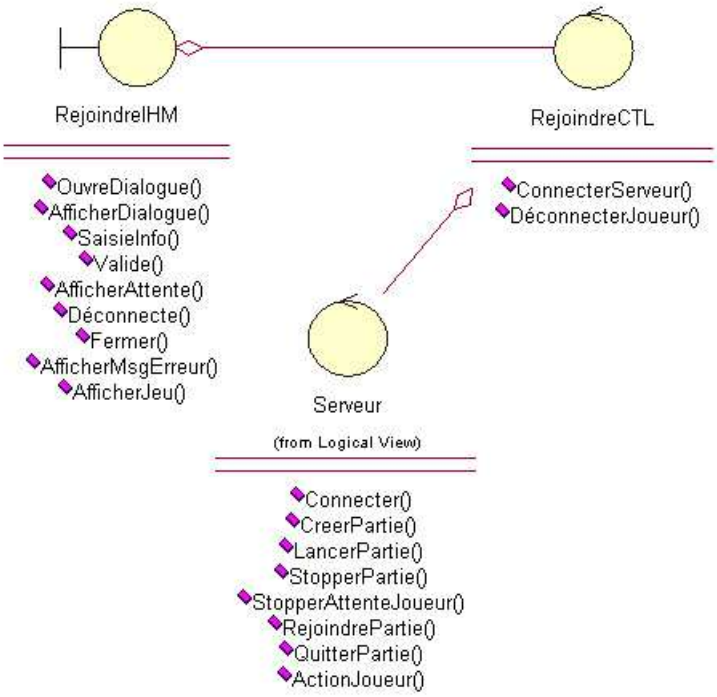
1.3.3.1. Diagramme de séquence



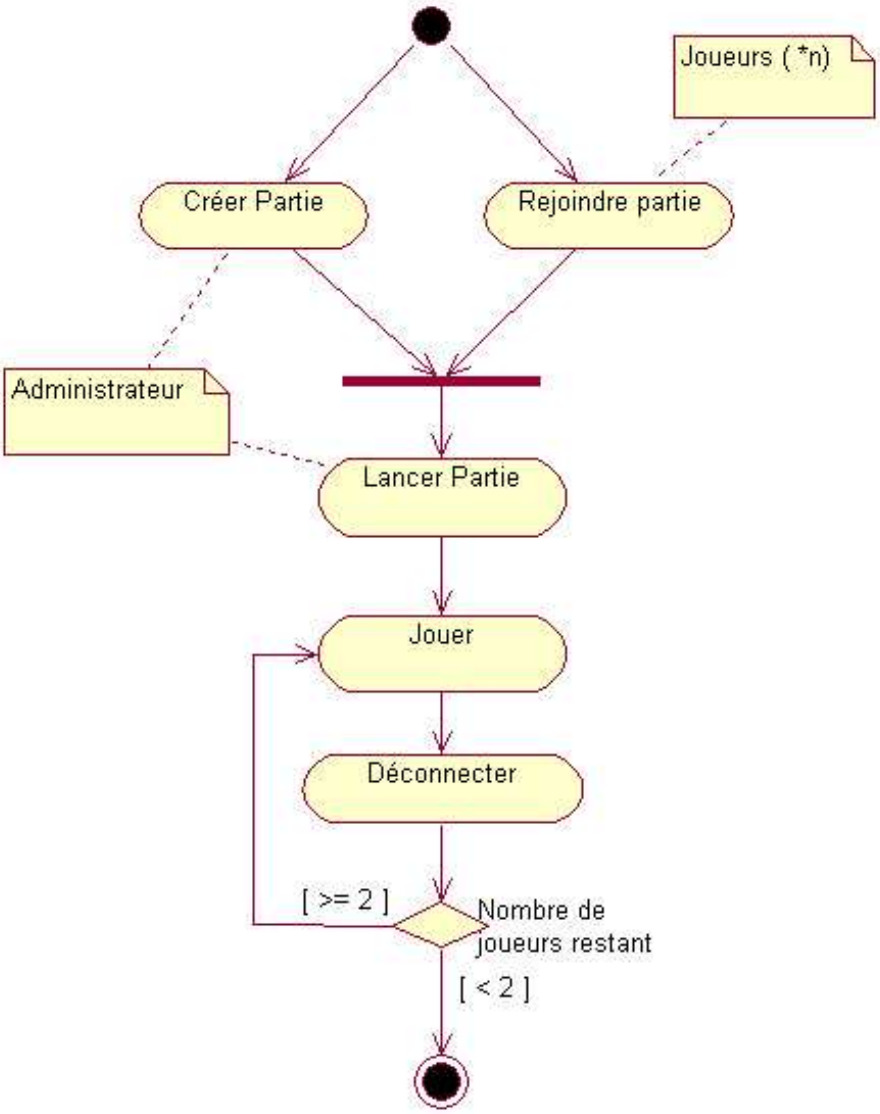
1.3.3.2. Diagramme de collaboration



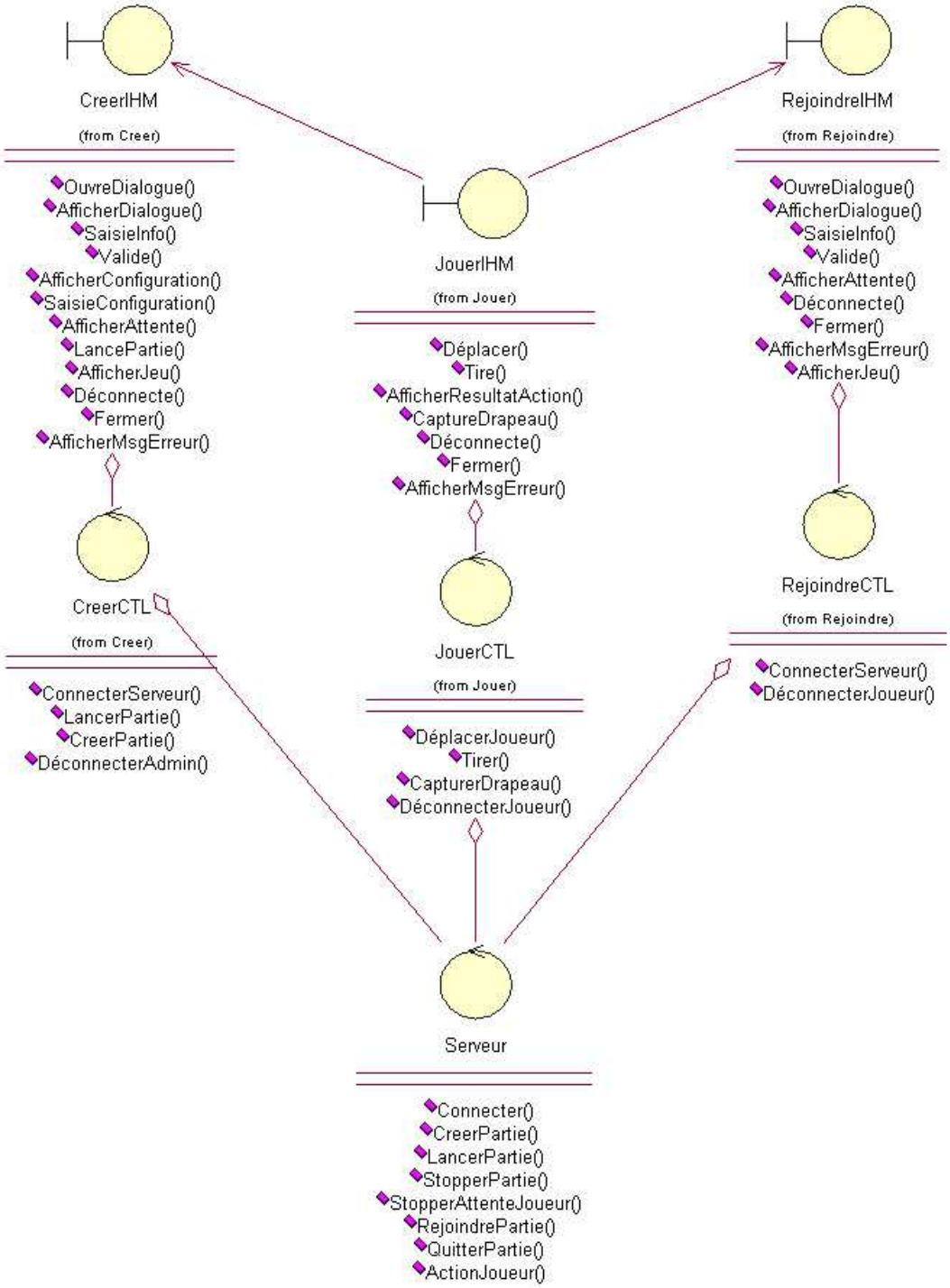
1.3.3.3. Diagramme de classe



1.4. Diagramme d'activité

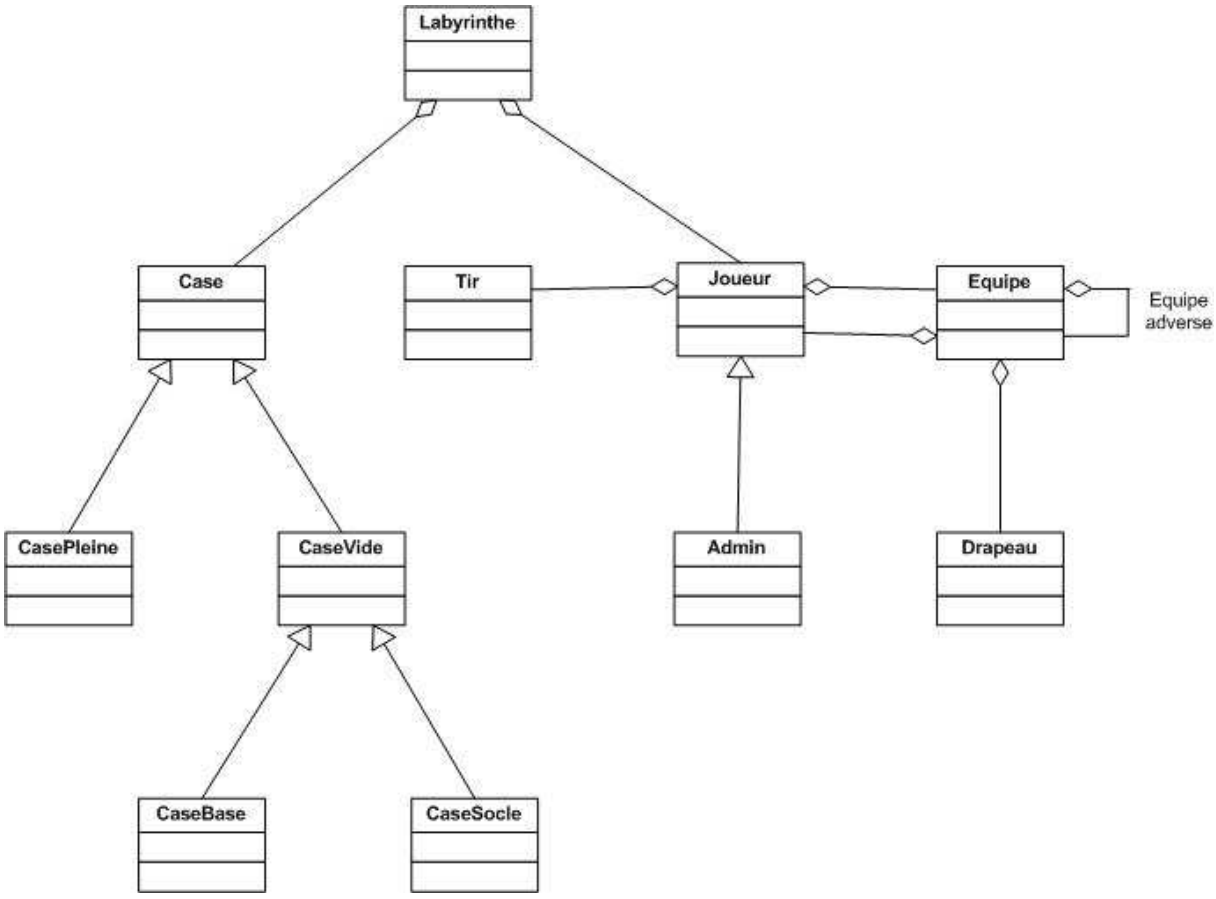


1.5. Diagramme de classe général UML



1.6. Diagramme de classe des éléments scéniques

Ce diagramme, issu d'un autre point de vue de l'étude, complémentaire de analyse UML, présente les différents éléments constituant la scène du jeu. Ceux-ci, ainsi que leurs relations, sont organisés dans un diagramme de classe, afin d'être intégré dans la conception orientée objet de l'application.



2. Conception et développement

2.1. L'architecture Java3D

Comme indiqué précédemment, cette application utilise les fonctionnalités de l'API Java3D fournie par Sun. Cette bibliothèque ayant été étudiée en cours, seuls quelques points précis allant au-delà du cours seront détaillés.

2.1.1. Gestion des collisions

Java3D fournit un ensemble de fonctionnalités pour détecter les collisions. Cependant, ces fonctions sont rudimentaires et ne permettent pas de gérer les collisions avec précisions. De plus, leur mise en oeuvre est rendue difficile par une Javadoc associée plutôt laconique, ainsi qu'un traitement quasi inexistant par le Tutorial de Sun sur Java3D.

Ces fonctionnalités utilisent deux façons différentes pour détecter les collisions : soit en utilisant les `Geometry` associé aux objets 3D, soit en utilisant, les `CollisionBounds`, des limites simplifiées de l'objet sous forme de cube ou de sphère invisible englobant celui-ci.

La détection des collisions utilisant les géométries, bien que nécessitant plus de calcul, est beaucoup plus précise. Cependant, elle est complètement buggée et plante lamentablement lorsqu'il y a plus de deux triangles en collision. Cette méthode n'est donc pas exploitable :

```
"C:\Program Files\Java\jdk1.6.0\bin\javaw" -classpath "C:\Documents and Settings\robert\jbproject\test\classes;
Exception in thread "J3D-GeometryStructureUpdateThread-1" java.lang.ArrayIndexOutOfBoundsException: 105
        at javax.media.j3d.GeometryArrayRetained.getVertexData(GeometryArrayRetained.java:10599)
        at javax.media.j3d.TriangleFanArrayRetained.intersect(TriangleFanArrayRetained.java:461)
        at javax.media.j3d.GeometryRetained.intersect(GeometryRetained.java:288)
        at javax.media.j3d.Shape3DRetained.intersectGeometryList(Shape3DRetained.java:2106)
        at javax.media.j3d.BHTree.doSelectAny(BHTree.java:440)
        at javax.media.j3d.BHTree.doSelectAny(BHTree.java:463)
        at javax.media.j3d.BHTree.doSelectAny(BHTree.java:463)
        at javax.media.j3d.BHTree.doSelectAny(BHTree.java:470)
        at javax.media.j3d.BHTree.doSelectAny(BHTree.java:463)
        at javax.media.j3d.BHTree.doSelectAny(BHTree.java:463)
```

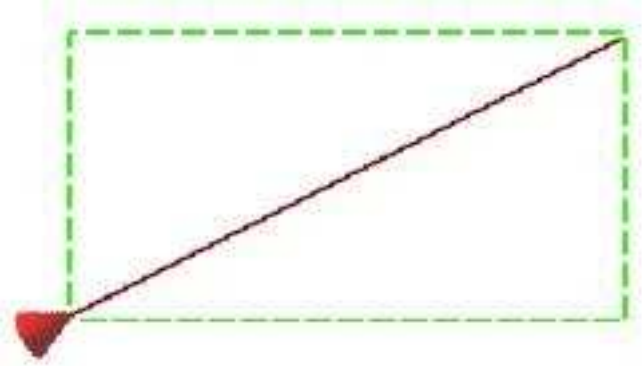
L'autre méthode utilise les Bounds. Ces bornes virtuelles sont par défaut souvent plus large que l'objet lui-même, déclenchant les évènements de collision alors que les objets sont seulement à proximité.

La méthode `setCollisionBounds()` permet de définir ces limites qui vont servir à la détection de la collision. Mais celle-ci n'affecte l'objet que si une fonction `WakeupOnCollisionXXX()` a été appelée dessus et seulement du point de vue du `Behavior` qui a appelé le `WakeupOnCollisionXXX()`.

Pour les autres objets qui entre en collision avec celui définit par `WakeupOnCollisionXXX()`, le `CollisionBounds` par défaut est utilisé, ignorant la directive `setCollisionBounds()`. Ce bounds par défaut est souvent mal adapté car il est généralement trop grand et reste orthogonal par rapport au repère du `VirtualUniverse`.

Pour illustrer ce problème, prenons le cas du tir, une forme cylindrique très longue. Si le tir est aligné sur un des axes du repère, les limites suivent à peu près le cylindre et sont donc acceptables :

