IMAC
Charles Cros

RAPPORT DE PROJET JAVA

CROLLYWOOD

Présenté et soutenu le 09 juin 2006 à l'Institut Charles Cros, Serris

Jonathan **BABY**
Julie **BILLARD**
Thomas **BOURGOIN**
François **BRAUD**
Catherine **CHANTHAVONG**
Sandra **DESROUSSEAUX**
Clément **DONY**
Séverine **FAYOLLE**
Anne-Laure **FOUQUE**
Stéphanie **JUILLARD**
Lucie **MOINEAU**

Tuteurs enseignants : **M. Pascal CHAUDEYRAC**
M. Venceslas BIRI

IMAC – Promotion 2007
Institut Charles Cros
Université de Marne-la-Vallée

RAPPORT
DE
PROJET JAVA

CROLLYWOOD

Présenté et soutenu le 09 juin 2006 à l'Institut Charles Cros, Serris

Jonathan **BABY**
Julie **BILLARD**
Thomas **BOURGOIN**
François **BRAUD**
Catherine **CHANTHAVONG**
Sandra **DESROUSSEAUX**
Clément **DONY**
Séverine **FAYOLLE**
Anne-Laure **FOUQUE**
Stéphanie **JUILLARD**
Lucie **MOINEAU**

Tuteurs enseignants : **M. Pascal CHAUDEYRAC**
M. Venceslas BIRI

IMAC – Promotion 2007
Institut Charles Cros
Université de Marne-la-Vallée

REMERCIEMENTS

Nous tenons à remercier nos professeurs M. Biri et M. Chaudeyrac pour leurs conseils tout au long du projet, pour leur patience et leur disponibilité.

Nous remercions également tous nos camarades de la promotion IMAC 2007 pour leur aide, leur bonne humeur et leur soutien.

TABLE DES MATIERES

INTRODUCTION

Ce projet a pour but de nous faire réaliser un environnement virtuel en 3D dans lequel on peut évoluer.
Ce programme est implémenté en Java et utilise les technologies Java 3D et RMI.
Le but de l'application n'était pas imposé mais celle-ci devait remplir certaines conditions, notamment la gestion des collisions, de l'ambiance sonore, du temps…

De même que pour notre projet de PHP, le groupe de projet était constitué de 11 étudiants IMAC 2ème année. Cela nous a permis d'aborder la gestion de projet avec une équipe conséquente.

Ce rapport présente le travail que nous avons effectué depuis 2 mois et demi.
Tout d'abord, nous présenterons le cahier des charges puis nous aborderons l'organisation du projet, ensuite nous expliquerons les différentes phases de sa conception et enfin nous exposerons les résultats.

I. Cahier des Charges

1. Contraintes du projet

Le projet devait consister en une application Java permettant de naviguer dans un environnement virtuel en 3D. Cet environnement est en fait une scène où se déplacent des personnages contrôlés par des utilisateurs connectés via le réseau RMI. Une gestion du temps doit être intégrée ainsi que l'éclairage et le son.

Nous devions déterminer le but de ce projet tout en respectant ces contraintes.

