Le petit chateron rouge

Rapport de projet

OpenGL, Programmation, réseau

Présenté et soutenu le 18 janvier 2006 à Serris

Benhamouche Lynda
Billard Julie
Desrousseaux Sandra
Dony clément
Fayolle Séverine

Enseignant : Mr Chaudeyrac Pascal et Mr De Sorbier François

Promotion 2007 – Diplôme IMAC

Introduction

Sur la base du chargement d'un monde et de personnages choisis par des utilisateurs sur des machines différentes et connectés à un même serveur, le présent projet vise à la réalisation d'un chat. Nous avions le choix avec la réalisation d'un jeu de laser mais avons préféré le chat, car ...

Il met en oeuvre nos acquis des précédents mini-projets (picking et chargement d'un md3) et mobilise de nouvelles compétences : réalisation d'un parseur de fichiers XML, mise en place d'une Factory, utilisation d'interfaces Qt, et importante partie réseau.

Dans ce rapport, nous détaillerons la façon dont nous avons organisé notre travail face à ce projet, avant de nous attacher à la conception du programme dans ses différentes entités, et de décrire les difficultés auxquelles nous nous sommes heurtés et les différentes améliorations que nous pourrions apporter à notre projet.

Il était une fois une petite fille de village, la plus jolie qu'on eût su voir : sa mère en était folle, et sa grand-mère plus folle encore. Cette bonne femme lui fit faire un petit chaperon rouge qui lui seyait si bien, et partout on l'appelait le petit Chaperon rouge.

Un jour, sa mère ayant fait des galettes, lui dit : "Va voir comment se porte ta mère-grand : car on m'a dit qu'elle était malade ; porte-lui une galette et ce petit pot de beurre." Le petit Chaperon rouge partit aussitôt pour aller chez sa mère-grand, qui demeurait dans un autre village.

En passant dans un bois, elle rencontra compère le Loup qui eut bientôt envie de la manger [...]. Le méchant Loup se jeta sur le petit Chaperon rouge et la mangea.

Suite à cet événement tragique, deux notables, le comte François De Sorbier de Pougnadoresse et son confrère Pascal Chaudeyrac trouvèrent un moyen pour que de jeunes filles et leur mère-grand puissent se parler à distance, et demandèrent à leurs meilleurs apprentis de créer un chat, ce projet s'appelant « le Petit Chateron Rouge » en mémoire à cette défunte petite fille, la plus jolie qu'on eût su voir ...

1. Organisation

a. Planning

Date(s)	Description
Du 5 au 18 Décembre	**Phase d'analyse**
18 Décembre	**Répartitions des taches**
Du 19 Décembre au 4 Janvier	**Conception et développement en individuel**
Du 5 au 15 Janvier	**Regroupement des premières versions des différentes parties et améliorations de ces parties Rédaction du rapport**
16 Janvier	**Tests du programme**

b. Répartition des tâches

Les taches ont d'abord été distribuées ainsi :

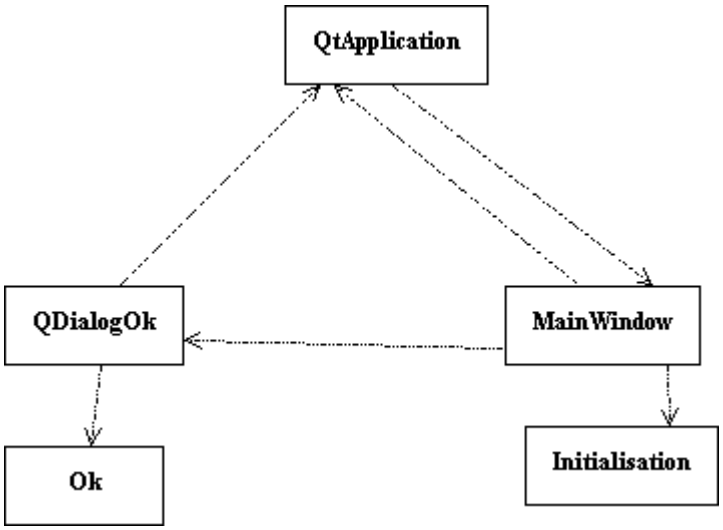
- Gestion des personnages md3 et picking : Clément
- Chargement et création des décors : Julie
- Réseau : Lynda et Séverine Interface
- Qt : Sandra

Cependant, au moment de la phase de conception du réseau, celui-ci sembla moins compliqué à mettre en place et par conséquent cette partie du projet ne nécessitait plus deux personnes. De plus, d'autres personnes ayant plus de difficultés à concevoir leur partie, Séverine s'est donc occupée seule du réseau permettant ainsi à Lynda d'aider les autres. Elle a ainsi principalement participé à l'élaboration du parseur XML avec Julie.