Si, maintenant le tir est orienté dans une direction quelconque, les limites restent représentées par un parallélépipède aligné sur le repère et sont donc beaucoup trop larges :



Par conséquent, lors de la collision de deux objets qui ont chacun un `Behavior` en attente de collision, la détection de collisions ne sera par déclenché de façon identique suivant les points de vue des `Behavior`, en fonction des `CollisionBounds` qui seront différents.

A noter que ce comportement n'est pas documenté par Sun.

De plus, ces fonctionnalités ne permettent pas de détecter les collisions entre plus de deux objets, ce qui implique un comportement erroné en cas de collision avec un mur et un drapeau par exemple, ou si l'avatar du joueur est touché par un tir en même temps qu'il se trouve sur la base.

Références : Tutorial Java3D de Sun, Chapitre 4-18.

Pour allez au-delà des restrictions de Java3D en matière de collision, il est donc nécessaire d'abandonner ces outils et d'utiliser des moteurs physique tel

que ODE, qui permettent un rendu plus réaliste. Cependant, ces bibliothèques sont généralement écrites en C et nécessitent des bindings spéciaux pour Java.

Des solutions alternatives utilisant des mécanismes plus "bas niveau" de Java, ou d'autres techniques, comme le *picking*, sont en cours de développement par des groupes de développeurs tiers, comme le projet j3d.org, mais ne sont pas à l'heure actuelle dans un état assez avancé pour être exploitables.

2.1.2. Évitement de collisions

L'API Java3D ne possède pas de fonction permettant de gérer les évitements. Cependant, il est possible de créer ces comportements en se basant sur les fonctionnalités Java3D de détection de collisions.

2.1.2.1. Principe

Le principe que nous avons utilisé est le même qu'en mécanique :
Lorsqu'un objet physique est, par exemple, posé sur une table, la table réagit par une force normale à son plan qui repousse l'objet et l'empêche de passer au travers de la table.

Transposé dans l'univers de Java3D, cette loi de mécanique élémentaire a été mise en oeuvre ainsi :

Lorsque l'avatar entre en collision avec un mur, on récupère les positions absolues des deux éléments afin de pouvoir ensuite calculer leurs positions relatives. En effet, l'avatar et les murs n'étant pas issus de la même branche, il n'est pas possible de retrouver directement leurs positions relatives via leurs `transformGroup` respectifs : il faut nécessairement le faire via les méthodes `getLocalToVWolrd()`.

Une fois la position relative obtenue, on l'utilise pour calculer la normale au plan de contact du mur. Ce vecteur est ensuite utilisé comme vecteur de translation pour corriger la position de l'avatar et ainsi l'empêcher d'entrer dans le mur.

Dans la hiérarchie de l'arbre3D, l'avatar est positionné par deux `transformGroup`. Le second est piloté par le `Behavior` qui a en charge la gestion des déplacements demandés par le clavier. Le premier est utilisé par la gestion de collision présentée ci-dessus pour corriger la position en fonction des murs. Ce dernier est positionné en haut de l'arbre pour rester en coordonnée absolue.
Deux `TransfromGroup` en cascade sont donc nécessaire. En effet, si on branche les deux `Behavior` sur le même `transformGroup`, ceux-ci entrent en conflit et le résultat final devient imprévisible.

2.1.2.2. Inconvénient de cette technique

Cette implémentation fonctionne bien mais présente quelques désagréments :

La réaction à la collision se faisant à posteriori, il en résulte un effet vibratoire désagréable. En effet, l'évitement de collision n'est pas effectué en temps réel puisqu'il faut qu'il y ai d'abord collision effective puis attente de la remontée de l'évènement associé, avant de pouvoir calculer la réaction et l'appliquer.

De plus, le `Behavior` en charge des déplacements étant plus rapide que celui gérant les collisions (plus de calculs), les murs deviennent "mous" si on entre en collision avec une vitesse trop élevée. Ceci est d'autant plus sensible avec la gestion d'accélération et d'inertie du `KeyNavigatorBehavior`, en fonction de la fréquence de répétition des touches.