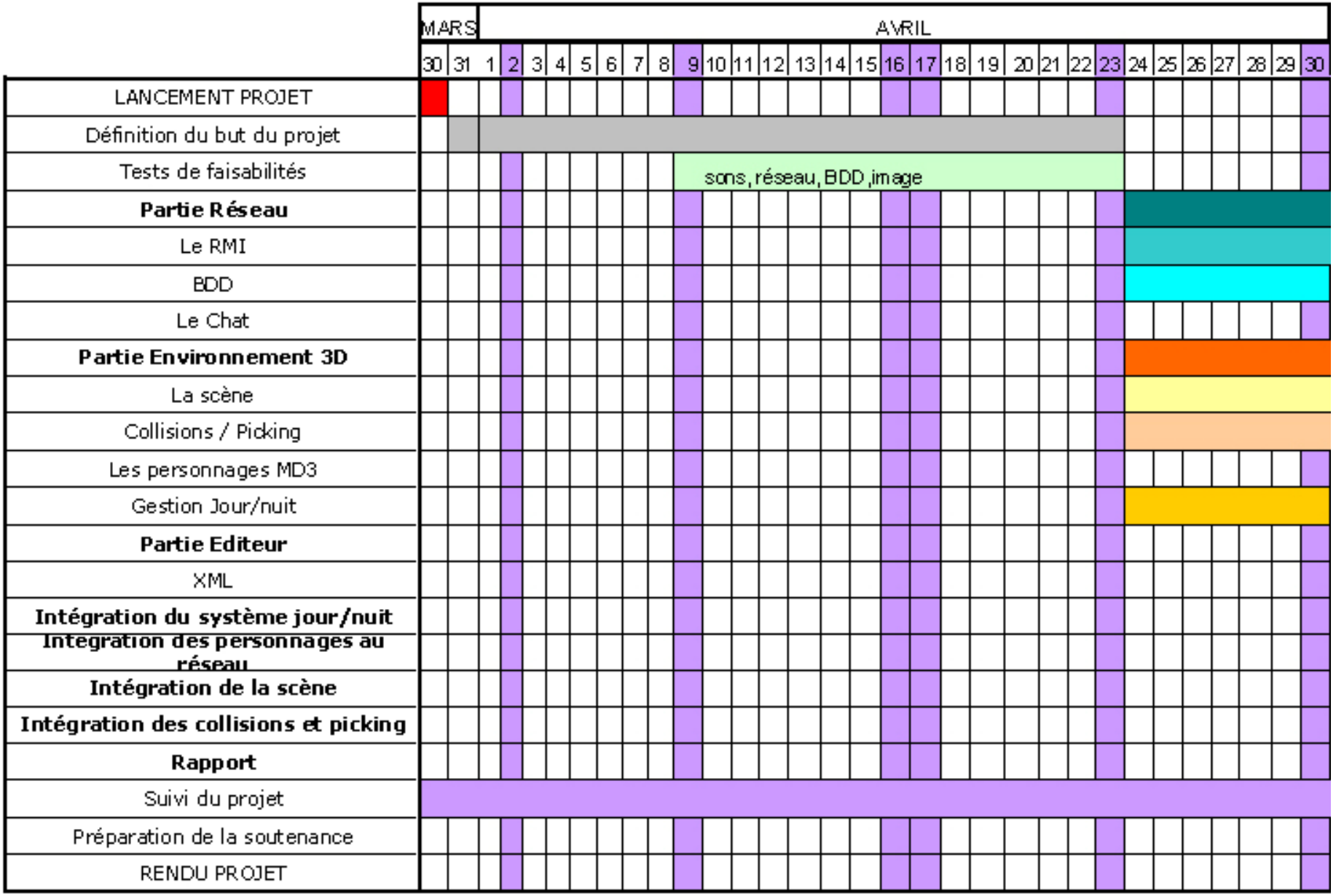
2. Idée de départ

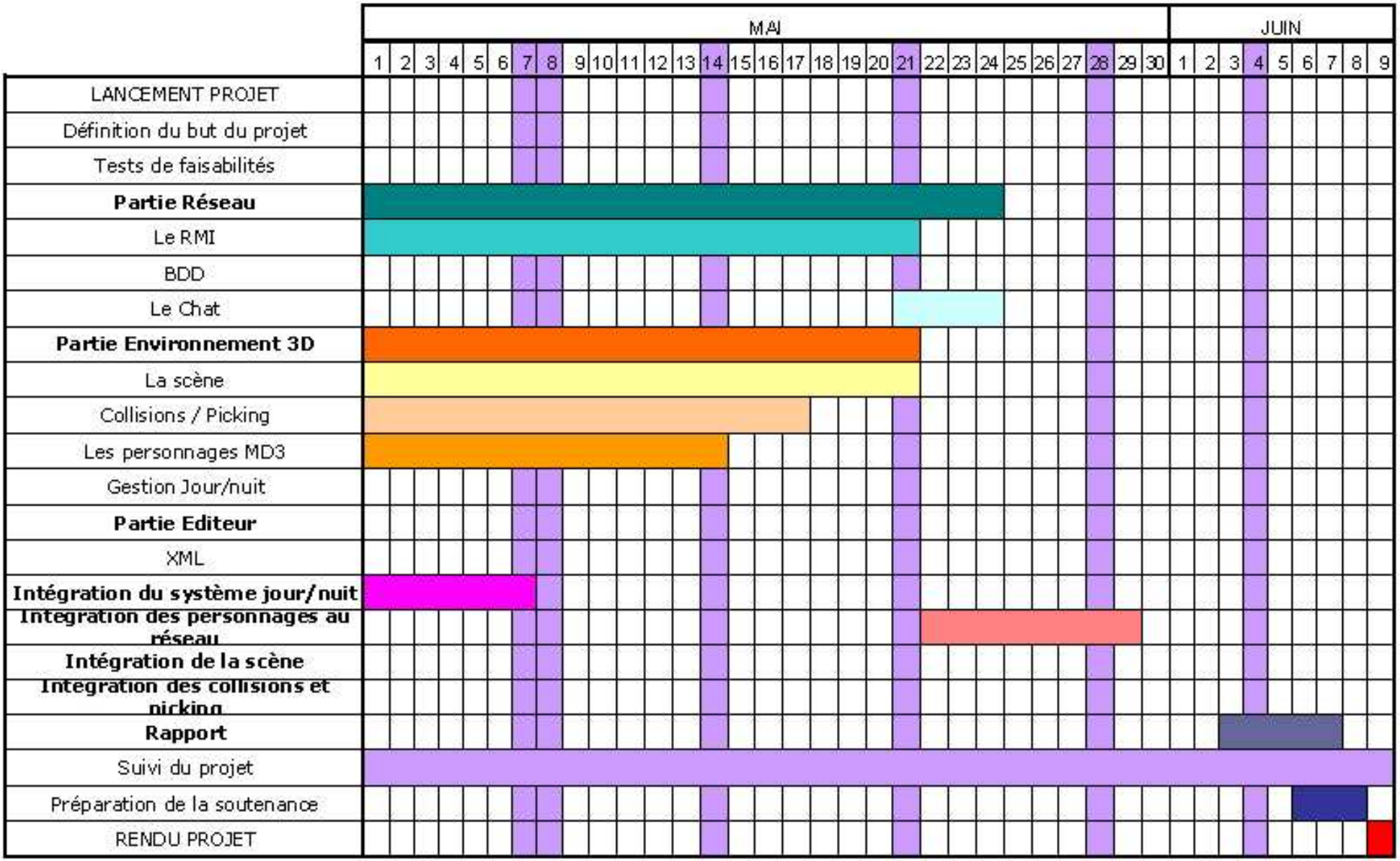
Nous avons décidé de créer une application qui permet de se déplacer dans un monde virtuel, une ville, où il est possible de pénétrer dans des bâtiments (maisons, cinéma, ..) afin d'y découvrir les oeuvres appartenant à d'autres utilisateurs. En effet, l'application permet grâce à un éditeur de créer un espace permettant d'exposer différents travaux comme des vidéos ou des sons. Les utilisateurs se connectent à l'application par le réseau RMI et ont la possibilité de discuter entre eux avec le Chat.

3. Fonctionnalité de base

L'utilisateur doit pouvoir se connecter facilement et utiliser le programme de la manière la plus simple. Il doit pouvoir se déplacer, discuter via le chat et utiliser les fonctionnalités de l'application de manière intuitive. Il doit pouvoir se déconnecter facilement.
L'utilisateur doit être enregistré dans une base de données pour pouvoir utiliser l'application.

4. Rétro planning


MARS
AVRIL
30 31 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
LANCEMENT PROJET
Définition du but du projet
Tests de faisabilités
sons, réseau, BDD, image
Partie Réseau
Le RMI
BDD
Le Chat
Partie Environnement 3D
La scène
Collisions / Picking
Les personnages MD3
Gestion Jour/nuit
Partie Editeur
XML
Intégration du système jour/nuit
Intégration des personnages au réseau
Intégration de la scène
Intégration des collisions et picking
Rapport
Suivi du projet
Préparation de la soutenance
RENDU PROJET


MAI
JUIN
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 1 2 3 4 5 6 7 8 9
LANCEMENT PROJET
Définition du but du projet
Tests de faisabilités
Partie Réseau
Le RMI
BDD
Le Chat
Partie Environnement 3D
La scène
Collisions / Picking
Les personnages MD3
Gestion Jour/nuit
Partie Editeur
XML
Intégration du système jour/nuit
Intégration des personnages au réseau
Intégration de la scène
Intégration des collisions et picking
Rapport
Suivi du projet
Préparation de la soutenance
RENDU PROJET

II. Organisation du projet

1. Le partage des modules

Ce projet faisant intervenir plusieurs parties distinctes, nous avons décidé de nous diviser en trois groupes. Le premier constitué de Stéphanie, Lucie et Anne-Laure ont travaillé sur la partie réseau du projet tandis que Clément et Jonathan se sont chargé du travail sur l'éditeur. Julie, Séverine, Sandra, François et Catherine ont, quant à eux, travaillé sur l'environnement 3D de la scène, qui comprenait de nombreux modules.

La partie réseau est composée de trois parties : Stéphanie et Lucie ont travaillé sur le RMI et sur la base de données tandis qu'Anne-Laure s'est occupé du Chat.

Le module environnement 3D comporte plusieurs parties. Julie a été en charge de la modélisation des décors de l'environnement. Sandra et Catherine ont intégré ces décors à une scène 3D JAVA et ont également géré le picking et les collisions. François les a aidé à régler des problèmes de vitesse de déplacement pour les collisions. Il s'est aussi occupé de la gestion du temps dans la scène (jour/nuit) et du chargement de fichiers XML pour l'intérieur des maisons.
Séverine s'est occupée des personnages MD3, de leur chargement, animation et déplacement. Elle a également travaillé sur le changement d'environnement.

Afin de faciliter la communication entre ces trois groupes, nous avons décidé de nommer un responsable pour chaque module : Catherine pour le module environnement 3D, Stéphanie pour le module réseau et Jonathan pour l'éditeur.
Parmi les trois responsables, nous avons choisi un chef de projet : Stéphanie.
De même, étant donné que nous sommes très nombreux dans le groupe, deux interlocuteurs ont été choisis parmi les dix pour dialoguer avec le tuteur du projet, M. Chaudeyrac. Il s'agit de Stéphanie et Lucie.

2. L'organisation du groupe

Composition du groupe

La classe a été scindée en trois groupes de onze personnes, les mêmes groupes que pour le projet de PHP.

De même que pour le projet de PHP que nous avons réalisé en parallèle, il s'agissait du premier projet, dans le cadre de notre scolarité à l'IMAC, où nous devions composer avec un groupe d'une telle envergure, et dont nous n'avions pas le choix des membres.

Les contraintes présentes dans le sujet

Le projet comporte donc une importante composante de gestion de groupe et d'organisation générale du travail, d'autant que certaines contraintes étaient fixées par le sujet.

Lieu et groupes de travail

Tous les membres du groupe ne logeant pas à proximité de l'université et étant donné que nous étions amenés à avancer notre travail le soir, environ la moitié du développement s'est effectué dans les locaux de l'institut, et l'autre moitié au domicile de chacun.

Ceux du groupe se sentant plus à l'aise pour développer à deux ont pu le faire, notamment Stéphanie et Lucie, Clément et Jonathan, ou Sandra et Catherine. Séverine, Anne-Laure et François ont davantage développé de façon autonome. Toutefois, ils ont été présents lors de l'intégration de leur module à ceux des autres.

Julie a développé chez elle mais communiquait régulièrement avec le groupe pour savoir ce qu'elle devait faire.

Suivi des avancées de chacun

Le travail à onze a nécessité une organisation et une communication efficaces.

- *Les réunions*

Pour cela, nous avons effectué des réunions hebdomadaires. Etant prévues à un moment invariant de la semaine, cela permettait d'organiser nos projets ou activités annexes en conséquence. Ainsi, à chaque, réunion la quasi-totalité des membres était présente.

