



Lycée La Fayette

Champagne-sur-Seine • Fontaineroux



Projet 1790-5

Scénarios d'intervention



Kelyan CHAUFFOURIER ; Franck GAILLARD ; Quentin DELION ; Gregory GAILLARD

Sommaire

1.Remerciements.....	4
2.Présentation.....	5
2.1.SDIS 77 / UDSP 77.....	5
2.2.Cahier des charges 1790.....	5
2.2.1.1790-1.....	5
2.2.2.1790-2.....	6
2.2.3.1790-3.....	6
2.2.4.1790-4.....	6
2.2.5.1790-5.....	7
3.Analyse.....	8
3.1.Ressources.....	8
3.2.Contraintes.....	8
3.3.Notion de scénario.....	9
3.4.Format des fichiers OBJ7/OBJ8.....	9
3.4.1.Format OBJ7.....	9
3.4.2.Format OBJ8.....	10
3.5.Modélisation.....	11
3.5.1.Diagramme de déploiement.....	11
3.5.2.Diagramme des cas d'utilisation.....	11
3.6.IHM SceneEditor.....	12
3.7.IHM ScenePlayer.....	13
3.8.Protocole UDP / X-Plane.....	13
3.9.Serveur.....	14
4.Spécifications.....	15
4.1.SceneEditor (Kelyan CHAUFFOURIER).....	15
4.1.1.Introduction:.....	15
4.1.2.Analyse des IHM.....	15
4.1.3.Format des fichiers SCN.....	18
4.1.4.Diagramme de classes.....	22
4.1.5.Conclusion.....	22
4.2.Stockage & accès (Franck GAILLARD).....	23
4.2.1.Cahier des charges personnel.....	23
4.2.2.Moyens mis en œuvre.....	23
4.2.3.Les classes QNetworkAccessManager.....	23
4.2.4.Protocole FTP et SFTP.....	24
4.2.5.Programmes de test.....	24
4.2.6.La classe QScenarioTransfert.....	26
4.2.7.Problèmes.....	30
4.2.8.Conclusion.....	31
4.3.ScenePlayer (Quentin DELION).....	32
4.3.1.Cahier des charges.....	32
4.3.2.L'Interface Homme-Machine.....	32
4.3.3.Diagramme de classes simplifié.....	33
4.3.4.Présentation de Qt/QML.....	34
4.3.5.Fonctionnement.....	34

4.3.6.Comparaison entre un fichier SCN et le résultat dans ScenePlayer.....	39
4.3.7.Conclusion.....	40
4.4.Liaison UDP (Gregory GAILLARD).....	41
4.4.1.Mon équipe.....	41
4.4.2.Mon cahier des charges :.....	41
4.4.3.Requête OBJN/OBJL et RPOS.....	41
4.4.4.Démonstration de l'action de ScenePlayer dans X-Plane.....	43
4.4.5.Difficultés rencontrées :.....	44
4.4.6.Conclusion :.....	44

1. Remerciements

L'ensemble des étudiants membres du projet 1790-5 tient à remercier le Service Départemental d'Incendie et de Secours ainsi que l'Union Départementale des Sapeurs-Pompiers de nous avoir confié ce projet très ambitieux et intéressant. Nous sommes tous très honorés d'avoir été choisis pour travailler sur ce projet qui a pour but de renforcer la sécurité civile.

Ce projet était une expérience très enrichissante, tant d'un point de vue personnel que professionnel. Grâce à lui, nous avons pu découvrir le monde de l'aéronautique, ainsi que des subtilités sur le vol en hélicoptère dont nous n'avions pas connaissance.

Nous souhaitons également remercier l'équipe pédagogique, notamment M. Alain Menu, qui nous a accompagné et aidé tout au long de notre projet.

2. Présentation

2.1. SDIS 77 / UDSP 77

Le SDIS 77 développe depuis quelques années (2003) une politique d'utilisation de moyens héliportés dans le cadre de ses missions péri-opérationnelles et opérationnelles. Ces missions aussi variées que la reconnaissance aérienne, la recherche de personnes, la projection de spécialistes (GRIMP, plongeurs, équipes cynophiles...) ou le transport sanitaire héliporté (TSH) impliquent l'ensemble des personnels opérationnels du SDIS 77.

La nécessité de réaliser des formations initiales et continues est primordiale au regard de ces missions. L'utilisation des hélicoptères opérationnels à des fins de formation est rendue difficile, car ces derniers sont utilisés de plus en plus pour les missions de secours. La réalisation d'un simulateur sur la base d'une cellule d'hélicoptère pourra répondre aux exigences de formation, de disponibilités et de variété des scénarios de missions. De la formation à la navigation destinée aux officiers, à la sensibilisation aux règles de sécurité aux abords des aéronefs impliquant chaque sapeur-pompier en passant par la para médicalisation à bord d'un hélicoptère impliquant le personnel de santé, la simulation est aujourd'hui un outil incontournable en matière de formation.