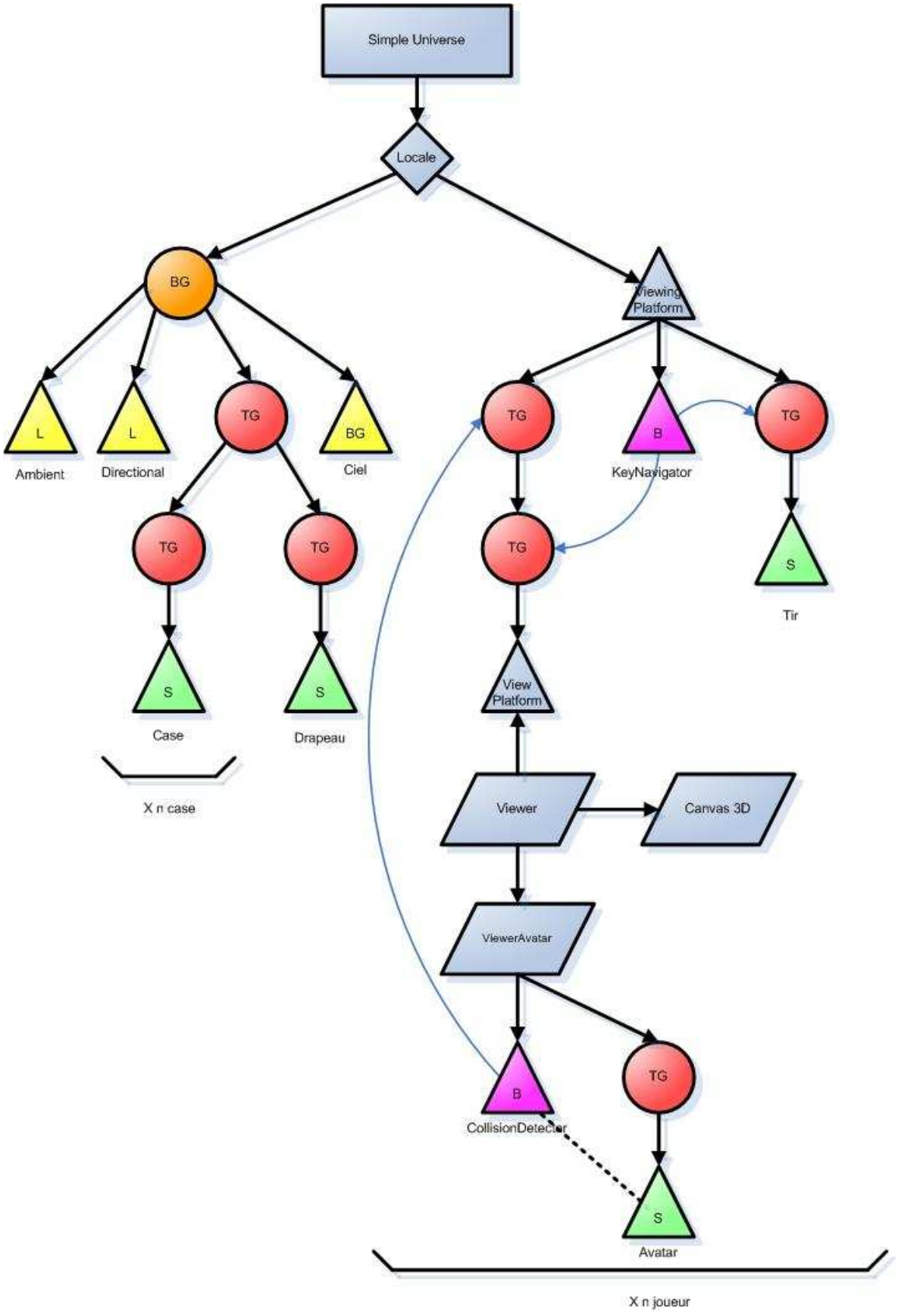
Concernant la réalisation du labyrinthe, bien que tous les objets "Mur" soient identiques, l'utilisation des `SharedGroup` pour alléger l'empreinte mémoire n'est pas possible. En effet, lors d'une détection de collision avec un objet appartenant à un `SharedGroup`, le `Behavior` déclenche une `IllegalArgumentException` non documentée qui rend l'ensemble inutilisable.

2.1.3. Arbre de la scène 3D

L'essentiel de la structure se situe dans la `ViewBranch`. On y retrouve les deux `transformGroup` en cascade, parent du `Viewer` associé au joueur, respectivement piloté par les `Behavior` :

- `CollisionDetector`, à l'écoute des évènements sur le Shape3D de l'Avatar.
- `KeyNavigatorBehavior` qui réceptionne les évènements claviers et qui pilote le fonctionnement du tir.

La `BranchGraph`, quant à elle, contient les divers éléments du labyrinthe et l'éclairage de la scène.


Simple Universe
Locale
BG
Viewing Platform
L
Ambient
L
Directional
TG
BG
Ciel
TG
B
KeyNavigator
TG
TG
TG
S
Tir
TG
TG
S
S
Case
Drapeau
X n case
View Platform
Viewer
Canvas 3D
ViewerAvatar
B
CollisionDetector
TG
S
Avatar
X n joueur

2.2. Mise en réseau

Pour établir les communications réseaux entre les différents éléments de l'application, deux solutions s'offrent à nous :

- soit définir un protocole réseau spécifique à notre application.
Cette solution est lourde à mettre en place et peu flexible.

- soit utiliser RMI[1],une interface de programmation Java, qui propose un ensemble de méthodes permettant d'accéder facilement à des ressources, c'est à dire des objets, situé sur des machines distantes.

C'est cette solution que nous avons retenue, une solution 100% Java, dans l'esprit de la programmation objet.

2.2.1. Fonctionnement de RMI

2.2.1.1. Introduction

RMI est une API[2] intégrée dans l'environnement Java qui permet d'invoquer des méthodes sur des objets distants de manière transparente pour le développeur, c'est-à-dire de la même façon que si l'objet était sur la machine locale.

Cette interface reprend les concepts des RPC[3], adaptés à une approche objet. C'est un des concurrents de la norme CORBA, mais d'une utilisation plus simple et réservé au Java.

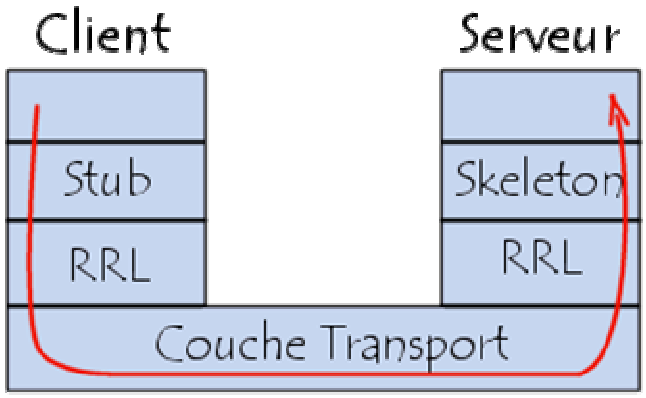
2.2.1.2. Structure

Les connexions et les transferts de données sur le réseau TCP/IP se font à travers un système de couches, inspiré du modèle OSI[4], afin de garantir une interopérabilité entre les programmes et les versions de Java.

De plus, la localisation du serveur n'est pas connue du client. Les objets distants sont chargés au travers d'une URL[5]. Un service de localisation intégré à RMI, appelé serveur de liaison et exécuté sur une machine connue à la fois du client et du serveur, permet d'identifier un objet donné à l'aide un mot clé associé à l'URL.

[1] RMI : Remote Method Invocation
[2] API : Application Programming Interface
[3] RPC : Remote Procedure Call
[4] Modèle OSI : modèle des communications réseaux en 7 couches.
[5] URL : Uniform Resource Locator

Voici le modèle des communications utilisé par RMI:



- Du coté client, la classe Stub (Souche) est le représentant local de l'objet distant qui implémente les méthodes accessibles de cet objet**.**
- Du coté serveur, la classe Skeleton (Squelette) est l'interface d'accès au véritable objet instancié sur le serveur
- La couche de référence (*RRL*, *remote Reference Layer*) traduit la référence locale de l'objet vers une référence à l'objet distant en effectuant la localisation. Elle est aussi appelé registre RMI.
- La couche de transport est chargée d'écouter les appels entrants ainsi que d'établir et gérer les connexions réseaux avec les systèmes distants.

Ainsi, une application client-serveur basé sur RMI met ainsi en oeuvre trois composantes :

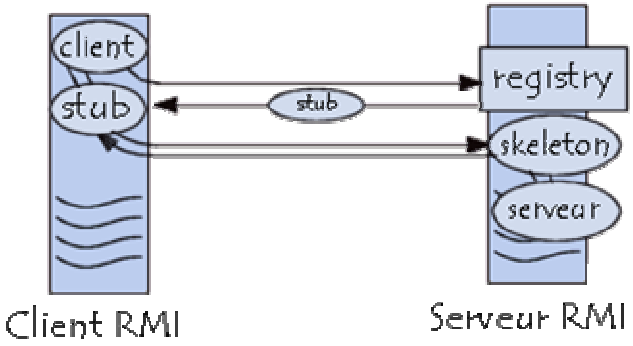
- une application cliente implémentant le Stub
- une application serveur implémentant le Skeleton
- un serveur de liaison réalisé par un processus tiers

De plus, les accès peuvent être contrôlés par le `SecurityManager`[6] de Java.

[6] SecurityManager : Système de gestion des accès Java

2.2.1.3. Fonctionnement

L'architecture de RMI est schématisée ci-dessous :



Dans notre cas, pour simplifier l'architecture, la partie serveur et le gestionnaire de liaisons sont exécutés sur la même machine.

Lorsqu'un objet sur une machine cliente désire accéder aux méthodes d'un objet distant, il effectue les opérations suivantes :

1. Il demande la localisation de l'objet distant au serveur de liaison.
2. En retour, il obtient une image virtuelle de l'objet distant, le Stub, ayant exactement la même interface que l'objet lui-même.
3. A l'appel d'une méthode sur le client, le Stub sérialise les arguments de la méthode, c'est à dire les transforme en une suite d'octets, puis les transmet au serveur via le réseau.
4. Sur le serveur, le Skeletton dé-sérialise les données envoyées par le Stub, puis invoque la méthode sur l'objet local.
5. Ensuite, il sérialise la valeur retournée par la méthode et expédie les données au client.
6. Sur le client, le Stub dé-sérialise la valeur retournée par la méthode distante et les transmet à l'objet appelant.

Avant d'effectuer ces opérations, il faut d'abord instancier, sur le serveur, l'objet que l'on souhaite partager, puis l'enregistrer auprès du registre RMI pour qu'il devienne accessible.

2.2.1.4. Utilisation

Pour utiliser RMI, il faut :

- Définir une interface qui sera implémenté à la fois par le Stub du client et par l'objet lui même, sur le serveur. Cette interface désigne toutes les fonctions publiques de l'objet qui vont être rendue accessible par RMI.

- Coder l'implémentation de cette interface, c'est à dire la classe de l'objet distribué sur le serveur.

- Générer les classes *Stub* et *Skeleton* correspondantes avec l'outil *rmic*[7]

- Ecrire le code de la partie serveur qui :
  - instancie l'objet implémentant l'interface
  - exporte le *Stub* de l'objet
  - attend des requêtes via le *Skeleton* de l'objet

- Ecrire le code de la partie client qui utilise l'objet distant en :
  - important un *Stub* d'objet distant
  - invoquant une méthode de l'objet distant via son *Stub*

Le serveur de liaison peut être mis en œuvre de deux façons différentes :

- soit en exécutant le code java :
  `LocateRegistry.createRegistry(1099);`
- soit en exécutant la commande système :
  `start rmiregistry 1099` (Pour Windows)
  `rmiregistry 1099 &` (Pour Linux)

A des fin de simplification pour l'utilisateur final, nous avons choisit la première méthode, la plus transparente du fait que le serveur de liaison est exécuté sur la même machine que l'applicatif serveur.