Les réunions duraient environ deux heures. En début de séance, un tour de table était fait concernant les avancées et ce qui restait à faire pour chacun des membres. Chacun expliquait également les problèmes auxquels il se heurtait en vue de trouver une solution à plusieurs, ou lorsque l'accord de tous les membres du groupe était nécessaire à une décision. En fin de séance, nous fixions les tâches à réaliser d'ici la prochaine séance, et plus globalement d'ici la fin du projet, afin de se donner un aperçu général de notre progression.

Chaque réunion donnait lieu à un compte-rendu rédigé par Stéphanie qui prenait des notes durant nos séances. Ainsi, Julie pouvait obtenir un détail des points qui avaient été abordés, ainsi que des responsabilités de chacun, ce qui permettait à tous de s'adresser à la bonne personne en cas d'interrogation sur un des points du projet. Parfois, nous avons téléphoné à Julie afin de savoir si le module qui lui avait été attribué lui convenait. Souvent, au cours des séances, nous avons également répondu collectivement à ses mails posant des questions sur le projet.

- *La « communication numérique »*

Stéphanie et Lucie ont créé un blog dont l'adresse est la suivante : http://lesptitsmalins.free.fr/

Cela a permis de tenir au courant les membres du groupe ainsi que le professeur des avancées de chacun, et ceci de façon détaillée. En effet, les comptes rendus de réunions étaient plus concis. De plus, à l'aide du blog, nous pouvions retrouver le post qui nous intéresse, sans parcourir l'intégralité de notre boîte mail, ce qui constitue un gain de temps appréciable.

Communication au sein du groupe

- *Points positifs*

Globalement, la communication ne fut pas le point faible de notre groupe. Nous sommes tous capables de dire avec précision qui fut en charge de quelle tâche.

En effet, les réunions régulières, la rédaction de comptes-rendus de fin de séance et l'utilisation active du blog ont permis une bonne visibilité du travail de chacun et des avancées du groupe.

- *Points négatifs*

La communication par mail a bien fonctionné, même si l'envoi au groupe a parfois posé problème : en effet, face à la réception de dizaines de messages journaliers, certains restaient sans réponse. Heureusement, les réunions permettaient d'aborder les problèmes non résolus.

Par ailleurs, lors des réunions, des conflits ont parfois eu lieu. Néanmoins, nous sommes arrivés à recadrer les discussions quand il le fallait.

Gestion du télétravail

- *De notre coté*

Travailler avec Julie ne fut en aucun cas une contrainte. En effet, cela a obligé à la mise en place de comptes-rendus et d'une communication efficace, ce qui fut profitable à tout le groupe. De plus, elle nous communiquait régulièrement ses avancées. Ainsi, nous n'avions par l'impression de travailler à distance.

- *Témoignage de Julie*

Pour ce projet, j'ai eu plusieurs tâches attribuées, mais je n'ai cependant pas pu m'investir autant que je le voulais dans ce projet. Lors d'une des premières réunions on m'a confié la réalisation de la scène 3D, mission que j'ai pu remplir. Par la suite il était prévu que j'intervienne là où le besoin se faisait sentir. Dans un deuxième temps, on m'a donc demandé de travailler avec François mais il avait malheureusement déjà fini sa partie sur la gestion du jour et de la nuit. On m'a donc confié un module pour le chargement de vidéo, module qui fut abandonné car je n'ai pas réussi à trouver assez d'éléments pour le faire. Ceci est dû en partie à la difficulté que j'ai ressentie de travailler et coder seule sur un programme que je n'avais pas compris, mais aussi au fait que je ne savait pas vraiment ce que je devais faire et comment le faire pour que cela réponde aux besoins du projet. Je dirais donc que la communication a été un peu compliquée, peut être à

cause de la façon dont nous avons appréhendée le projet et de la difficulté que nous avons eu à trouver des tâches qui me convenait.

III. CONCEPTION

1. Tests de faisabilité

a) Le son

Les recherches son ont été effectuées par François et Sandra. Elles ont donné lieu à un document Word envoyé à l'ensemble du groupe, et expliquant de façon détaillée comment charger un son dans une application Java. Par ailleurs, un programme a été réalisé en vue de charger un fichier son sur une page Web utilisant le mécanisme des applets Java.

Quelques points importants ont été dégagés. Pour commencer, afin d'être correctement lu par Java, un fichier son doit être au format Sun (.au, µlaw). Le transfert du format .wav vers un .au peut se faire à l'aide de logiciels comme Goldwave. Ensuite, comme nous l'avons dit, la classe implémentant la lecture du son doit réaliser un import de java.applet.* et hériter de Applet.

Les objets sonores manipulés sont de type AudioClip. Le nom du fichier à charger est placé dans la méthode getParameter("NOMSON") renvoyant une chaîne s, que l'on passe en paramètre de la méthode getAudioClip(getDocumentBase(), s) permettant de charger le son à l'url désignée par getDocumentBase(). Pour jouer le son, on utilise play(), et stop() pour l'arrêter. On peut le jouer en continu grâce à loop().

On intègre par la suite l'applet à la page en ajoutant dans le code html :

```
<applet code="NomClasse.class" >
<param name="NOMSON" value="nomson.au">
</applet>
```

Nous avons aussi trouvé sur Internet (*http://www.javafr.com/codes/JOUER-SON-WAV-JAVA_29515.aspx*) et retravaillé une classe java qui charge un son au format wav, et qui peut le jouer avec une méthode play. Cette classe utilise de manière interne des objets tels que AudioInputStream, et AudioFormat, et DataInputStream.

L'avantage de cette classe est qu'elle permet de charger des sons au format wav, format qui est plus connu pour l'utilisateur final que le format au. De plus le programme utilisant cette classe n'est pas forcément une applet.

b) L'image

Recherche et choix du loader MD3

Nous souhaitions que les avatars des utilisateurs soient animés afin de rendre l'univers 3D plus réaliste. Par conséquent, nous avons orienté notre recherche vers un loader MD3 permettant d'utiliser les animations prévues dans la structure des fichiers.

La phase la plus compliquée était de trouver un loader MD3 codé en java dont le code était suffisamment bien organisé pour pouvoir être utilisé dans un autre programme.

Le premier loader MD3 intéressant trouvé était très complet : il récupérait toutes les informations du MD3 ce qui permettait tous les affichages possibles du personnage (en filaire, en polygones, texturé, non texturé, non animé, animé manuellement, animé en continu, toutes les animations à la suite, une animation particulière, choix de la texture parmi les textures disponible). L'inconvénient était que l'affichage se faisait grâce à OpenGL : utiliser ce loader nécessitait que l'on fasse tout le projet en OpenGL, ce qui complexifiait énormément le projet puisqu'il aurait fallu coder nous-même des outils qui existent déjà en Java3D (ex : picking). Une autre solution aurait été de repasser tout le code OpenGL du loader en Java3D, mais vu la complexité du code, cela aurait été trop long à comprendre et à adapter.

Le second loader MD3 retenu était un peu moins complet que le précédent mais implémentait les fonctionnalités que nous désirions à savoir la création d'un objet 3d représentant le personnage 3d, la texturisation de celui-ci et enfin l'animation image clé par image clé du personnage. Les problèmes rencontrés avec ce code étaient la mauvaise utilisation de certaines classes qui provoquait des erreurs au moment du chargement des différents fichiers composant le MD3, ainsi que l'impossibilité de charger des textures au format TGA (ce qui correspond au format de texture de la majorité des MD3). Le premier problème fut assez simple à corriger, par contre, pour le second problème, il a fallu trouver un loader TGA puisque le loader d'image de Java ne gère pas ce format. Nous avons réussi à trouver un loader TGA simple d'utilisation qui nous a donc permis de compléter assez facilement le code de chargement des textures afin que celui-ci puisse charger les textures JPEG et les textures TGA. Le seul inconvénient qui demeure est que certaines textures TGA ne se chargent pas très bien, notamment la texture de la tête du personnage Harley (problème également rencontré lors du premier semestre avec le loader MD3 C++, mais ce ne sont pas les mêmes textures qui s'affichent mal).