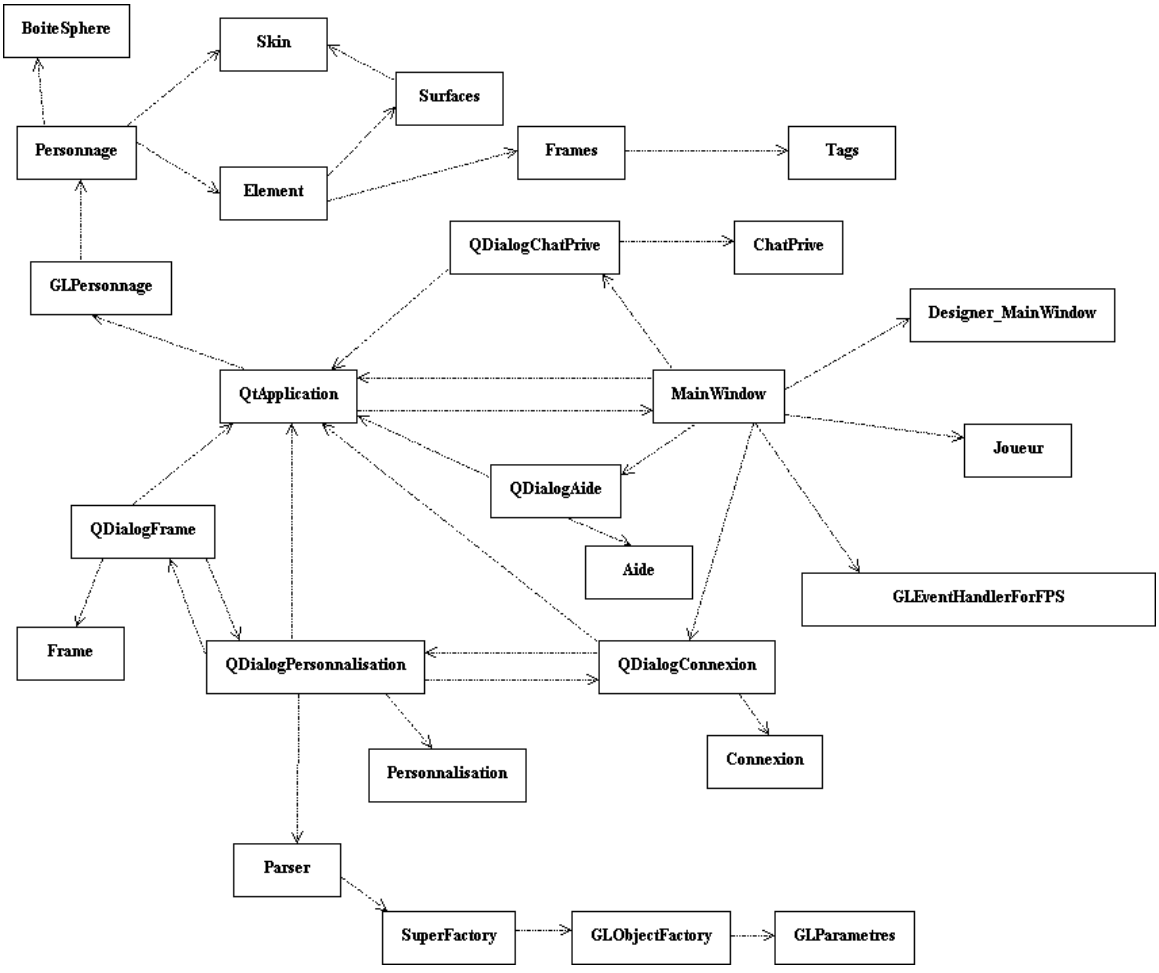
Lors de la phase d'intégration des différentes parties du programme, la répartition des taches s'est faite en fonction de ce qui nous restait à faire. Le chargement des décors et la gestion des personnages md3 étant les deux parties qui ont posé le plus de problème, Sandra et Séverine ont donc regroupé les interfaces Qt et le réseau, nous permettant ainsi d'obtenir une première version de chat basique avec discussion publique et discussion par groupe. Elles se sont ensuite occupées du chargement du personnage md3 en fonction de l'avatar choisi. Le parseur XML une fois fini a été intégré à l'interface Qt tandis que Lynda a aidé Clément sur la gestion des personnages md3 et le picking. Cette partie a ensuite été intégrée dans l'interface Qt par Clément et Sandra puis liée au réseau par Clément et Séverine.

2. Mise en œuvre

Liens entre les classes coté serveur



Liens entre les classes coté client



a. Le picking et le MD3

Le picking

Le picking consiste à appliquer une boîte englobante à des objets et de déterminer l'intersection entre cette boite et un rayon, et ainsi de récupérer l'identifiant de l'objet et pouvoir agir dessus. Nous avons le choix entre l'utilisation de boites sphériques ou de boites cubiques en fonction de la forme de l'objet.
Le déplacement dans le monde se faisant en FPS, le rayon que l'on jette doit être recalculer en fonction de la position de la caméra et du point sur lequel on a cliqué. Si le rayon intersecte plusieurs boites, le programme récupère uniquement l'objet le proche de la camera.