A noter qu'une restriction de sécurité interdit l'export d'objet pour les serveurs RMI qui ne sont par sur la même machine que le serveur de liaison. Pour contourner cette limitation, il faut donc créer un `RegistryProxy` qui va s'occuper d'enregistrer dans le registre RMI les objets exportés par les serveurs RMI distants.

2.2.2. Architecture réseau

L'architecture retenue pour la mise en réseau multi-joueur est de type client léger - gros serveur. L'idée était de créer la scène 3D et d'exécuter tous les traitements sur le serveur et de n'envoyer aux clients que l'affichage, c'est à dire les objets `Canvas3D`, via RMI.

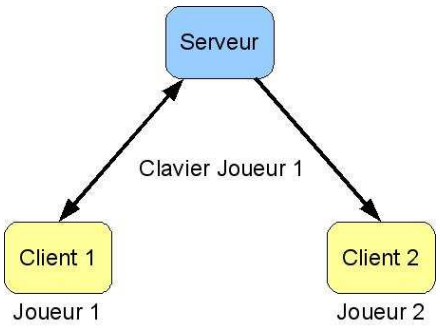
Cette solution était séduisante puisqu'elle ne nécessitait pas de structure de synchronisation, la scène et l'environnement de jeu étant unique. Cependant, aucun des objets de la scène3D, dont le `Canvas3D`, ne sont sérialisables. Cette approche n'est donc pas réalisable avec RMI.

[7] rmic : Outils spécifique fournit par Java

La structure client serveur a donc été repensée pour tenir compte de ces restrictions : Tous les traitements ainsi que la scène 3D sont placés sur le client. Il y a donc autant d'*univers* identiques que de clients (donc de joueurs). Seul les commandes claviers passent par le réseau. Le serveur n'a donc plus que le rôle de répartiteur d'évènements claviers.

2.2.3. Principe de communication

Sur chaque client, le clavier du joueur n'est pas raccordé directement au personnage[8] « local » mais est seulement envoyé au serveur. Il n'y a donc pas de personnage fonctionnant en local : ils sont tous « distants » et récupèrent les évènements claviers depuis le serveur. Ce passage par le réseau y compris pour le personnage du joueur local peut paraître surprenant mais ceci permet d'introduire les mêmes latences réseaux pour tous les personnages, qu'ils soient locaux ou distants et ainsi rester synchrone.

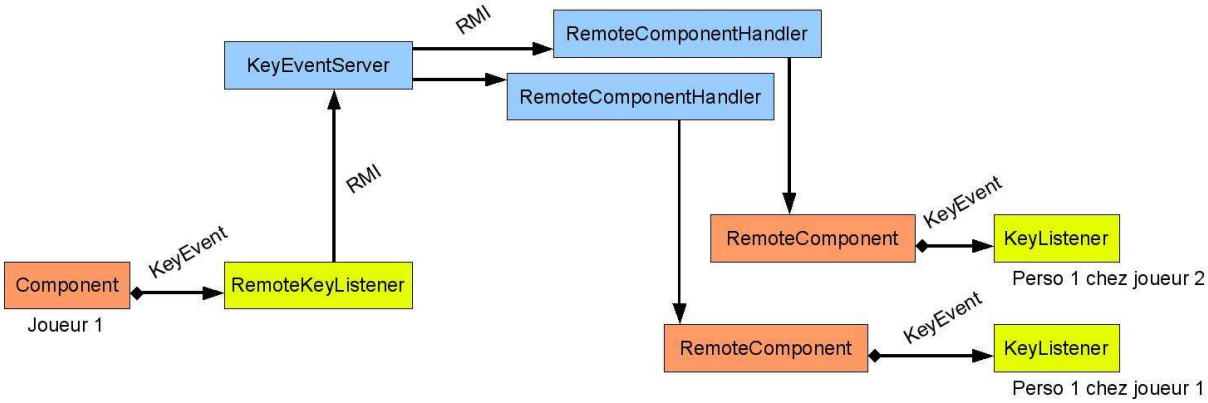


Le serveur a donc le rôle de distributeur d'événements claviers. Ceci est réalisé en transmettant les `KeyEvent` au serveur via RMI qui les ré-envoi à son tour à tous les clients.

Dans l'architecture AWT de Java, les `KeyEvent` proviennent d'un `Component` associé à l'interface qui les transmet au `KeyListener` en charge du traitement souhaité.

L'interception des `KeyEvent` au niveau de l'interface du client se fait par un "KeyEvent-RMI Wrapper" qui hérite de la classe `KeyListener` et qui les envoie au serveur. Leur injection dans la scène 3D au niveau du `KeyNavigatorBehavior` se fait par la même technique, mais cette fois ci en héritant de la classe `Component`.

[8] Personnage : représentation du joueur dans le jeu



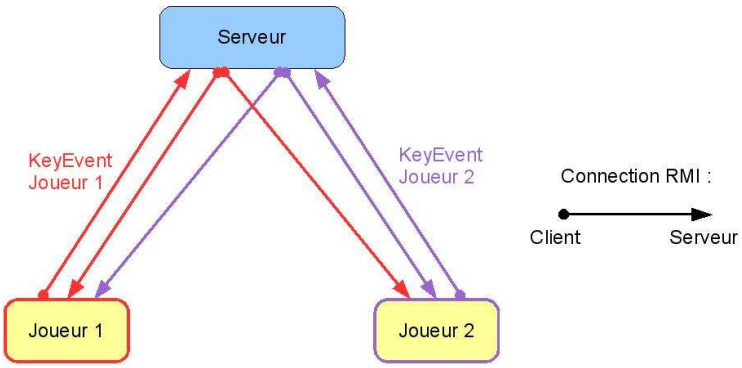
Il n'y a donc pas de message à l'initiative du serveur, les communications sont asynchrones pour les clients receveurs et initiée par le client émetteur.

Remarque sur le fonctionnement de RMI

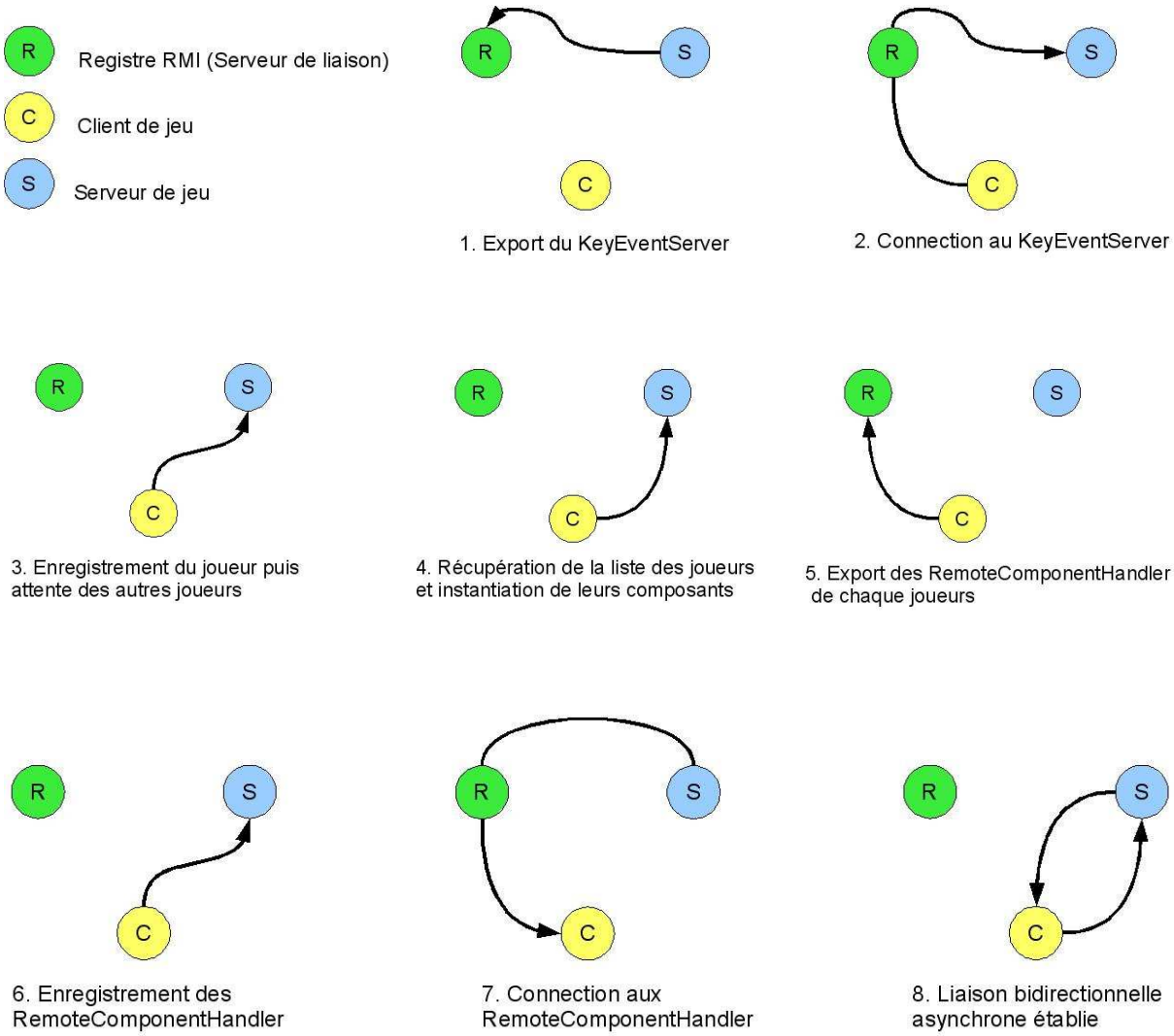
Lors du passage d'un objet quelconque, instancié sur le client, en paramètre à une méthode exportée par RMI, celui-ci devient disponible sur le serveur, mais perd ses références vers les autres objets du client. En clair, si on fait un `setXXX()` sur l'objet instancié sur le client, une fois transmis au serveur, le `getXXX()` correspondant retourne *`null`*.

Par conséquent, les communications bidirectionnelles asynchrones ne sont pas possibles avec une seule connexion RMI.

Il est donc nécessaire de créer plusieurs connections client serveur tête-bêche pour obtenir le flux bidirectionnel asynchrone requis par nos communications :



En plus de la gestion et de la distribution des `KeyEvent`, le serveur s'occupe de l'enregistrement des joueurs et de leurs composants avant le début de la partie, dont le schéma suivant présente le déroulement :



2.2.4. Avantage et inconvénient de cette solution

Avec cette technique, il n'y a pas besoins de transmette les informations concernant les tirs, les collisions ou les autres évènements de jeu. Ceux-ci étant calculés en local sur chaque client, à partir des évènements claviers, ils restent parfaitement synchrones. De plus, les latences réseaux étant identiques pour tous les personnages d'un même client, la charge réseaux n'influence pas le jeu.

Cependant, à cause de la gestion des accélérations qui diffère suivant la puissance de la machine ainsi que la gestion des collisions qui n'est pas franche, il apparaît très rapidement un décalage dans la position d'un même personnage sur les clients différents.

Pour résoudre ce problème, la solution a consisté à utiliser les mêmes canaux de communication RMI pour envoyer régulièrement la position du personnage local vers les personnages distants, afin de garantir l'égalité des positions.