Harley chargée à l'aide du loader MD3 java

Harley chargée à l'aide du loader MD3 C++ du premier semestre

c) Le réseau : choix du RMI

Le but de la partie réseau du projet est de faire transiter des objets qui peuvent être stockés aussi bien sur le client que sur le serveur. Nous ne savions pas quelle méthode utiliser pour implémenter la solution. Nous avons fait des recherches sur Internet concernant Java et le réseau qui nous ont amenées à 2 solutions possibles : les sockets TCP et UDP ou le RMI (*Remote Method Invocation).*

Pour commencer, nous avons fait quelques petits tests avec les librairies telles que java.net.* afin de tester des classes comme NetworkInterface, InetAddress ou encore HttpURLConnection. Ces premiers tests nous ont permis d'afficher les adresses ip de la machine, l'adresse localhost ou même encore de nous connecter à une url afin de récupérer des informations par http.

Nous avons ensuite testé les sockets TCP et UDP et le RMI sur des petits programmes simples. Ces recherches nous ont permis de comprendre que pour notre projet, nous devions utiliser le RMI, qui est plus simple à développer et qui convient plus à notre besoin.

Nous avons donc créé un premier programme avec le Rmi pour afficher la date. Ce programme contenait 4 classes, ce sont les classes de base à implémenter pour le RMI.

Les deux premières classes nécessaires sont : l'interface ne contenant que la déclaration des méthodes qui doit hériter de *java.rmi.Remote et la classe* qui utilise l'interface précédente mais qui implémente les méthodes décrites dans l'interface.

Voici le fonctionnement du RMI avec appel de méthodes distantes :

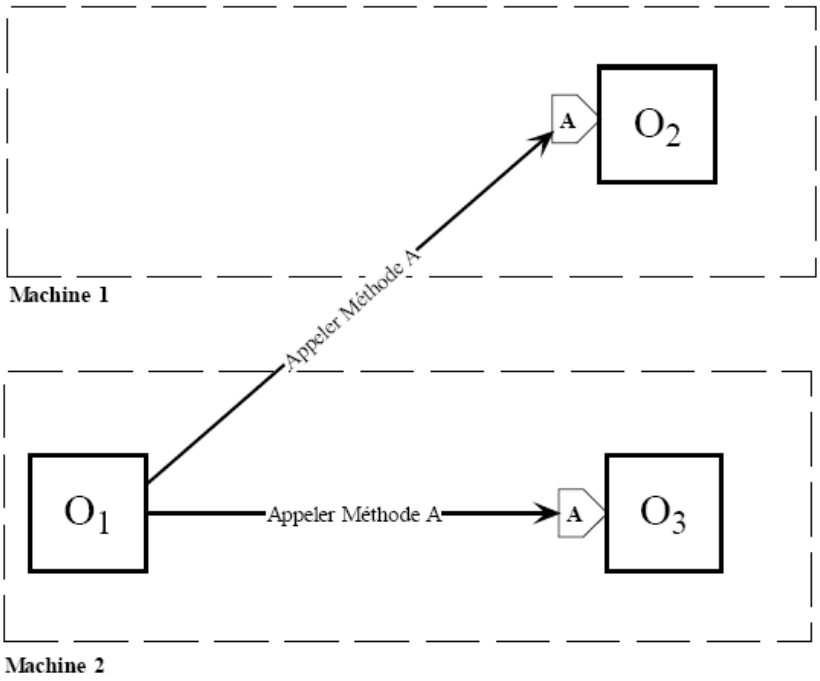


Figure 5 : Appel de méthodes distantes

Les deux autres classes sont la classe pour le serveur qui va créer une instance de la classe *RemoteDateImpl* et la classe DateClient qui va se connecter au client et récupérer les informations implémentées dans l'interface côté serveur.

Ce premier test nous a permis de voir que l'on pouvait passer n'importe quelle sorte d'objet par le RMI à condition que ces méthodes soient décrites dans l'interface.

d) La BDD

Nous avions prévu d'enregistrer certaines informations dans une base de données. Notre BDD est une base PostGreSQL.
Nous avons donc testé l'utilisation d'une BDD avec Java.
L'accès à une BDD se fait via l'API JDBC (Java DataBase Connectivy) de Sun. Pour utiliser JDBC, il suffit d'importer le package *java.sql*.
Pour établir la connexion, il faut d'abord charger le pilote correspondant à la l'utilisation d'une BDD PostGreSQL puis faire une instance de la Classe Connection pour se connecter à la base.
Une fois la connexion établie, on peut exécuter les requêtes SQL nécessaires.

2. Le module réseau

a) Petite définition

Un réseau est le résultat de l'interconnexion de plusieurs machines entre elles. Les utilisateurs de ces machines ou des applications qui s'y exécutent, échangent par l'intermédiaire du réseau des informations.

b) Intérêt de ce module

Le but principal de notre projet est de réaliser une interaction, c'est-à-dire que chaque utilisateur peut en générant un évènement, interagir avec ou sur d'autres personnes.

Cette interaction a été mise en place dans ce module via la technologie RMI (Remote Method Invocation) développée et fournie par Sun.

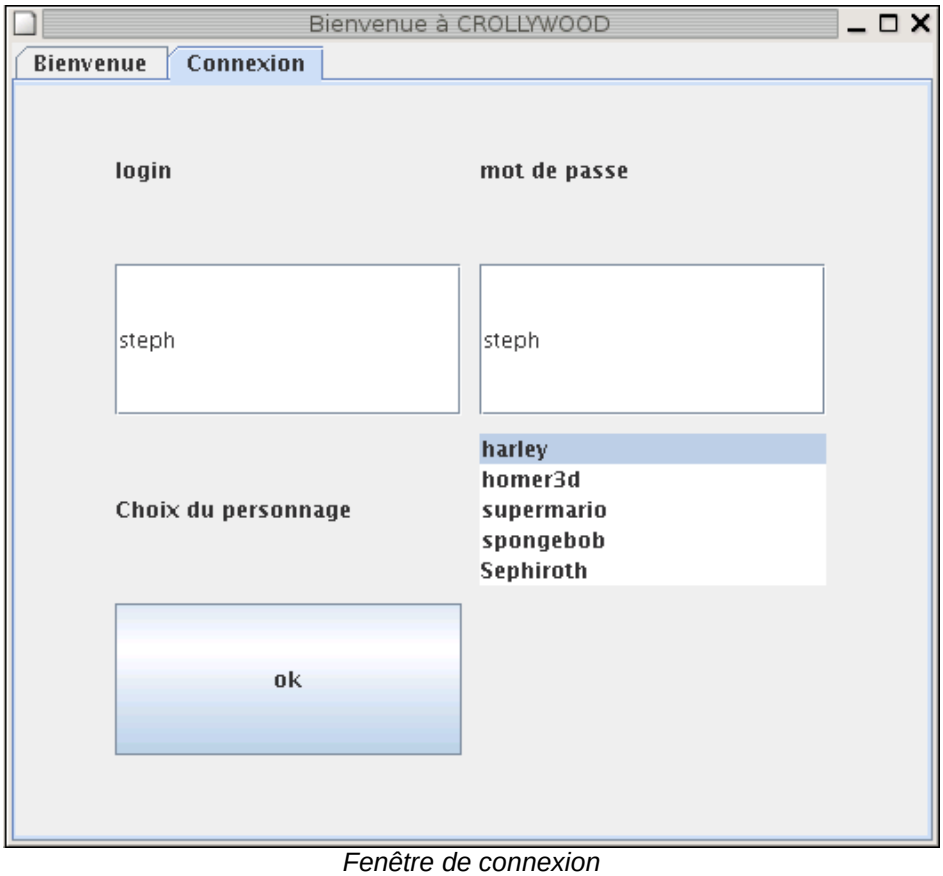
c) La BDD

Il nous fallait un moyen de garder, dans un endroit centralisé, les données des utilisateurs entre deux utilisations du programme. Ces données regroupent le login et mot de passe, la dernière position de l'avatar dans le monde 3D, ainsi que les informations sur l'espace personnel où il expose ses oeuvres.

Le moyen le plus efficace de garder ces informations de manière durable est la base de données. De plus, l'utilisation d'une base de données en Java est relativement simple, grâce aux classes de l'API.
Ayant vu le fonctionnement de PostgreSQL ce semestre, nous avons préféré l'utiliser plutôt que MySQL.

Des requêtes SQL sont utilisées pour vérifier les informations de connexion fournies par l'utilisateur, puis s'il peut se connecter, récupérer le reste des données, comme la position de l'avatar dans la scène.