Le MD3

Le fichier MD3 est un format d'image 3D utilisé dans le jeu Quake3. Afin d'intégrer ce format, le programme suit trois phases :

- Lecture et stockage des informations du fichier MD3
- Héritage entre Personnage MD3 et la classe GLObject
- Affichage du personnage MD3.

La réunion du picking et du MD3

Notre programme n'utilise les boîtes englobantes que sur les éléments des personnages 3D (les fichiers MD3), c'est à dire les jambes, le torse et la tête. Nous avons décidé d'utiliser des boites sphériques car le chat ne nécessite pas une aussi grande précision que celle du FPS, et que celles-ci sont plus faciles à mettre en oeuvre.

Problèmes et solutions

Problème : Les fichiers MD3 possèdent des informations pour appliquer des boîtes englobantes mais ces données sont fausses. Par conséquent, le picking est également faux.
Solution : Nous avons donc dû recalculer les informations pour obtenir des valeurs plus réalistes. Cela consiste tout d'abord à obtenir le centre géométrique de l'élément du corps en calculant le milieu des coordonnées extrema de l'élément, puis à récupérer la distance la plus élevée entre chacun des points constituant l'élément et le centre géométrique pour connaître le rayon de la boite englobante.

Problème : Les boîtes englobantes sont centrées par défaut aux coordonnées (0.0, 0.0, 0.0). Elles n'englobent donc pas leur élément de personnages respectif.
Solution : Nous avons donc dû faire subir au centre de la boite toutes les transformations que leur élément respectif avaient également subies.

Problème : Le déplacement des personnages utilise des méthodes GLUT telles que *rotate* et *translate*. Or les coordonnées des personnages et des boîtes englobantes ne subissent aucune modification, le picking est donc faux.
Solution : Les coordonnées des personnages ne subissent pas les transformations car elles n'interviennent pas dans le picking. Mais les coordonnées des boîtes englobantes doivent subir les transformations, par contre, elles ne subissent que les translations, ainsi que les rotations car les boîtes sont désaxées.

b. La construction des objets graphiques

La conception des classes

Pour construire les objets graphiques de la scène il nous a été demandé d'utiliser le principe des factories et de singletons. On aura donc une SuperFactory de type singleton dont toutes les factories des objets hériteront.

SuperFactory.hpp	Singleton
static SuperFactory& getInstance () SuperFactory () GLObjectFactory& getGLObjectFactory(std::string name)	
std::map <std::string,GLObjectFactory*> mesFactories static SuperFactory* maSuperFactory	

Pour créer une classe singleton, il faut réaliser les étapes suivantes :

On place le constructeur en privé pour qu'une seule instance de la classe puisse être créée. L'unique instance de la classe est dans la classe elle-même en privé. Pour y accéder on utilise la méthode getInstance () qui permet de récupérer l'instance de la classe. On n'oublie pas de faire un test sur maSuperFactory pour savoir si elle a déjà était construite ou non. Sinon on la construit.

La map mesFactories stocke les factory des objets à construire en fonction d'un string qui correspond au nom de l'objet (« sphère, cube, plan… »). La factory de l'objet est retournée grâce à la méthode getGLObjectFactory (std ::string name).

Le constructeur de la classe contient le parcours et la lecture des .so. On parse le répertoire ou sont situées les librairies dynamiques et on récupère le retour (un GLSphereFactory par exemple…) de la fonction f du plugin. On stocke ce retour ainsi que le nom récupéré par la méthode getname () de la factory dans la map mesFactories.

GLObjectFactory.hpp	Interfaces
virtual rt::gl::GLObject& createGLObject(GLParametres &p)=0 virtual std::string getName()=0;	

La classe GLObjectFactory est une interface, c'est-à-dire que toutes ses méthodes sont des virtual pures. Elles doivent être obligatoirement redéfinies lors d'un héritage pour être utilisées. Tous les objets auront une classe GL<nomdelobjet>Factory qui héritera de cette classe.

GLParametres.hpp
void addParametre(std::string s, void* arg) void *getParametre(std::string s)
std::map <std::string,void*> mapParametre;

La classe GLParametres est la classe qui permet de construire des objets graphiques avec des paramètres (position, taille, texture…). On ajoute un paramètre dans la map grâce à addParametre (). On les récupère grâce à getParametre ().
Les paramètres des objets sont ajoutés dans la factory de chaque objet.
Pour construire le paramètre de l'objet on associe le nom du paramètre lu dans le fichier xml à une chaîne de caractère, lu elle aussi dans le fichier xml, castée en void*.
On retourne ensuite le nouvel objet avec tous les arguments lus et récupérés.

GLSphereFactory.hpp
rt::gl::GLObject& createGLObject(GLParametres &p) std::string getName()

Permet la construction des objets avec les paramètres lu dans le xml. La méthode retourne un new GLSphere(….).