Le déplacement des personnages étant calculé en temps réel en fonction des évènements clavier, une fréquence d'1Hz, pour la synchronisation des positions, est suffisante pour obtenir des déplacements précis et identiques tout en n'induisant qu'une charge réseau minime.

2.2.5. Mesure de l'utilisation de la bande passante

Ces mesures réseaux avaient été prévues, à l'origine, pour mesurer les débits et valider l'exploitation du jeu sur un réseau local standard (100 Mbit/s), puisque la première solution avec le serveur centralisant tous les traitements laissait supposer de gros transferts réseaux vers les clients.

La structure ayant changée, ces mesures sont moins capitales, mais montre bien l'efficacité de la nouvelle solution. En effet, seul les évènements claviers transitant par le réseau, le volume de données communiquées est faible. De plus, il varie en fonction de l'activité du joueur.

Le débit maximal, c'est à dire avec les touches appuyées de façon continue, est d'environ 15ko/s. Le débit moyen au cours d'une partie varie aux alentours de 5-8ko/s.

Quant à elle, la synchronisation périodique des positions respectives à une fréquence d'1 Hz, n'occupe qu'un débit de 0,8ko/s.

Ces chiffres représentent les débits montant mesurés depuis le client vers le serveur. Ils sont à multiplier par le nombre de joueurs pour obtenir le débit descendant sur chaque lien.

2.3. Diagramme de classe final

Le diagramme de classe final de l'application a été conçu a partir du dernier diagramme de l'analyse UML et du diagramme de classe des éléments scéniques. Il a été étendu pour intégrer les contraintes de l'architecture de l'environnement Java, et plus particulièrement des spécificités de Java3D.

Nous retrouvons sur ce diagramme, en plus de la structure issue de l'analyse, la partie qui est dérivée de l'arbre de la scène 3D. Bien que les deux n'aient pas de rapport direct, la structure objet de Java3D permet d'intégrer facilement l'arbre dans une conception objet.

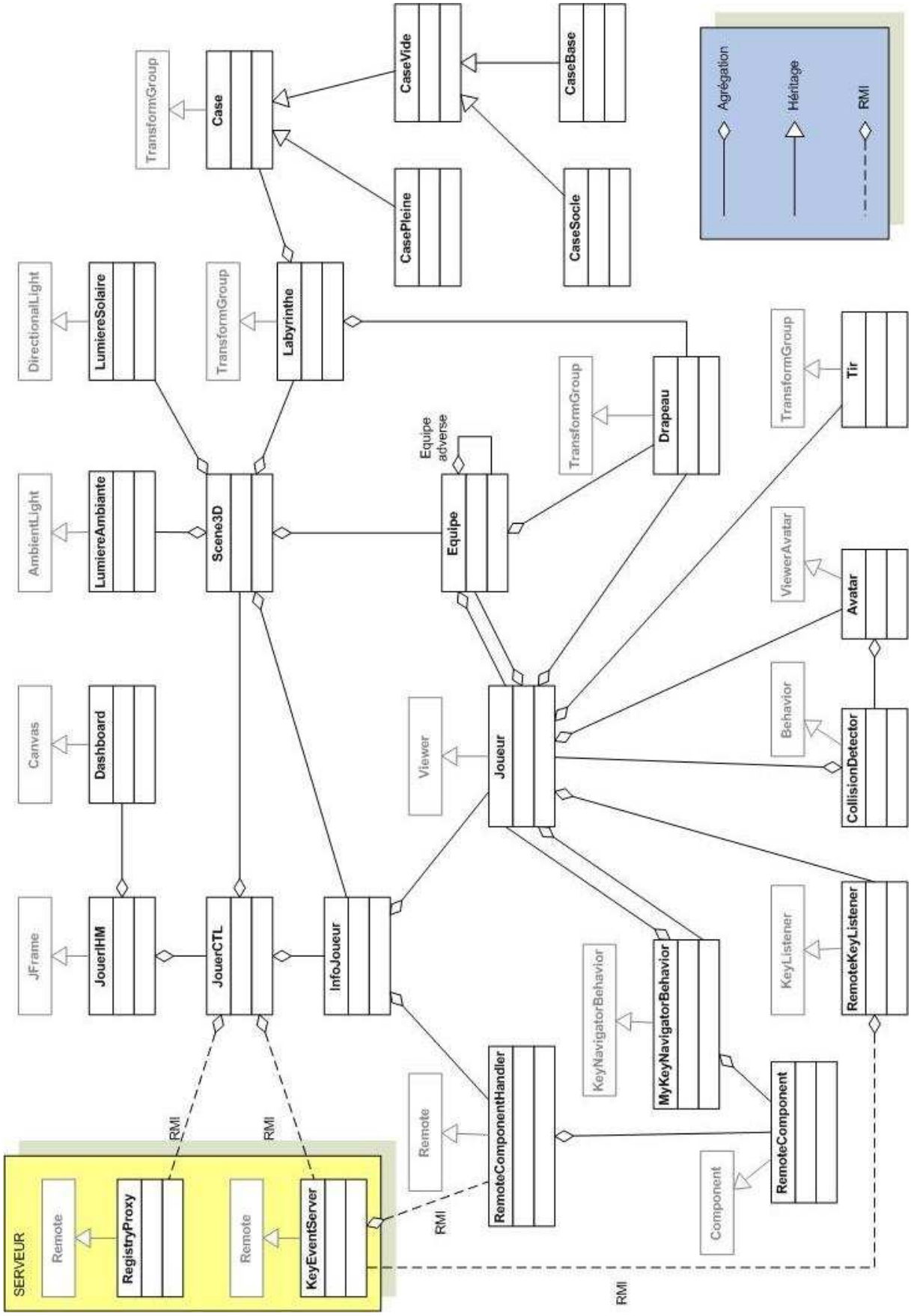
Par rapport à l'analyse UML initiale, ce diagramme présente quelques différences :

- Les use case Jouer et Rejoindre ont fusionnés, pour correspondre au fonctionnement spécifique de Java3D.
- Le use case Créer n'a pas été implémenté de façon graphique, les paramètres sont saisis sur la ligne de commande du serveur et le départ de la partie se fait de façon automatique lorsque le nombre maximal de joueur est atteint (deux par défaut).

De plus, on retrouve dans la partie de gauche, à l'intérieur et autour du cadre *Serveur,* la structure de la mise en réseau via RMI présentée précédemment.

Les classes en gris sont celle fournie par Java3D, sauf la classe `Remote` qui fait partie du package RMI.

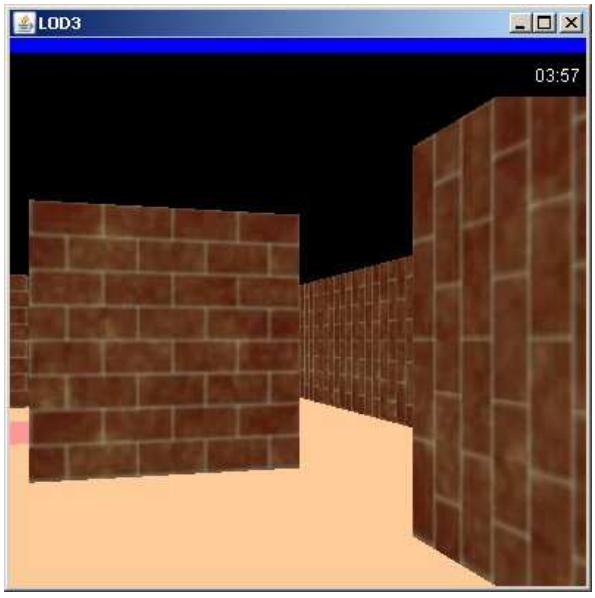
Pour chaque classe, une documentation *Javadoc* a été réalisée, afin de faciliter la compréhension et l'intégration des différents composants, et éventuellement leurs réutilisations dans un autre projet. Un exemple, pour la classe `Equipe`, est présenté en annexe 3.


Agrégation
Héritage
RMI
TransformGroup
Case
CaseVide
CaseBase
CasePleine
CaseSocle
DirectionalLight
LumiereSolaire
TransformGroup
Labyrinthe
TransformGroup
Drapeau
Tir
Equipe adverse
AmbientLight
LumiereAmbiante
Scene3D
Equipe
ViewerAvatar
Avatar
Canvas
Dashboard
Viewer
Joueur
Behavior
CollisionDetector
JFrame
JouerIHM
JouerCTL
InfoJoueur
KeyNavigatorBehavior
MyKeyNavigatorBehavior
KeyListener
RemoteKeyListener
Remote
RemoteComponentHandler
RemoteComponent
Component
SERVEUR
Remote
RegistryProxy
Remote
KeyEventServer
RMI
RMI
RMI
RMI

3. Présentation de l'interface (IHM)

Voici comment se présentent la scène et les éléments du programme.

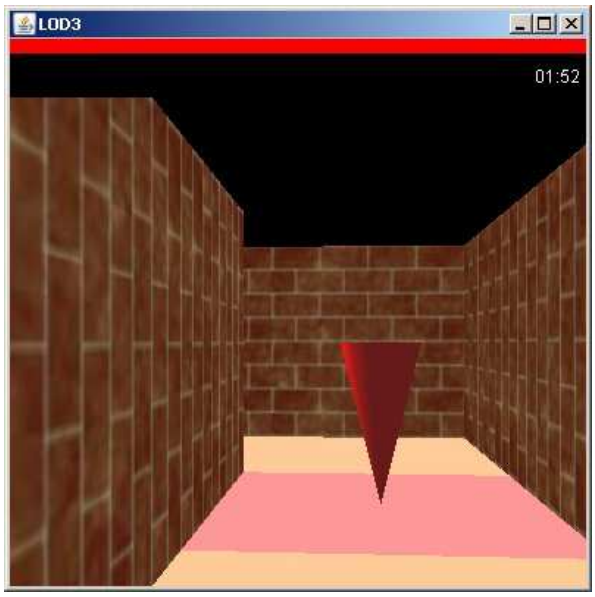
3.1. L'interface, la scène



Deux éléments à noter :

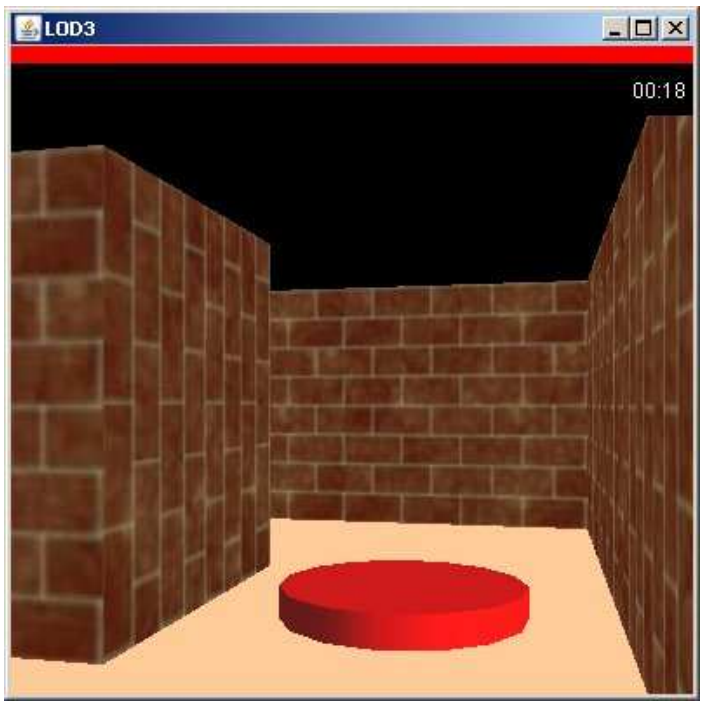
- La barre, en haut de la fenêtre, à la couleur de l'équipe (ici, bleue)
- Le chronomètre affichant le temps de la partie en haut à droite.