Lors de l'exécution des requêtes SQL, on crée tout d'abord un « objet ordre » qu'on envoie au serveur. Si la requête est correcte, on récupère le résultat (curseur) sinon SQLException génère des erreurs. Le but du curseur est de récupérer un résultat de requête faisant plusieurs lignes.



Fenêtre de connexion

d) La connexion Clent/Serveur

Nous avons implémenté le reste du réseau via la technologie RMI (*Remote Method Invocation*).
Cette technologie permet d'appeler depuis un client des méthodes sur un objet du serveur comme s'il se situait sur le client.
Il faut pour cela que l'objet implémente une interface disponible aussi sur le client (par exemple ObjetDistant), et soit accessible par le registre RMI. Le client peut alors récupérer un "lien" vers l'objet sous forme d'instance de la classe Remote, qu'il caste en ObjetDistant et peut ensuite utiliser.

Notre application se présente sous la forme d'une application serveur et de plusieurs applications clients qui se connectent au serveur. En réalité, les deux parties jouent le rôle de client et de serveur :

Fonctionnement du serveur.

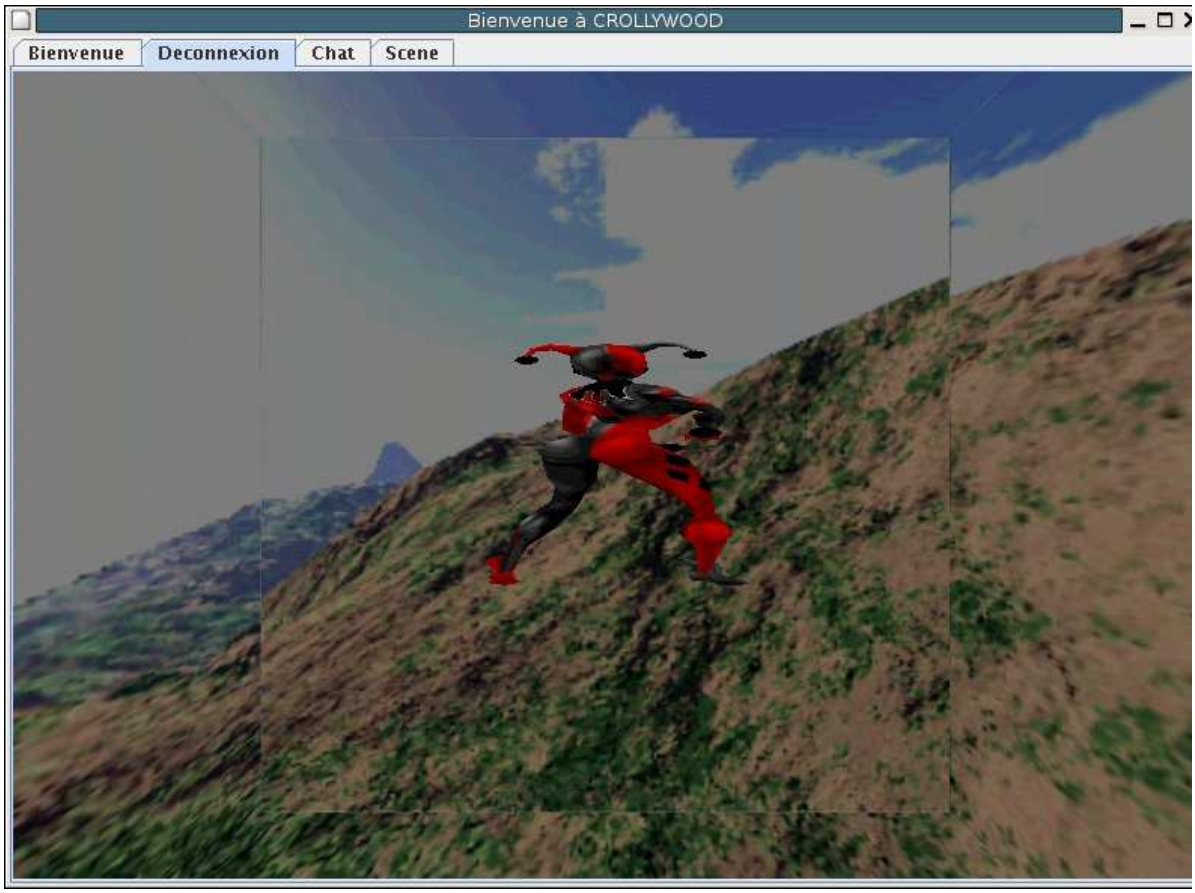
Le serveur dispose d'une instance de la classe Jeu, qui implémente l'interface JeuDistant connue par les clients. Cette interface permet au client de prévenir le serveur de sa connexion/déconnexion, d'envoyer la mise à jour de sa position, ... Elle gère également le changement de scène et l'animation des md3 par la mise à jour de ses positions chez les autres clients.

Fonctionnement du client.

Le client implémente l'interface ClientDistant, connue par le serveur. Lors de la connexion, le client envoie une référence vers lui-même au serveur, ce qui permettra à ce dernier de l'utiliser comme un objet distant. L'interface ClientDistant gère le changement de scène, l'apparition et la disparition d'un personnage md3 et la mise à jour de la scène.

Le chargement des personnages md3 est réalisé juste après la vérification du login de l'utilisateur. 5 personnages sont alors chargés. Cette opération prend un peu de temps mais grâce à cette méthode si un autre client se connecte, l'avatar de celui-ci apparaîtra immédiatement dans la scène. Cela évite de faire attendre le client au milieu d'une partie.

Pendant le jeu, les actions réalisées par l'utilisateur (déplacement, changement de monde, etc...) sont communiquées au serveur pour mise à jour des autres clients.



Fenêtre du client (vue de l'avatar du 2ème client connecté)

e) Le chat

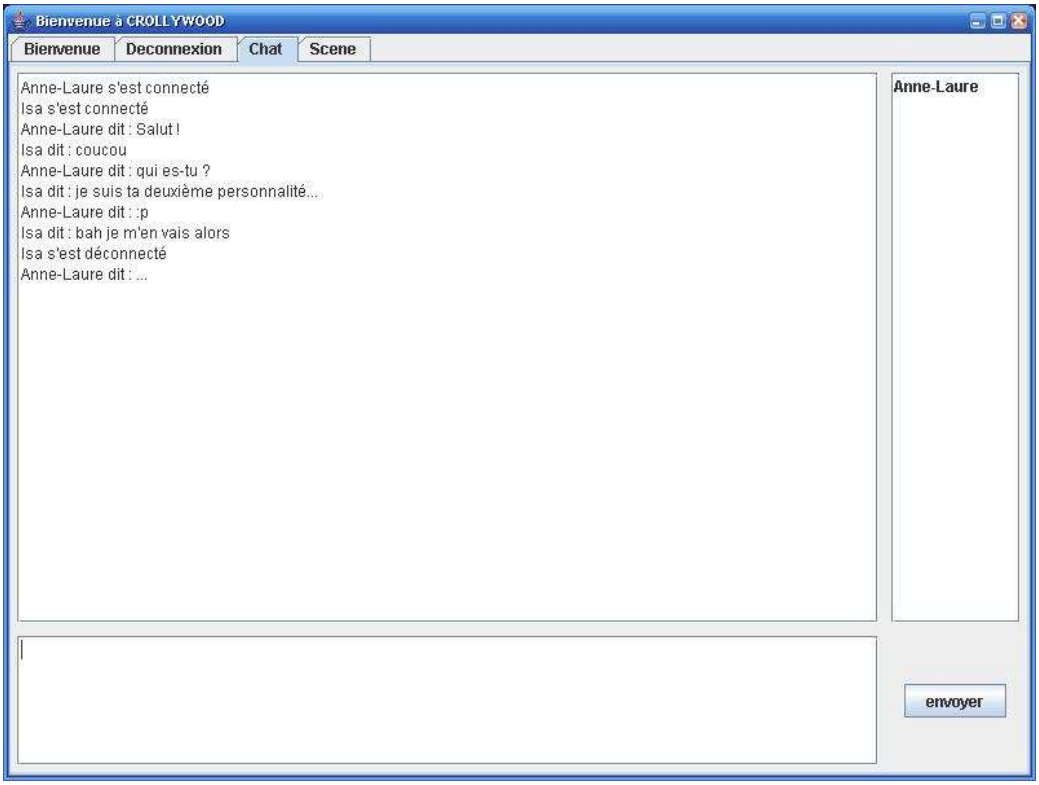
Une interface de chat permet aux visiteurs du monde de discuter entre eux. Ce module fonctionne grâce au RMI.