GLSphere.hpp
GLSphere (double x, double y, double z, double rayon, int nbDiv, std::string texturefichier) virtual void draw ();

Construction graphique de l'objet grâce aux primitives opengl. Placage des textures, normalisation des normales et construction de la CallListe.

Plugin_Sphere.cpp
return new GLSphereFactory ();

Fichier permettant de construire la librairie dynamique en .so. Ces fichiers .so seront ensuite envoyés aux clients via le serveur.

Problèmes et solutions

Problème : Lorsque l'on charge de nombreux objets texturés dans une même scène, on s'aperçoit que le rafraîchissement de la scène, lorsque l'on clique dans la fenêtre, est très lent voir même inexistant.

Solution : Pour régler ce problème il faut mettre en place des callLists et charger des textures de petites tailles (128x128 pixels).

Le parseur XML

a) Choix de l'API

Le parseur XML a pour but de parcourir le fichier XML et d'extraire toutes les informations relatives aux objets que contiendra, à l'affichage, la scène du chat. Chacun de ces objets est une primitive (cube, tore, pyramide) construite par les fichiers .so pour lesquels on définit des propriétés d'affichage (position, taille, transformations, …).

Les primitives des objets qui pourraient être ajoutées à la scène sont nommées entre les balises <primitives></primitives>. Puis les objets qui seront par la suite affichés sur la scène sont détaillées entre les balises <complexes></complexes>.

Voici un exemple d'un fichier XML où l'on définit les objets à ajouter à la scène :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<scene>
	<primitives>
		<primitive>sphere</primitive>
		<primitive>cylindre</primitive>
	</primitives>
	<complexes>
		<!-- POMMIER1-->
		<complexe>
			<primitive type="sphere">
				<forme>
					<position>4.0 3.5 -6.0</position>
					<rayon>1.3</rayon>
					<nbDiv>50</nbDiv>
				</forme>
				<texture>
					<fichier>Texture/feuille1.jpg</fichier>
				</texture>
			</primitive>
			<primitive type="cylindre">
				<forme>
					<position>4.0 2.0 -6.0</position>
					<hauteur>4.0</hauteur>
					<rayon>0.3</rayon>
					<nbDiv>50</nbDiv>
				</forme>
				<transformation>
					<rotation>0.0 0.0 0.0 0.0</rotation>
				</transformation>
				<texture>
					<fichier>Texture/bois3.jpg</fichier>
				</texture>
			</primitive>
		</complexe>
	</complexes>
</scene>
```

Pour parcourir les fichiers XML, on a donc utilisé un analyseur syntaxique pour lequel nous avions deux choix :

- DOM (Document Object Model) basé sur un mode hiérarchique
- SAX (Simple API for XML) basé sur un mode événementiel.

Pour notre part, nous avons choisi SAX car contrairement à DOM, ce parseur permet de parcourir le fichier XML selon les événements que nous lui indiquons et de traiter seulement ce qui est nécessaire. Ainsi, le résultat que nous retourne le parcours du fichier XML correspond exactement à ce que nous voulions récupérer comme informations.

Ce n'est pas le cas de DOM qui, lui, récupère tout le contenu du fichier XML avant de pouvoir faire le traitement sur celui-ci. Ceci peut être un problème s'il l'on traite des fichiers trop gros, ce qui entraînerait alors une lenteur dans le processus de traitement et une possible saturation de la mémoire tampon dans laquelle l'intégralité du fichier est mise.

La construction du parseur

L'API SAX définit quatre interfaces pour le traitement d'un fichier XML : DocumentHandler, ErrorHandler, DTDHandler, EntityResolver; dans notre cas, nous utilisons seulement la DocumentHandler et plus précisément la classe XMLDefaultHandler dans laquelle nous retrouvons toutes les méthodes événementielles pour la récupération d'informations :

- *startDocument ()* renvoyant un événement lié à l'ouverture du document.
- *startElement ()* renvoyant un événement lié à la rencontre d'un nouvel élément.
- *characters ()* renvoyant les caractères rencontrés.
- *endElement ()* renvoyant un événement lié à la fin d'un élément.
- *endDocument ()* renvoyant un événement lié à la fermeture du document.

Nous avons alors fait hérité notre parseur de la classe XMLDefaultHandler et implémenté ses méthodes de façon à indiquer les différentes opérations à effectuer selon notre position dans le fichier XML.

StartDocument () : indique l'ouverture du fichier XML.

StartElement () : indique quelle balise est ouverte et récupère le nom des objets contenus dans l'attribut type des balises <primitive></primitive>.

Characters () : récupère le nom des primitives se trouvant entre les balises <primitives><primitive>*nom*</primitive></primitives> et toutes les propriétés des objets contenues entre les balises de propriétés (ex : <position>*1.0 2.0 1.0*</position>. Chacune de ces propriétés est alors rajoutée à la liste des paramètres de l'objet courant définit par l'attribut type de la balise primitive en cours.

EndElement () : ajoute l'objet traité à la map des objets en associant par paire son nom et ses paramètres.

EndDocument () : indique la fin et donc la fermeture du fichier XML

Nous avons aussi défini un constructeur ainsi que deux autres méthodes :

- getPrimitives ()
- getObjs ().

Le constructeur Parseur (QFile & file) :

Construit une instance de l'analyseur qui parsera le fichier XML, dont le nom est passé en paramètre au constructeur.

std::vector<std::string> & getPrimitives() :

Retourne une référence du « vector » contenant tous les noms des primitives objets qui pourraient être définies par la suite dans le XML dans la partie :
<complexes></complexes>

std::multimap<std::string, GLParametres*> & getObjs() :

Retourne une référence de la map contenant tous les noms des objets associés à leurs paramètres qui devront être affichés sur la scène du chat.

L'affichage des objets

Le « vector » des primitives des objets définies par leur nom et la map des objets définis par leur nom et leurs paramètres sont récupérés dans le main d'affichage de la fenêtre QT grâce aux méthodes getPrimitives() et getObjs().
La map et le vector sont parcourus par un iterator de façon à ce qu'un objet de la map soit pris en compte si et seulement si le nom de celui-ci correspond bien à l'un des noms contenus dans le vector des primitives.
Si c'est le cas, on récupère alors le nom et les paramètres de l'objet afin de le construire et de l'ajouter ainsi à la scène de la fenêtre QT.

Ajout de l'objet à la fenêtre QT :