3.2. Le drapeau



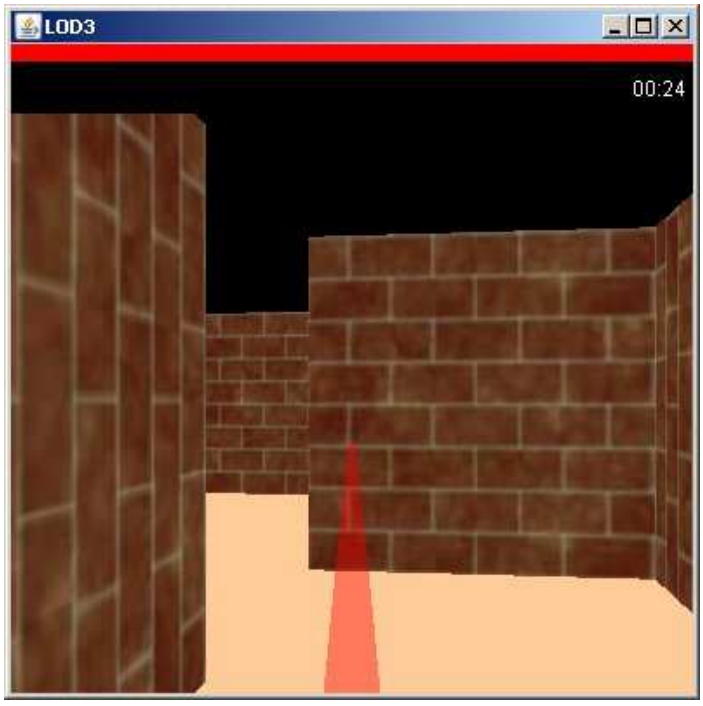
Ici, il s'agit du drapeau de l'équipe rouge à capturer par l'équipe bleue.

3.3. La base



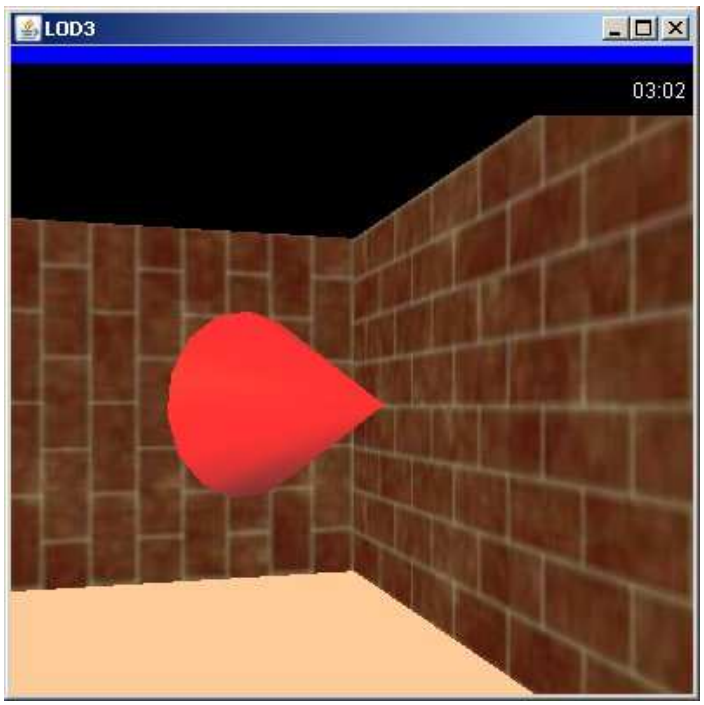
La base de son équipe sur laquelle il faut retourner le drapeau. Point de départ du jeu.

3.4. Le laser



Le laser qui apparaît lorsque nous tirons.

3.5. Le personnage



Le personnage représenté par un cône dont la pointe indique sa direction.

4. Les tests

Cette application a été développée en utilisant la méthode Extreme Programming. A partir de là, nous avons dû suivre une procédure où les tests jouent un rôle très important puisque leur validation permet l'avancement du projet.

La phase de test est une activité très importante. En effet 60 % des défauts sont issus de la phase d'analyse et 50% sont corrigés pendant la phase d'exploitation.

Il faut rappeler qu'un codage peut ne représenter que 10% des ressources de développement, dans une approche moderne d'un projet le reste revient surtout à l'étude, l'intégration et les tests.

Trois types de test ont été retenus pour ce projet :

- tests unitaires
- tests d'intégration
- tests de validation

4.1. Les tests unitaires

Cette partie de la programmation est testée directement après sa programmation. Ces tests doivent correspondre aux spécifications prévues et ne

doivent pas comporter de boucles infinies. Ils doivent vérifier si une valeur rentrée fait sortir une valeur prévue.

Cette application étant développée autour d'une conception orientée objet, chaque classe peut être testé indépendamment. Ces tests, qui peuvent être exécuté de façon automatique, ont été écrits en même temps que la classe concernée.

Chaque classe contient une méthode statique runTest(), permettant d'exécuter un ensemble de tests sur cette même classe et d'afficher le résultat avec notamment :

- des tests de partition sur les paramètres en entrée
- des tests aux limites des variables d'entrées et sorties
- des tests avec des paramètres valides pour contrôler le fonctionnement normal
- des tests avec des valeurs en entrées aléatoires

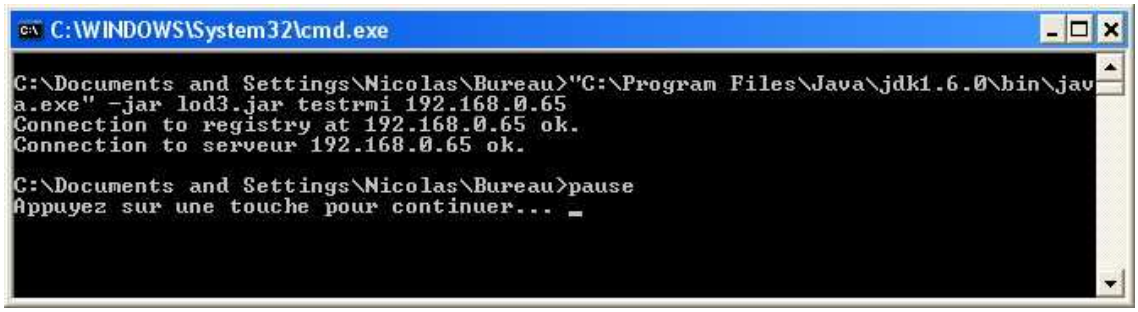
4.2. Les tests d'intégration

Le but de ces tests est de vérifier la cohérence du système. Deux tests d'intégration ont été prévus. Ils sont réalisables par l'administrateur ou l'utilisateur.

4.2.1. Test réseau

Ce test permet d'établir la connexion entre la partie serveur et la partie client et de vérifier le bon fonctionnement des échanges RMI ainsi que la configuration réseaux des machines en jeux.

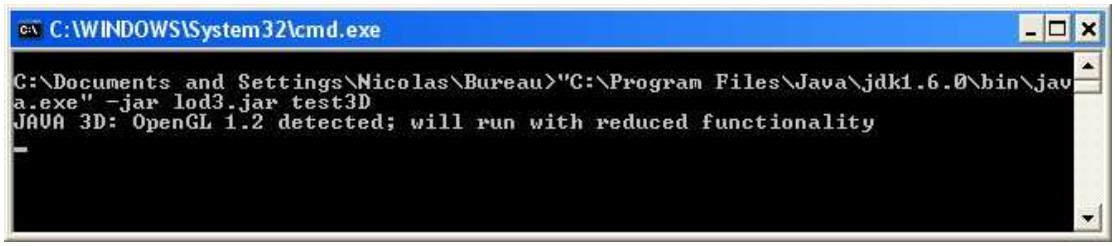
Il est exécuté en spécifiant l'option `testrmi` suivi de l'adresse IP du serveur sur la ligne de commande. Le serveur doit être démarré.



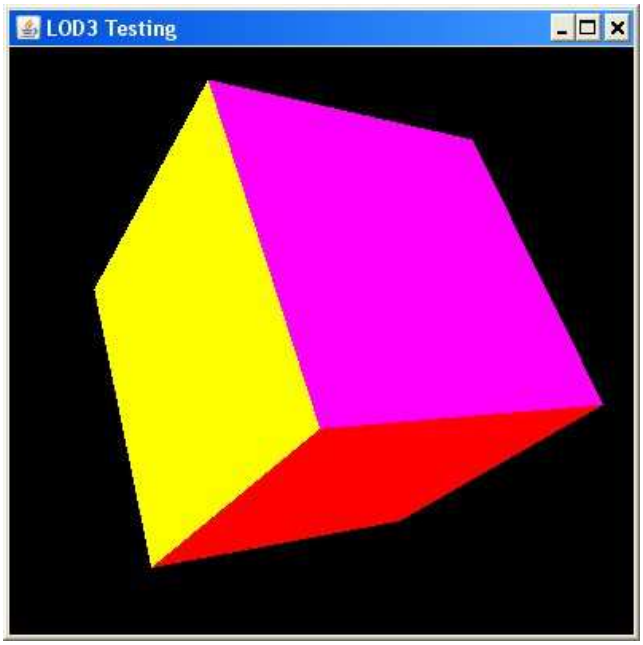
4.2.2. Test de l'API Java 3D

Ce test permet de valider la bonne installation de la machine virtuelle java et des extensions Java3D sur le poste client. Ce test ne nécessite pas la présence du serveur.

Il est exécuté en spécifiant l'option `test3D` sur la ligne de commande.



Résultat attendu :



4.3. Les tests de validation

C'est le cahier des charges et des spécifications qui va déterminer les tests de validations.

La validation correspond au processus d'évaluation à la fin de son développement pour confirmer que le système n'est pas défaillant et qu'il répond aux spécifications des besoins du client.

Ces tests vont ainsi permettre de valider les fonctionnalités réelles de l'application par rapport au cahier des charges.

Deux principaux tests ont été détaillés pour cela.