Le serveur abrite une instance de la classe Chat. Cette classe implémente l'interface ChatDistant, ce qui permet aux clients d'interagir avec elle.

Du côté serveur, le chat stocke la liste des noms des clients connectés, la liste des messages, un compteur du nombre de messages et un compteur du nombre d'évènements (connexions et déconnexions) depuis le lancement du serveur.

Le client peut récupérer ces informations grâce à des fonctions. Il a lui aussi un compteur de messages (initialisé sur celui du serveur à la connexion) qui lui permet de savoir quels messages il n'a pas encore reçu, et un compteur d'évènements, qui lui permet de savoir quand faire la mise à jour de la liste des connectés.

Des fonctions de l'interface ChatDistant permettent aussi aux clients d'envoyer des messages ou de prévenir de leur connexion ou déconnexion.



Interface du chat

Du côté client, le chat a une interface graphique simple, avec la liste des connectés, la fenêtre de messages, et un champ de texte pour écrire ses messages. Un simple appui de la touche Entrée permet d'envoyer le message.

Un thread très léger tourne en fond pour mettre le tout à jour, vérifiant périodiquement l'arrivée de nouvelles informations sur le serveur.

Des améliorations auraient été possibles, notamment des discussions privées ou un formatage du texte plus agréable, mais n'ont pas été implémentées par manque de temps.

3. Le module environnement 3D

a) La scène

Modélisation

Les personnages de notre monde doivent évoluer dans un monde 3D. Le monde sera réalisé sous 3DS et ensuite intégré au reste du projet. La scène sera bâtie sur un plan plat. Nous avons décidé de nommer notre projet Crollywood en référence à Charles Cros et au cinéma. L'élément majeur de la scène sera donc un cinéma nommé crollywood ou sera entreposé toutes les œuvres audiovisuelles des utilisateurs. Un deuxième élément, un phonographe, contiendra les œuvres musicales.
Ensuite, chaque utilisateur qui se connectera et créera son univers se verra attribuer une maison qui contiendra toutes ses productions.

Intégration des objets modélisés

Dans le but d'intégrer les éléments du décor dans notre scène, nous avons exporté tous les objets modélisés en .obj, format supporté par Java. Ensuite, nous avons pu charger ces objets seulement leurs textures n'étaient pas affichées. Nous avons aussi essayé de les exporter et de les charger au format .3ds au moyen du fichier .jar de Microcrowd mais cela n'a pas réussi à régler le problème.

Changement d'environnement

Lorsqu'une collision est détectée avec un objet de la scène correspondant à une porte, on récupère les informations de cette porte pour savoir quel environnement se cache derrière, et donc quel environnement doit être chargé.

Pour modifier l'environnement chez le client même, on détache tout d'abord du BranchGroup principal la scène principale, puis on charge le nouvel environnement et on l'attache au BranchGroup principal.
Chez les autres clients, on détache le MD3 du client qui vient de quitter la scène principale afin de la faire disparaître.

Ensuite, dans le nouvel environnement, si on détecte une collision avec la porte, on retourne à la scène principale. Cela se traduit chez le client même par le détachement de l'environnement (et la mise à null de sa référence afin de libérer l'espace mémoire), puis l'attachement de la scène principale que l'on avait conservée (ceci afin d'éviter de charger plusieurs fois une scène assez importante et vers laquelle on revient sans cesse).
Chez les autres clients, on fait réapparaître le MD3 du joueur qui revient en le rattachant au BranchGroup principal de la scène.

Gestion picking, collisions

- Création du monde et Déplacement caméra

Nous souhaitions implémenter une vue à la première personne lors de la navigation au sein de notre monde virtuel. Il fallait donc que caméra et personnage soient liés. Au départ, les tests de liaison ont été implémentés entre la caméra d'une part et un objet simple de type sphère d'autre part. Le début de nos tests s'effectuait avec une vue à la troisième personne afin de mieux visualiser les déplacements de l'objet. Nous avons ensuite intégré le personnage MD3 et le FPS.

- Principe

Nous commençons par créer une scène ajoutée à notre univers. Nous créons un TransformGroup lié à la caméra , puis un BranchGroup pour le personnage MD3 que nous lions au premier TransformGroup.

Les événements clavier (CustomKeyNavigatorBehavior) permettent de déplacer le personnage qui sera donc suivi par la caméra. Une translation du personnage permet de passer d'une vue à la troisième personne à une vue à la première.

- Le picking

Le picking est déjà implémenté en Java grâce aux classes PickCanvas et PickResult. Il a donc été relativement facile de l'intégrer. L'utilisation du picking permet en outre de savoir quel est l'objet cliqué. Cela peut être intéressant pour connaître le lieu choisi par l'utilisateur lors d'un clic souris, ou le personnage sur lequel a été effectué un clic en vue de savoir qui est dans la scène.

Le problème qui s'est posé était que le picking donnait correctement le nom des objets de type .obj mais pas celui des personnages MD3 dont nous connaissions seulement la position. Pour corriger ce problème, lors de l'intégration avec la partie réseau, nous avons utilisé la méthode setUserData qui prend en paramètre le nom du client.

- Les collisions

Pour réaliser nos tests de collision, nous avons utilisé un programme existant. Il s'agissait de deux sphères changeant de couleur lorsqu'un cube les heurtait. Par la suite, nous avons remplacé le cube par notre personnage MD3. Les collisions sont détectées grâce aux boîtes englobantes des objets.

L'information de collision est envoyée si un objet entre dans la zone de collision (WakeupOnCollisionEntry). Cela affectait true à la variable *DetecteurCollision.coll*. Dans ce cas, nous empêchons le personnage d'avancer.

Au début nous utilisions une méthode qui consistait à distinguer deux cas lorsqu'une touche était appuyée, lors d'un évènement clavier. Soit le joueur n'était pas en collision et alors l'état des touches changeait normalement (appui sur une touche => key_state = true , relachement de la touche => key_state = false). Soit le joueur était en collision et alors l'appui sur les touches ne provoquait rien, l'état des touches ne pouvait plus changer, excepté la touche opposée au dernier déplacement.

Lors de nos tests, nous nous sommes heurtés à un problème. Le personnage, dont la vitesse de déplacement était trop rapide du fait de son accélération, traversait les objets.

Pour résoudre ce problème, nous avons alors opté pour une méthode radicalement différente : l'état de la touche est changé normalement même s'il y a collision. En revanche, dans la méthode integrateTransformChanges, appelée une fois par frame, nous détectons la collision, et s'il y a collision, nous reculons sur une position précédente, celle lors de l'appel précédent à integrateTransformChanges, au moment où il n'y avait pas encore de collision.

Le fait est que cette méthode n'est pas parfaite, parce qu'il se peut que la position précédente sur laquelle on retourne soit en fait une position où il y avait en fait collision. En effet, l'état de la collision, calculé par DetecteurCollision n'est pas mis à jour suffisamment souvent, pas aussi vite que integrateTransformChanges. Donc, il peut arriver que *DetecteurCollision.coll* nous informe qu'il n'y a pas collision alors qu'il y a collision en fait.

Cependant la plupart du temps ça marche, il suffit de régler une certaine distance de mise à jour entre la position courante et la position précédente, de manière à ce que cette dernière ne soit pas mise à jour trop souvent. Cette distance ne doit pas être trop petite, auquel cas cela risque de générer des erreurs de collisions plus souvent, mais pas trop grande non plus auquel cas quand on collisionnera le décor, on reculera beaucoup trop.

Dans le cas critique où on serait quand même en collision en revenant en arrière, nous avons opté pour la solution de bouger aléatoirement jusqu'à ce qu'on ne soit plus en collision.