```
Eh->addGLObject(SuperFactory::getInstance().getGLObjectFactory(nom).createGLObject(parametres));
```

eh : handler de la fenêtre QT

Problèmes et solutions

La difficulté majeure rencontrée fut lorsqu'il a fallu retourner les objets à intégrer à l'affichage de la fenêtre QT. En effet, ces objets étaient, à la base, construits dans le code du parseur, à la fermeture du fichier XML.
La solution fut alors de créer une map qui stockerait seulement le nom des objets associés à leurs paramètres. Cette map est alors récupérée dans le main d'affichage afin d'être parcourue comme expliqué précédemment.

L'autre difficulté concerne la structure du fichier XML qui ne doit comporter aucun saut de lignes; le parseur traitait le saut de ligne comme correspondant à un texte vide entre deux balises et était ainsi considéré comme étant une propriété à ajouter aux paramètres des objets ce qui créait des erreurs à l'exécution du programme.

c. Le réseau

Fonctionnement du serveur

b) Initialisation du serveur via QT

- Saisie du numéro du port
- Saisie du nombre de joueurs
- Choix de l'univers

Une fois cette saisie validée, une nouvelle fenêtre QT s'affiche avec un bouton « quitter » et un thread est lancé pour accepter les connexions des clients.

c) Connexion des clients

Tant que l'on ne quitte pas
    Lancement d'un thread d'écoute pour chacun des clients qui se connectent
Fin Tant Que

d) Ecoute d'un client

Pour chaque répertoire à vérifier
    Réception de la liste des fichiers du client dans ce répertoire
    Comparaison avec les fichiers du serveur
    Envoi des fichiers manquants ou à mettre à jour
    Envoi de la fin de la mise à jour
Fin Pour

Envoi du nom de l'univers à charger au client.

Réception de la personnalisation du client (pseudo, groupe, nom de l'avatar…)
Transfert de ces données à tous les autres clients enregistrés
Enregistrement du client dans les données du serveur

Tant que le client ne se déconnecte pas
    Réception et traitement des messages du client
Fin Tant Que

Transfert de l'avis de déconnexion du client à tous les autres clients
Suppression du client des données du serveur
Fermer socket
Fin du thread

e) Traitement des messages

Selon code :

- code 1 : réception des données pour mettre à jour l'affichage de l'avatar du client chez les autres clients
  - Transfert de ces données à tous les autres clients enregistrés

- code 2 : réception d'un message chat
  Public : Transfert du message à tous les autres clients enregistrés
  Groupe : Transfert à tous les autres clients du groupe
  Privé : Transfert au client concerné

- code 3 : Réception des données du client (pseudonyme, groupe, avatar, …)
  - Transfert des données au client spécifié (c'est-à-dire au nouveau client qui récupère ainsi les informations de tous les clients déjà connectés)

Fonctionnement du client

a) Connexion au serveur via QT

- Saisie de l'IP du serveur
- Saisie du numéro de port du serveur

La validation de cette saisie entraîne la mise à jour des fichiers du client et l'affichage d'une nouvelle fenêtre de personnalisation :

```
Pour chaque répertoire à vérifier
Envoi de ses fichiers
    Tant qu'il y a des fichiers à mettre à jour ou à créer
        Réception du fichier
        Création ou mise à jour du fichier
    Fin Tant Que
Fin Pour