4.3.1. Lancement de la partie

But du test :

- Vérifier que le serveur est lancé et qu'il communique avec les clients et inversement
- Vérifier que la partie se lance

Déroulement du test :

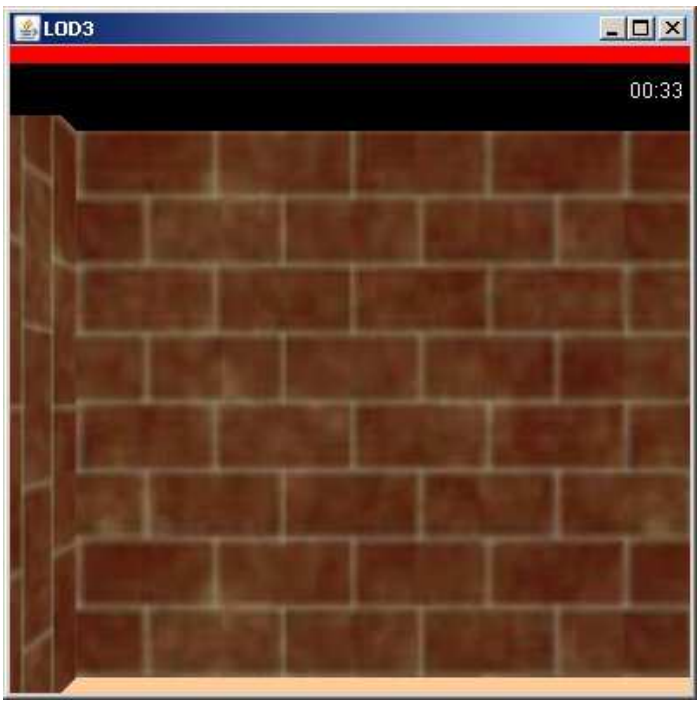
- Nous lançons les programmes exécutant la partie serveur et celle concernant les clients. Le serveur doit être lancé avec le chiffre 1 en option sur la ligne de commande pour n'attendre d'un seul joueur.

- La fenêtre pour indiquer l'adresse IP du serveur apparaît sur le client :

- Nous indiquons l'adresse
- Une nouvelle fenêtre apparaît pour demander le pseudonyme du joueur :

- Nous saisissons le nom du joueur

- Après validation, la partie se lance, la fenêtre suivante doit s'afficher :



4.3.2. Test de contrôle du jeu

But du test :

- Vérifier que les principales commandes fonctionnent

Déroulement du test :

- La partie une fois lancée, nous commençons à exécuter les principales commandes décrites dans le cahier des charges à savoir :

- [CTRL] : tirer
- [UP] : avancer
- [DOWN] : reculer
- [LEFT] : tourner a gauche
- [RIGHT] : tourner a droite
- [SHIFT] : courir
- [ALT] + [LEFT] : se déplacer sur la gauche
- [ALT] + [RIGHT] : se déplacer sur la droite
- [R] : réinitialisation (suicide)
- [ESC] : quitter le jeu

5. Bilan

5.1. Réponse au cahier des charges

En raison des impératifs et contraintes professionnelles et personnelles de chacun, le planning prévisionnel n'a pas pu être tenu dans les termes prévus au départ. De plus, la réorganisation de l'architecture client serveur durant le développement a nécessité une redistribution des taches en cours de projet. Malgré tout, le développement en suivant les méthodes de travail "Extreme Programming" a permis de garder la maîtrise du déroulement du projet.

Au final, le cahier des charges du lot initial a été complété en quasi totalité, seules quelques fonctionnalités secondaires sur le plan technique n'ont pas été réalisées :

- Il n'y a pas de temporisation lorsqu'un personnage est touché. Celui-ci réapparaît immédiatement.
- Lors de la fin de la partie, le récapitulatif ne s'affiche que sur la console et le jeu n'est pas bloqué.

Cependant, certaines fonctionnalités intéressantes des lots optionnels 1 et 3 ont été implémentées :

- L'utilisation de textures (nécessaire pour avoir une bonne vision de la scène en 3D)
- La possibilité d'augmenter le nombre de joueurs au delà de 2 (découle de l'architecture générique utilisée pour la mise en réseau)

5.2. Facteurs de risque

Cette réalisation a présenté plusieurs facteurs de risques pressentis lors de l'étude préliminaire, pour rappel :

- Gestion des collisions (entre les joueurs, les murs et les tirs d'armes)
- Gestion du réseau
- Synchronisation des environnements 3D des différents joueurs par le réseau

Ces risques avaient bien été identifiés, certains d'entre eux ont effectivement posé problèmes.

C'est le cas de la gestion des collisions avec les outils natifs de java3D, qui, en plus d'être *unstables*, ne présentent que des fonctionnalités sommaires. L'utilisation de bibliothèque tierce, comme des moteurs physique, permettrait d'aller au-delà de l'implémentation actuelle. *(Cf. Gestion des collisions)*

Concernant la mise en réseau multi joueur et la synchronisation des environnements, les deux sont dans les faits intimement liés. La mise au point de l'architecture utilisant la technologie RMI et son intégration au coeur du système AWT, en jouant avec l'héritage, a permis de répondre à ce problème avec élégance et efficacité. *(Cf. Mise en réseau)*

En conclusion, nous pouvons dire que globalement, ce projet a rempli le contrat qui était de fournir un jeu 3D de type FPS fonctionnant en réseau. Même si l'application finale n'a pas satisfait pleinement et scrupuleusement le cahier des charges, le résultat est quand même au rendez-vous. Nous avons bien ce jeu 3D opérationnel dont le nom est "LoD³" et dont l'avenir est très prometteur.

LASER GAME 3D

LoD³ - Manuel d'utilisation

1. Le jeu

Lâché dans un labyrinthe, vous devez aller récupérer un drapeau. Il s'agit du drapeau de votre ennemi qui fera tout pour vous empêcher d'arriver à vos fins. Le drapeau se situe sur un socle quelque part dans le labyrinthe. Un endroit que vous devez trouver !

Une fois que vous avez trouvé le drapeau, après l'avoir récupéré, vous devez le ramener à votre point de départ, en quelque sorte sur votre territoire, votre Q.G. représenté de la même façon que votre adversaire par un drapeau planté dans son socle.

N'oubliez pas, non plus, que vous aussi, vous avez un drapeau à protéger. Vous devez, bien évidemment, empêcher votre adversaire de capturer et de ramener votre drapeau à son Q.G..

Pour empêcher votre adversaire de récupérer votre drapeau, ou bien, une fois capturé, l'empêcher de le ramener à son Q.G., vous pouvez le neutraliser grâce à l'arme laser qui est en votre possession.

2. Pré requis

Pour pouvoir exécuter ce programme, sous Linux ou Windows, pour devez avoir installé, sur votre machine, le logiciel Java disponible ici :

http://www.java.com/fr/

Ainsi que les bibliothèques Java3D, disponible ici :

https://java3d.dev.java.net/

De plus, pour obtenir un bon rendu graphique une carte graphique 3D est conseillée.

Et, si vous souhaiter jouer à plusieurs en réseaux, vous aurez besoin d'une carte réseau correctement configurée, sans firewall.

3. Lancement d'une partie

Les binaires de ce programme sont présentés sous la forme de deux archives exécutable JAR, une pour le serveur (`lod3-srv.jar`), l'autre pour le jeu (`lod3.jar`).

Avant de lancer le jeu, il vous faut d'abord démarrer le serveur.

Ceci peut être fait en double cliquant sur l'archive lod3-srv.jar si vous souhaiter les paramètres par défaut, ou sur la ligne de commande par :

```
java –jar lod3-srv.jar N
```

Avec N le nombre de joueur dans la partie.

Pour rejoindre une partie, il suffit d'exécuter l'archive `lod3.jar` et ensuite de choisir le serveur qui a lancé une partie en indiquant son adresse IP, puis de saisir un pseudo.

4. Environnement du jeu

La vision de l'environnement du jeu est à la première personne, vous ne voyez pas le personnage que vous incarnez.

De plus, le pistolet laser n'apparaît pas à l'écran. Il n'y a que la trace du tir du laser qui est visible.

Dans le déroulement du jeu, lorsqu'un joueur capture un drapeau, un voyant (de la couleur du drapeau) est affiché à l'écran. Ce voyant est aussi affiché sur l'écran de l'adversaire, l'avertissant ainsi que son drapeau a été capturé.

Un chronomètre est affiché en haut à droite de l'écran afin d'indiquer le temps passé pour la partie en cours.

5. Les commandes

En cours de partie :

- [CTRL] : tirer
- [UP] : avancer
- [DOWN] : reculer
- [LEFT] : tourner a gauche
- [RIGHT] : tourner a droite
- [SHIFT] : courir
- [ALT] + [LEFT] : se déplacer sur la gauche
- [ALT] + [RIGHT] : se déplacer sur la droite
- [R] : réinitialisation (suicide)
- [ESC] : quitter le jeu

6. Règles

Pour capturer le drapeau, le joueur doit simplement entrer en collision avec l'objet le représentant. Ce dernier disparaît de son socle. Le joueur ne peut prendre que le drapeau de l'équipe adverse.

Chaque joueur dispose d'une arme laser pour tirer sur l'adversaire. Lorsque qu'un joueur est touché par le laser adverse, il meurt. Le joueur est "mort" et ne peut plus joué pendant une durée de 5 secondes. Il est désactivé. Pendant ce laps de temps, il ne voit pas ce qu'il se passe dans le jeu. Ensuite il réapparaît à l'emplacement de la base de son équipe et peut recommencer à jouer.

Il n'y a pas de limite sur le nombre de tir possible avec l'arme laser.

Une fois pris, le drapeau ne peut pas être reposé, sauf dans le cas où le joueur est touché par le laser adverse. A ce moment-là, le joueur meurt dans les mêmes conditions évoquées ci-dessus, et le drapeau, quant à lui, retourne sur son socle d'origine.

La partie est gagnée et terminée lorsqu'une équipe a rapporté le drapeau adverse sur son socle. Un récapitulatif des scores de l'ensemble des joueurs est alors affiché (temps de la partie et nombre de fois mort pour chaque joueur).