Le son d'ambiance :

Grâce aux tests réalisés en début de projet, nous avons pu implémenter rapidement le module son du programme. Nos tests permettaient soit de lancer un son .au dans une applet, soit de jouer un son .wav dans une application. Notre programme n'étant pas basé sur un système d'applets, nous avons intégré la deuxième solution et l'avons optimisée. En effet, nous avons intégré les threads, afin que l'application continue à tourner lorsqu'un son joue, ce qui n'est pas le cas autrement. De plus, nous avons créé deux classes ayant des objectifs différents. La première permet de charger un fichier son passé en paramètre et de le lire une fois ou en boucle selon le booléen passé en second paramètre. La seconde parse le répertoire de nom passé en premier paramètre, place tous les son .wav dans une liste, et les lit successivement, jusqu'à repartir ou non du début, selon le second paramètre. Cela permet donc, si le projet est repris, de ne pas avoir à rentrer en dur le nom des fichiers son à lire, mais de seulement placer ceux que l'on souhaite dans le dossier qui sera parsé.

b) Les personnages md3

Chargement

Nous avons décidé que l'utilisateur serait obligé de choisir son avatar MD3 parmi une liste prédéfinie de MD3. Ainsi, lorsque l'utilisateur lancera l'application, celle-ci chargera dans des objets *MD3Loader* l'ensemble des MD3 de la liste.

Lors de la connexion au serveur, l'application enverra aux autres clients des données concernant son utilisateur dont entre autre le nom de l'avatar choisi. Les autres clients recevant ces infos, étant déjà connectés et se promenant déjà dans le monde virtuel, charger un MD3

chaque fois qu'un autre client se connecte serait trop lourd et ralentirait considérablement l'application. C'est pourquoi tous les MD3 sont chargés au lancement de l'application, permettant ainsi de n'avoir qu'à créer une nouvelle instance du MD3 choisi chaque fois qu'un nouvel utilisateur se connecte : ce qui prend beaucoup moins de ressource que le chargement.

Chaque application crée donc un nombre limité de *MD3Loader* et autant de *MD3ModelInstance* que de clients connectés.

Animation

Le chargeur MD3 choisi prévoit l'animation possible du personnage 3D affiché. Pour cela, il suffit donc d'utiliser sur un objet *MD3ModelInstance* la méthode *setAnimation* qui permet d'initialiser l'animation en spécifiant quel type d'animation on veut (marche, course, …) et sur quelle partie du MD3. Ensuite, pour donner l'illusion que le personnage s'anime, il suffit de passer à l'image clé suivante grâce à la méthode *nextFrame.* Cette méthode est appelée par exemple tant que l'on reste appuyé sur une flèche du clavier pour simuler le mouvement de marche.

Déplacement

Un joueur se déplace en First Person Shooter. Par conséquent, ce qui donne la sensation de se déplacer dans la scène 3D est en fait le déplacement de la caméra. Le déplacement de la caméra se fait grâce à des modifications successives de sa matrice de transformation. Nous récupérons donc cette matrice chaque fois qu'elle est modifiée et l'envoyons aux autres clients afin que ceux-ci l'applique au MD3 du joueur qui s'est déplacé : le MD3 bougera de la même façon que la caméra du joueur.

c) Gestion jour/nuit

L'environnement est composé d'un cube texturé dont les faces sont tournées vers l'intérieur (on appelle ça une skybox). Pour optimiser le rendu, les coordonnées de textures sont telles qu'il n'y a qu'une seule texture pour l'ensemble de la skybox.

Le système de gestion du jour et de la nuit est basé sur l'utilisation d'un objet *ColorInterpolator* (qui est un objet de l'api Java 3D qui hérite de *javax.media.j3d.Interpolator*). Il permet de faire une modification de la luminosité de la skybox en fonction du temps. Il suffit de régler ce temps sur 24 heures pour que le système jour/nuit fonctionne comme dans la vraie vie.

Cependant, pour que le jour et la nuit soient en corrélation pour tous les clients (en effet, à priori le jour et la nuit sont décalés pour chaque client en fonction de l'heure à laquelle il se connecte), il est nécessaire au démarrage de vérifier l'heure courante et d'ajuster la luminosité, au moyen de la classe *Calendar,* qui nous fournira l'heure courante. Remarque : cette dernière fonctionnalité n'a pas à été implémentée à l'heure où ce rapport est écrit.

4. Le module éditeur

a) Le chargement XML

L'intérieur des maisons

L'application devra offrir la possibilité aux utilisateurs d'arpenter des intérieurs en trois dimensions, présentant différents objets, éventuellement interactifs.
L'emploi de fichiers XML est une solution simple pour décrire cet environnement, le stocker, mais aussi le modifier (dans un simple éditeur de texte ou dans celui que proposerait le logiciel). Notre format de document devra permettre de décrire les éléments suivants :

- Architecture de la pièce
- Coordonnées et orientation des objets
- Dimensions et textures des objets
- Interactions utilisateur possibles avec les objets

Resituant ce travail par rapport au précédent projet « Chat 3D », nous avons réfléchi à organiser la scène en trois types d'objets : des objets primitifs, des objets complexes et des objets composés (composés de plusieurs objets primitifs, complexes ou composés). Un système de SuperFactory et Factory d'objets permettant de gérer tout type d'objet de façon générique.

Nous utiliserons le modèle DOM plutôt que SAX, d'abord, pour sa simplicité, mais aussi car SAX ne permet pas de générer des documents, ce dont nous aurons besoin pour réaliser l'éditeur.

L'architecture par défaut :

Par défaut, 4 murs inamovibles ferment l'espace éditable pour un intérieur. Ces murs, gèrent les collisions comme les autres, seule une « porte » prédéfinie (figurée par un porche, un portail ou autre) permet de retourner vers l'environnement extérieur.
La texture de ces éléments pourra être modifiée, tandis que leurs dimensions, positions et interactions seront fixées par défauts.

Les éléments de la « DTD » :

<environnement> : le conteneur principal dans lequel on trouvera l'ensemble des informations du fichier. Indispensable pour respecter la structure XML.

<parametres> : élément conteneur optionnel (en prévision d'extensions) pour régler plus finement les éventuels paramètres globaux de la scène

<architecture> : conteneur pour les éléments d'architecture par défaut (murs inamovibles et porte)

<objets> : conteneur pour l'ensemble des « objets » de la scène (œuvres, meubles, zones d'interactions, modèles 3D)

<point3d> ou **<position>** ou **<scale> :** un point 3D (soit la position d'un objet) ou un vecteur d'homothétie, définis par leurs trois coordonnées (x, y, z)

```
<point3d>
        <x>1</x>
        <y>1</y>
        <z>1</z>
</point3d>
```

Mur : permet de définir des pièces, gère les collisions avec les personnages. Défini par deux points 3D (haut/gauche, bas/droite), qui déterminent sa position ET son orientation. Une texture est indiquée pour les deux faces.

- **<mur> :** hauteur fixe (plafond), profondeur fixe
- **<fenetre> :** idem, avec un « trou » pour la fenêtre
- **<muret> :** idem, mais à hauteur variable entre un minimum et un maximum

<stairsUp> et **<stairsDown> :** des escaliers, permettant de changer d'étage (versions montantes et descendantes)

Les personnages et les sons

Rappelons que l'intérieur d'une maison est chargé à partir d'un fichier XML distant se trouvant sur un serveur HTTP, serveur qui contient aussi des données, md3, images, et sons.

Exemple de fichier XML :

```
<environnement>
      <parameters>
      </parameters>
      <images>
            <raoul.jpg>
                  <position>
                        <x>1</x>
                        <y>1</y>
                        <z>1</z>
                  </position>
                  <scale>
                        <x>10</x>
                        <y>2</y>
                        <z>1</z>
                  </scale>
            </raoul.jpg>
      </images>
      <objects>
```

```
            <homer3d>
                  <position>
                        <x>1</x>
                        <y>1</y>
                        <z>1</z>
                  </position>
                  <scale>
                        <x>10</x>
                        <y>2</y>
                        <z>1</z>
                  </scale>
            </homer3d>
      </objects>
      <sounds>
            <harmonica.wav>
                  <position>
                        <x>1</x>
                        <y>1</y>
                        <z>1</z>
                  </position>
                  <scale>
                        <x>10</x>
                        <y>2</y>
                        <z>1</z>
                  </scale>
            </harmonica.wav>
      </sounds>
</environnement>
```

La balise parameters définit des options éventuelles. Il n'y a pas d'options dans la version actuelle, ce qui implique que ce qui est entre ces balises n'est pas pris en compte, mais pourrait éventuellement servir à l'amélioration future des fonctionnalités du programme.