Réception du nom de l'univers à charger.
```

b) Personnalisation du client via QT

- Saisie du pseudo
- Saisie du groupe
- Choix de l'avatar

La validation de cette saisie entraîne le lancement d'un thread d'écoute du serveur, l'envoi de ces données au serveur et l'affichage de la fenêtre principale.

c) Actions possibles dans la fenêtre principale

- Dès que le joueur bouge :

➔ Envoi des données pour mettre à jour l'affichage chez les autres clients

- Dès que le joueur clique sur envoyer message :

➔ Envoie du message aux clients concernés

d) Ecoute du serveur

Tant que le client ne quitte pas
    Réception des messages
    Traitement des messages
Fin Tant Que

e) Traitement des messages

Selon code :

- code 1 : Réception des données pour mettre à jour l'affichage d'un autre client
  - Mise à jour des données d'affichage de ce client et rafraîchissement

- code 2 : Réception d'un message tchat
  Global : Affichage du message à la suite des autres dans la fenêtre globale
  Groupe : Affichage du message à la suite des autres dans la fenêtre groupe
  Privé : Affichage du message à la suite des autres dans la fenêtre privée de l'expéditeur

- code 3 : réception des données d'un nouveau client
  - Enregistrement de ces données (pseudo, nom de l'avatar, groupe, …), chargement de l'avatar, affichage
  - Envoi des infos du client à ce nouveau client (message de code 4)

- code 4 : Réception des données des autres clients enregistrés
  - Enregistrement de ces données (pseudo, nom du personnage, groupe,…), chargement de l'avatar, affichage

- code 5 : Réception de l'avis de déconnexion d'un autre client
  - Suppression des données de ce client

f) Quitter

- Dès que le joueur clique sur quitter

➔ Envoi de l'avis de déconnexion
➔ Fermer socket

Problèmes et solutions

Problème : Lorsque plusieurs clients parlent en même temps à un client particulier, les blocs d'informations peuvent se croiser et une information venant d'un client peut être parasitée par un bout d'information venant d'un autre client ce qui peut perturber le client qui reçoit ces messages.

Solution : Ces messages passent forcément par le serveur. C'est donc au niveau du serveur qu'il faut faire attention à ce que les threads qui transmettent les informations de leur client ne transmettent pas ces informations en même temps. Pour cela, nous utilisons des mutex qui sont verrouillés lorsqu'un thread envoie un bloc d'informations et déverrouillés une fois cet envoi fini. Ainsi, un seul thread à la fois peut envoyer un message à un client. Cette solution limite les risques de parasitage de l'information mais ne supprime pas tous les risques.

Problème : Les fichiers de taille importante ne peuvent pas être envoyés en une seule fois.

Solution : Les fichiers sont envoyés ligne par ligne.

d. Les interfaces graphiques

Description des interfaces

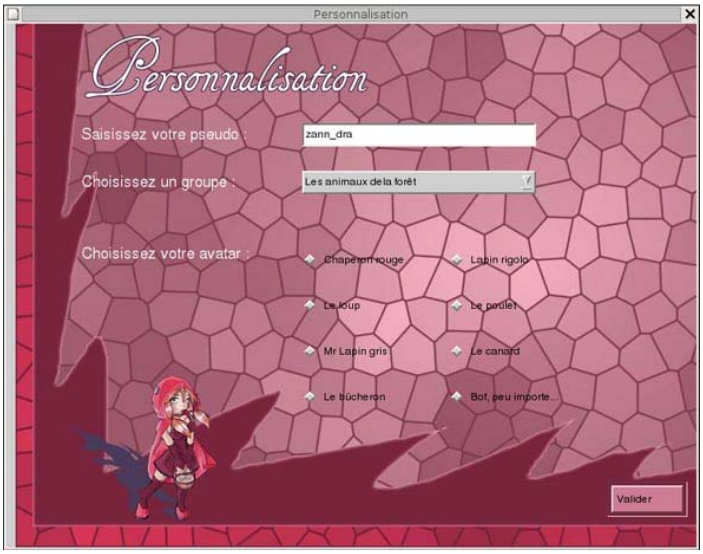
Les interfaces font le lien entre l'utilisateur et le programme. Elles ont été créées en QT (version 4.1) à l'aide du logiciel QTDesigner. Dès le départ elles ont obéi à un cahier des charges précis établi avec les autres membres du groupe en fonction de leurs besoins. Néanmoins, elles ont été amenées à évoluer au cours du temps, esthétiquement bien-sûr, mais également dans leur structure. De nouveaux champs tels que le choix d'un univers ou une fenêtre d'aide ont ainsi été rajoutés.

Nos interfaces sont les suivantes :

- **Côté serveur,** une fenêtre de configuration

- **Côté client,** Une fenêtre de type Qdialog qui prend en charge la connexion du client

- Une fois validée, elle laisse place à un autre Qdialog, la fenêtre de personnalisation :

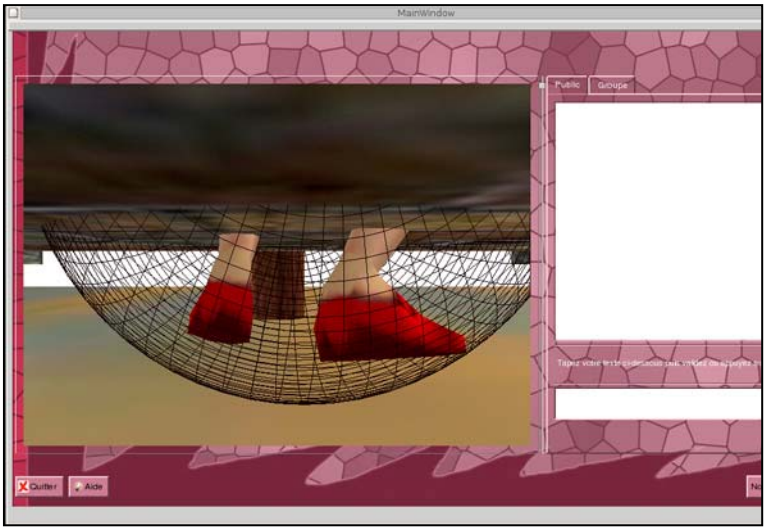


- Dans cette fenêtre, lorsque l'utilisateur coche le nom d'un des avatars, une petite Qdialog s'affiche en parallèle avec l'affichage du personnage sélectionné et la possibilité, grâce à deux curseurs, de choisir la position du buste et des jambes du personnage.

- Un bouton valider/Changement d'avatar, permet de revenir à la fenêtre précédente afin de, soit :
  - choisir un autre avatar si le précédent ne convenait pas.
  - valider la personnalisation.