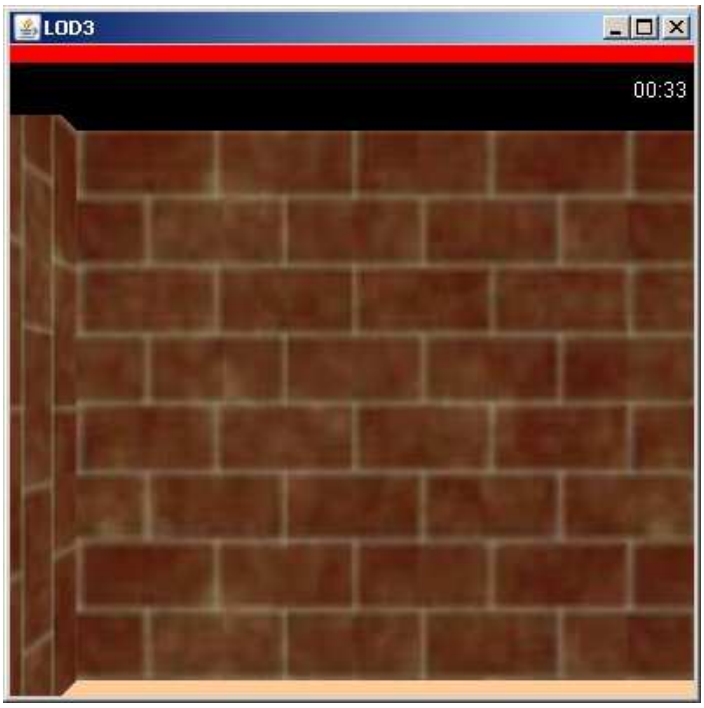
7. Stratégie

Afin de remporter le jeu, plusieurs stratégies sont possibles :

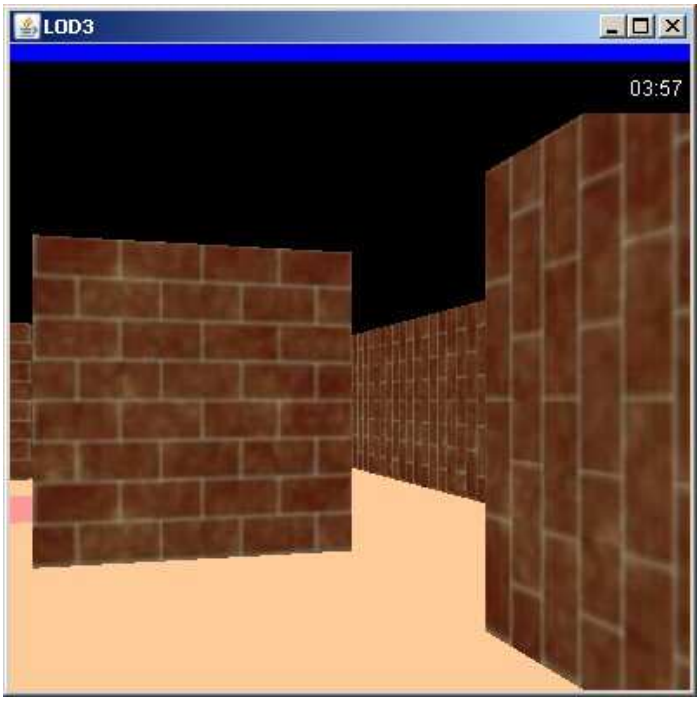
- Vous pouvez avoir une stratégie défensive en empêchant avant tout que l’on capture votre drapeau.
- Vous pouvez avoir une stratégie d’attaque. Vous allez au plus vite récupérer le drapeau en n’hésitant pas éliminer votre adversaire sur le chemin.
- Vous pouvez combiner ces deux premières stratégies, c’est-à-dire, par exemple, courir récupérer de suite le drapeau de votre adversaire puis empêcher votre adversaire de ramener votre drapeau sur son socle.

8. Les écrans

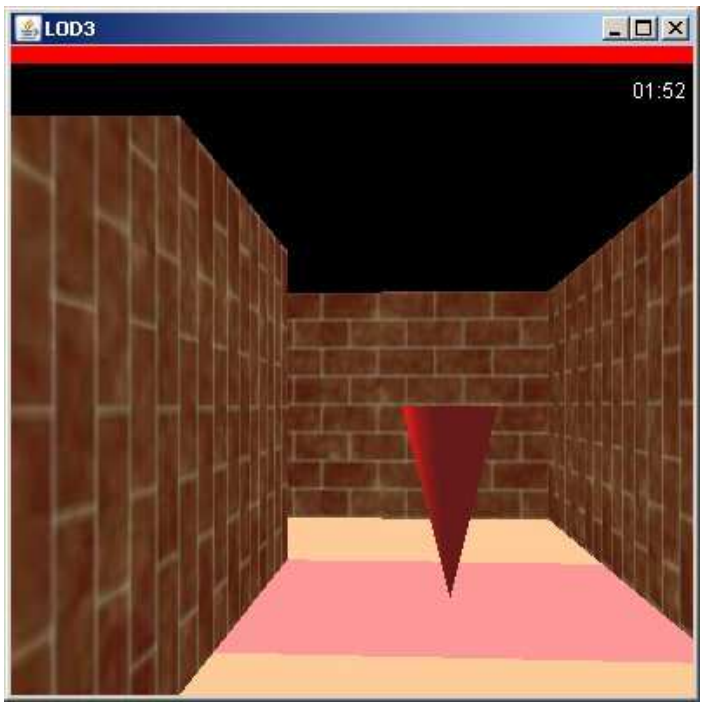
- Votre écran au lancement du jeu :



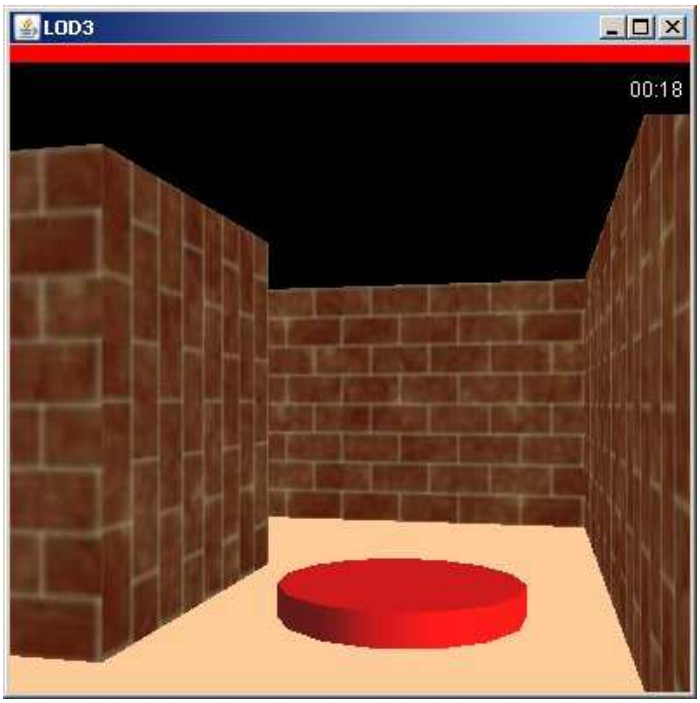
- Le labyrinthe, lorsque vous avancer dans celui-ci :



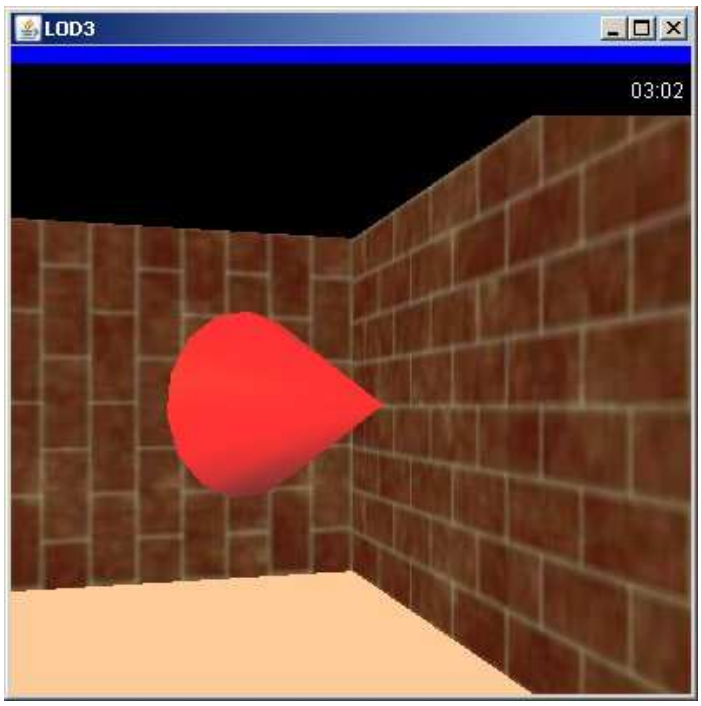
- Le drapeau à capturer (ici, le drapeau rouge à capturer par l'équipe bleue) :



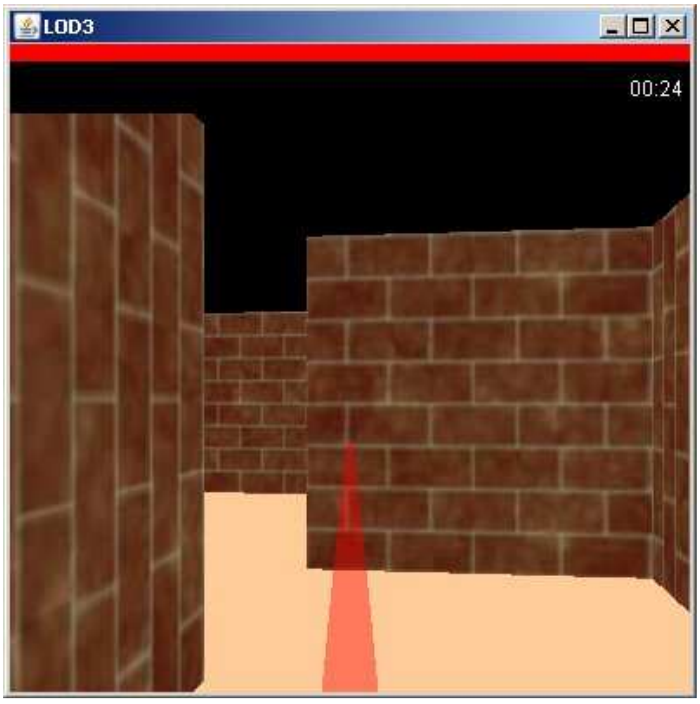
- Le socle sur lequel vous devez ramener le drapeau :



- Votre personnage (ou votre adversaire) :



- Le laser lorsque vous tirez :



9. Précautions

Evitez de jouer si vous êtes fatigué ou si vous manquez de sommeil. Assurez-vous que vous jouez dans une pièce bien éclairée en modérant la luminosité de votre écran. En cours d'utilisation, faites des pauses des dix à quinze minutes toutes les heures.

Avertissement sur l'épilepsie (décret n°96-360 du 23 avril 1996 relatif aux mises en garde concernant les jeux vidéo). Certaines personnes sont susceptibles de faire des crises d'épilepsie lorsqu'elles sont exposées à diverses stimulations lumineuses. Ces personnes s'exposent à des crises lorsqu'elles regardent la télévision ou jouent à certains jeux vidéo. Même si vous n'avez jamais été sujet à des crises d'épilepsie, vous pouvez être épileptique sans le savoir. Si vous êtes épileptique, consultez votre médecin avant de jouer à un jeu vidéo, ou immédiatement si vous présentez l'un des symptômes suivants lorsque vous jouez : vertige, troubles de la vision, contraction musculaire, mouvement involontaire, troubles de l'orientation, perte momentanée de conscience ou convulsion.

10. Mentions légales

Ce programme, ainsi que son code source et tous ses éléments constitutifs, sont soumis au gauche d'auteurs. L'utilisation, la modification et la distribution de ce programme sont autorisés conformément aux dispositions de la GNU General Public Licence, version 2 ou plus.

LASER GAME 3D

Annexes