La hiérarchie correspondante sur le serveur HTTP doit être alors :

- Racine [dossier]
  - contents.xml
  - images [dossier]
    - raoul.jpg
  - objects [dossier]
    - homer3d.zip [dossier du md3 homer3d zippé]
  - sounds [dossier]
    - harmonica.wav

Le chargement du XML se fait au moyen de la classe *Loading*, qui lit en local ou à partir du serveur HTTP le fichier XML. Dans le cas où il s'agit du serveur HTTP, cette classe télécharge les fichiers de données dans un dossier temporaire, décompresse les fichiers zip dans le cas des MD3, et charge le tout en mémoire vive.

Dans le cas des "images", celles-ci sont chargées sous forme de cube contenant sur chaque face l'image en question. Dans le cas des "objects", le md3 correspondant est chargé dans la scène (mais il est non animé).

Pour les images et les objets 3d, il est possible dans le XML de définir la position et l'échelle en XYZ de l'objet chargé. Changer l'échelle permet par exemple de donner au cube 3D texturé une allure de tableau.

Pour ce qui est des sons, ils ne sont pas encore chargés à l'heure où le rapport est écrit. (gageons qu'ils le seront au moment de la soutenance, sous forme de cube en 3D cliquable à la souris, utilisant le picking).

5. Intégration des modules

Une fois le réseau opérationnel, nous avons commencé à intégrer les différents modules.
Nous avons commencé par intégrer le système de gestion Jour/Nuit puis les personnages md3. A ce niveau, nous avons rencontrées quelques difficultés pour gérer l'apparition et la disparition des personnages chez les autres clients. Une fois ce problème résolu, nous avons intégré le picking et la gestion des collisions. Nous avons en dernier lieu inclus le son d'ambiance.

IV. BILAN

1. Les problèmes rencontrés

Nous nous sommes répartis les différents modules du projet, ce qui nous a facilité le travail dans un premier temps car chacun des modules pouvait être programmé indépendamment des autres. Cependant, rassembler toutes ces parties, une fois les modules réalisés, s'est avéré être une tâche difficile.
Concernant la partie environnement 3D, le regroupement a été d'autant plus difficile que les différentes personnes ayant travaillé dessus ont chacune créé une scène principale avec des éléments différents.

2. Les résultats obtenus

Nous sommes assez satisfaits de notre travail malgré le fait que notre application ne soit pas encore totalement aboutie. En effet, faute de temps nous n'avons pas pu créer l'application que nous nous imaginions au départ. Cependant, pratiquement tous les modules ont été réalisés et fonctionnent bien. Il serait donc possible de reprendre ou de continuer ce projet plus tard.

3. Les améliorations possibles

Cette application n'est pas totalement complète, il reste encore à ajouter des fonctionnalités.
Au lieu d'un Chat, nous pourrions aussi permettre à l'utilisateur de sélectionner dans la scène un autre utilisateur et de dialoguer uniquement avec celui-ci.
Nous pourrions également ajouter d'autres objets dans la scène : laboratoire pour les travaux de recherches, des arcades pour les programmes de jeux…
Nous regrettons également que notre univers n'ait pas réellement un thème général, une ambiance qui se manifesterait dans les éléments du décor, les personnages, l'interface de l'application ou dans l'ambiance sonore.

BIBLIOGRAPHIE

Livres :

ROUSSEL G. - DURIS E. - BEDON N. - FORAX R., *Java et Internet*, 2002, Vuibert, 557p
FLANAGAN D - GACHET A, *Java in a Nutshell, Manuel de référence*, 2002, O'Reilly, 1128p

Sites Internet :

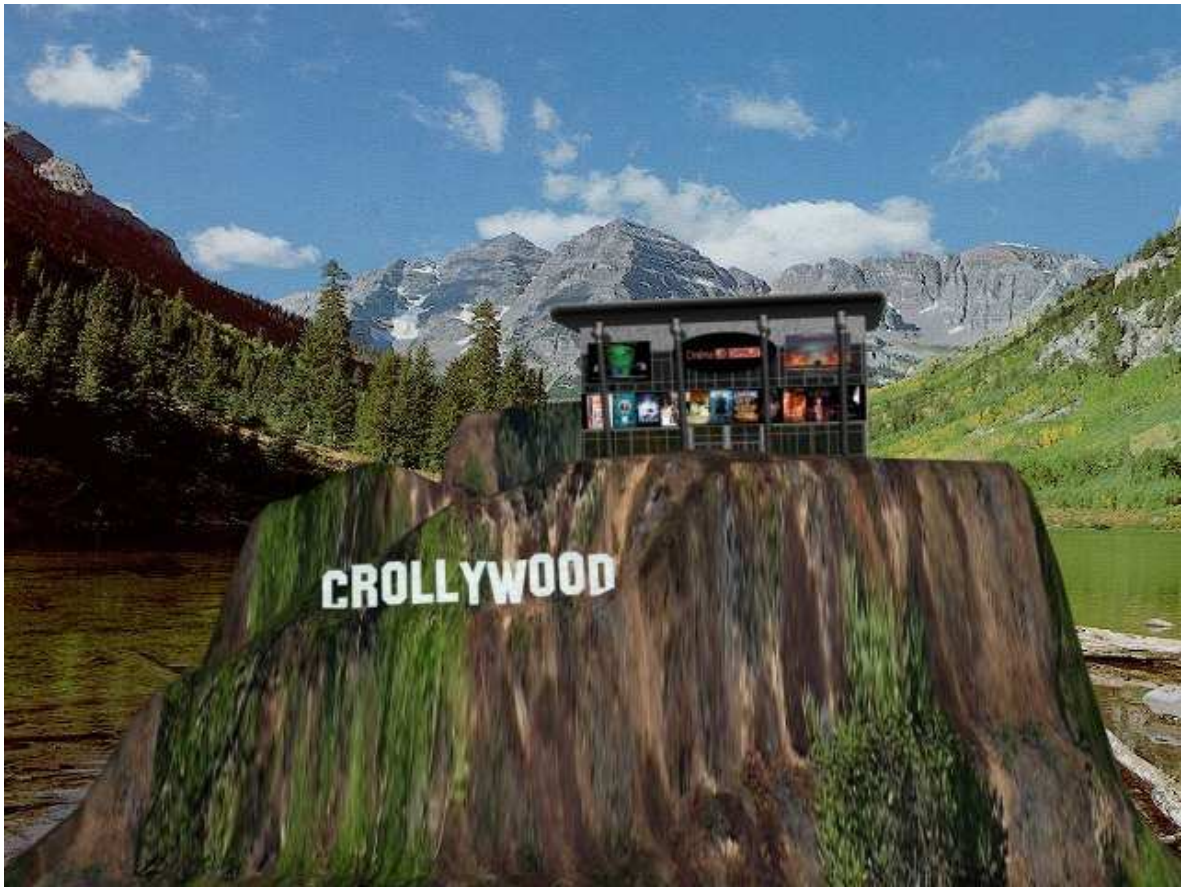
http://perso.orange.fr/jm.doudoux/java/tutorial/index.htm
http://pages-perso.esil.univ-mrs.fr/~tourai/Java/node45.html
http://joggplayer.webarts.bc.ca/html/docs/api/kiwi/ui/AudioClip.html
http://java.sun.com/j2se/1.4.2/docs/api/java/applet/AudioClip.html
http://perso.wanadoo.fr/jm.doudoux/java/tutorial/chap017.htm
http://www.eteks.com/coursjava/applet10.html
http://b.kostrzewa.free.fr/java/td-images/avecapplet.html
http://www.self-access.com/java/jvSon1.htm
http://www.javafr.com/codes/JOUER-SON-WAV-JAVA_29515.aspx

ANNEXES

1. Modélisation du cinéma
2. Schéma de conception

1. Modélisation du cinéma

Le cinéma



Le cinéma intégré dans la scène