- Cette dernière validation met fin à la personnalisation.

- L'utilisateur découvre alors la MainWindow, restée en fond durant ces premières étapes, qui contient sur sa gauche la scène venant d'être chargée (univers et personnages) et sur sa droite le chat public et de groupe. L'utilisateur peut quitter le programme, afficher une aide, ou se reconnecter, et cela à tout moment.



Le clic sur un personnage de la scène, une fenêtre de chat privé mais on ne peut pas encore parler avec l'autre avatar.

Le choix des éléments s'est fait dans un compromis entre esthétique et ergonomie.

Les trois étapes clés

Il y eut trois phases délicates en rapport avec les interfaces :

1. **Création des fenêtres et lien entre les différentes interfaces.**

- Au départ, les fenêtres étaient toutes des **MainWindow**. Or, un programme ne contient qu'une unique MainWindow qui comme son nom l'indique, est la fenêtre principale, c'est-à-dire pour nous
  - celle de la scène et du chat côté utilisateur,
  - celle de configuration du serveur côté serveur.

Nous désirions des fenêtres qui s'enchaînent bien distinctes les unes des autres. Il a donc fallu transformer les fenêtres initiales en boîte de dialogue (**Qdialog)**, qui, à la différence des **Qwidgets** qui changent seulement le contenu d'une fenêtre existante, constituent des fenêtres indépendantes.

- La démarche de création d'une fenêtre est la suivante :

  - Une fois enregistrée en **NomDeLaFenetre.ui** sous QTDesigner, il faut créer une classe de type **QdialogNomDeLaFenêtre** (ex : QdialogConnexion,) qui hérite en public de Qdialog afin de récupérer ses méthodes, et en privé de **Ui_ NomDeLaFenetre** (ex : Ui_Connexion.).
  - Sur le même modèle, une classe MainWindow hérite de Designer_MainWindow.ui et de MainWindow. A la compilation, les .ui vont générer un .hpp (ex : Connexion.hpp) contenant la déclaration des champs tels qu'ils sont créés et nommés dans QTDesigner, et leurs propriétés. Pour construire une interface, il faut faire un **setupUi (this)** dans le constructeur de la classe héritant du .ui.

- Pour faire se succéder les fenêtres, il suffit alors d'appeler le constructeur de la classe **QdialogNomDeLaFenetreSuivante** dans une des méthodes de la classe de la fenêtre courante.

2. Lien entre un élément de l'interface et une action et récupération des données.

- Pour récupérer les données entrées ou choisies par l'utilisateur, il faut utiliser le nom des champs présent dans le .hpp généré par le .ui et lui appliquer une méthode décrite dans le documentation de QT 4.1.
  - Par exemple : nbJoueurs->value() récupère dans un int la valeur du QspinBox nommé nbJoueurs dans QtDesigner.

- Le lien entre le clic d'un bouton et l'événement (définit par une méthode) qu'il doit engendrer se fait dans le constructeur de la fenêtre courante par le biais de la méthode connect qui prend en paramètre le type du signal (ici : cliqué) et la méthode pour l'événement stockée dans un slot.
  - Par exemple : connect (this->quitter, SIGNAL (clicked ()), this, SLOT (quit())); associe au bouton quitter de la fenêtre courante la méthode quit (), que l'on déclare comme public slots dans le .hpp.

3. Affichage du personnage choisi par l'utilisateur

- Au début du projet, nous nous contentions d'afficher un personnage par défaut, dont nous entrions le nom dans le constructeur de la classe GLPersonnage. Il a donc fallu transformer la classe GLPersonnage afin que son constructeur n'ait plus d'argument et implémenter une méthode **setName ()** dans Personnage qui prend en paramètre le nom de l'avatar choisi lors de la validation de la fenêtre de Personnalisation.

- Un **GLPersonnage PersonnagePrincipal** est déclaré dans la **QTApplication**, classe qui fait l'appel à la **MainWindow** qui n'affiche au départ pas de personnage ni de scène. C'est seulement à la validation de la fenêtre de Personnalisation que s'effectue le setName de personnage principal, et que l'on ajoute le personnage à la liste des objets à afficher, c'est à dire à notre **GLEventHandlerForFPS** déclaré dans la mainWindow.

- Cela nous a posé problème car l'initGl de la classe Personnage qui utilise le nom de l'avatar, appelé par l'initGL de GlEventHandlerForFPS déclaré dans la mainWindow était appelé trop tôt, soit avant que l'initialisation ait été faite.

Améliorations finales

Une fois ces problèmes réglés, il a fallu améliorer les interfaces en implémentant :

- Le rafraîchissement de la fenêtre de chat chaque fois qu'un client parle dans ladite fenêtre, sans quoi il fallait attendre qu'un utilisateur réécrive et envoie son message pour qu'il voie s'afficher les messages précédents ;
- L'affichage de l'arrivée et du départ des clients dans la fenêtre de chat général ;
- L'affichage du nom de celui qui parle ;
- La possibilité de se reconnecter et donc de redémarrer le processus à zéro (lien vers la première fenêtre de connexion) ;
- Le non-accès en écriture du champ d'affichage des messages de chat ;
- La possibilité grâce à la touche tabulation de passer d'un champ à un autre dans un ordre prédéfini, grâce au mode Edit tab Order de QTDesigner ;
- La possibilité d'appuyer sur entrée à la place de cliquer sur valider dans chacune des fenêtres;
- Le redimensionnement de la fenêtre principale, qui a supposé d'associer tous les éléments et de les placer dans des QVBoxLayouts et des Widgets.
- Pour finir, nous avons défini une identité visuelle et un logo et l'avons décliné sur chacune des interfaces.

Problèmes

- La reconnexion ne fonctionne pas, sans doute à cause d'un problème de réseau.
- Le redimensionnement de la fenêtre principale est limité. On ne peut pas vraiment la réduire beaucoup, car il a fallu fixer une taille minimum à la scène sans quoi elle apparaissait très petite.

3. Résultats

a. Ce qui marche…

- La connexion du serveur sur le réseau avec le choix de l'univers.
- L'initialisation et la personnalisation (pseudo, avatar) du client.
- La connexion du client au serveur via le réseau.
- Le chargement des objets texturés, avec prise en compte de l'univers choisi, et des personnages dans la scène.
- La mise à jour de la scène au cours du jeu (nouveaux clients connectés au serveur).
- L'envoi des messages entre personnes du même groupe, en public, et en privé.
- Le placement et le déplacement de la caméra via les touches clavier et la souris.
- Le picking sur les boîtes englobantes des MD3.
- L'application du choix de la position du personnage dans la scène après la personnalisation de l'avatar par le client.
- Génération d'un personnage aléatoirement.
- Le picking sur les personnages pour l'envoi des messages privés (affichage de la fenêtre uniquement (mais peut être qu'un jour…)).

b. Ce qui ne marche pas…

- La lenteur des déplacements au clavier dans la scène.
- Le déplacement des autres personnages dans la scène.

c. Bugs connus

- Problème lors du chargement des fichiers de temps en temps
- Lors de la connexion des clients, le serveur se déconnecte de temps en temps

d. Améliorations possibles…

- Configuration des touches clavier.
- Gestions des collisions.
- Affichage dans la fenêtre de chat privé d'une image de l'avatar de notre interlocuteur.
- Possibilité de passer à la ligne dans le champ d'écriture du chat
- Améliorer l'affichage de la scène !!

Conclusion

Ce projet a élargi nos compétences car il était très complet et mobilisait des connaissances en C , en infographie et en réseau. Mais au-delà, l'expérience a été particulièrement enrichissante car elle nous a mis dans des conditions quasi-professionnelles de travail en groupe avec des compétences personnelles diverses, un délai à tenir et un cahier des charges bien précis. Le projet nous a ainsi fait réaliser l'importance de la phase de conception et celle d'une répartition des tâches judicieuse ainsi que d'une communication importante dans le groupe, car le plus dur n'a pas forcément été le développement de chacun des modules mais le rassemblement final des différentes entités du projet.

Bibliographie, Webographie

Pour la documentation Qt et Qtdesigner

http://doc.trolltech.com/4.1/index.html
http://doc.trolltech.com/4.1/designer-manual.html

Pour les modèles md3

http://www.planetquake.com/polycount/

Pour l'aide

http://www-igm.univ-mlv.fr/~pchaudey/
http://www-igm.univ-mlv.fr/~fdesorbi/web/