Les objectifs de ce projet sont multiples :

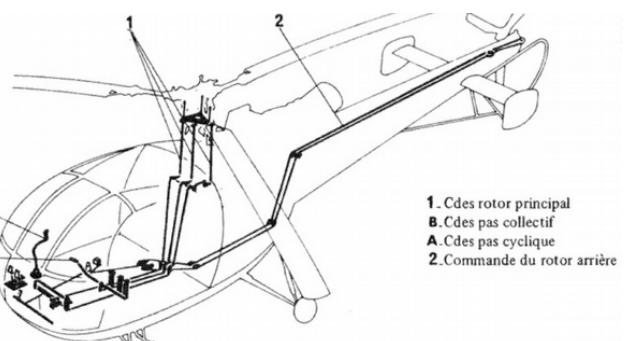
- Formation des officiers sapeurs-pompiers à la navigation aérienne en les mettant en situation ;
- Familiarisation des personnels avec l'environnement physique d'un hélicoptère ;
- Entraînement à la réalisation des missions (reconnaissances, évaluations...).

2.2. Cahier des charges 1790

2.2.1. 1790-1

La cellule fournie par le client dispose des commandes de vol traditionnelles d'un aéronef à voilure tournante :

- levier de commande du pas collectif du rotor principal ;
- manche cyclique (inclinaison longitudinale et latérale du rotor principal) ;
- palonnier de contrôle du pas du rotor arrière.



Il est donc envisagé de placer des capteurs de déplacement liés aux bielles ; ces capteurs peuvent par exemple être de type potentiomètre à câble. La mise en place des capteurs sur la cellule sera assurée par l'équipe enseignante.

Le simulateur de vol retenu est X-Plane version 10 ou supérieure. Ce logiciel professionnel est conçu pour recevoir les informations de contrôle d'attitude des aéronefs via une manette de jeu multiaxes.

L'enjeu du projet 1790-1 est donc de concevoir un « joystick » capable de récupérer les positions angulaires des commandes de vol de l'Alouette III et de se comporter comme un périphérique standard USB/HID.

2.2.2.1790-2

Le tableau de bord principal de la cellule fournie par le client est dépourvu de tout instrument. La solution retenue, en accord avec le client, consiste à remplacer ces instruments par une représentation virtuelle dynamique sur un écran standard ; écran sur lequel sera placé un masque à trous afin de reproduire au mieux le tableau de bord d'origine ; les coins supérieurs arrondis interdisent en effet la mise en place d'une dalle graphique à l'intérieur du pupitre. La réalisation et mise en place du masque seront assurées par l'équipe enseignante.

Le simulateur de vol retenu est X-Plane version 10 ou supérieure. Ce progiciel professionnel est capable de fournir périodiquement, sous forme de datagrammes UDP/IP (X-Plane Datarefs), toutes les informations nécessaires à l'animation des instruments de vol.

Instruments gyroscopiques :

3 – indicateur d'assiette (horizon artificiel)

11 – indicateur de virage/dérapage (bille)

7 – conservateur de cap

Instruments utilisant la pression statique :

1 – indicateur d'altitude (altimètre)

2 – indicateur de vitesse verticale (variomètre)

4 – indicateur de vitesse (anémomètre)

1	2	3	4
5	6	7	8
9	10	11	12

13

Instruments de navigation :

5 – VOR

6 – ILS

Autres instruments :

8 – indicateur/calculateur de pas

13 – horloge/chronomètre

L'objectif du projet 1790-2 est donc de développer une application dédiée (C++/Qt) assurant le dessin et l'animation d'instruments sans réglages (VSI + AH + DG + TC) à partir des données temps-réel fournies par le progiciel X-Plane. L'application doit être conçue de manière à accepter facilement l'intégration d'autres instruments ; les dimensions, le positionnement (relatif et/ou absolu) sur la dalle écran et les Datarefs associés de chaque instrument doivent être paramétrables (par exemple par fichier de configuration type CSV).

2.2.3.1790-3

Le projet 1790-3 est complémentaire du projet 1790-2. Son premier objectif est de permettre la remise en service de divers éléments des tableaux de bord tels que les voyants d'alarme ou autres interrupteurs. La liste de ces éléments inclut les organes de réglage des instruments de vol.

Le deuxième objectif du projet 1790-3 est de compléter « the standard six » par le développement des instruments réglables ALT (réglage QNH) et ASI (réglage TAS).

2.2.4.1790-4

Le projet 1790-4 (4 étudiants) concerne la finalisation de l'instrumentation des commandes de vol de la cellule, de la reproduction d'instruments du tableau de bord et de l'interfaçage de l'ensemble avec le progiciel de simulation X-Plane. L'équipe 1790-4 a donc pour charge de fiabiliser, de compléter et de documenter techniquement les solutions dégagées lors de la phase 1 du projet (session 2016).

La phase 1 du projet a permis d'équiper chacun de ces 4 axes d'un capteur potentiométrique rotatif à câble et d'en renvoyer les informations vers X-Plane par l'intermédiaire d'un Arduino Uno vu comme un périphérique standard USB/HID (joystick). La structure du « joystick » doit permettre d'évoluer dans le futur vers un nombre d'axes plus important, par exemple pour intégrer d'autres commandes telles que le débit (commande lié à la turbine) ; et vers la prise en charge de boutons, par exemple pour prendre en charge ceux situés sur les manettes du manche cyclique et du collectif. Le

prototype réalisé doit être finalisé et amélioré en termes de performances.

Cette phase 2 du projet doit assurer la finalisation des 6 instruments de base. Le tableau de bord sera complété par des instruments fournis par l'équipe enseignante (emplacements 5, 6 et 8). Le code produit doit être harmonisé, documenté, et disposer d'un dossier technique incluant notamment les diagrammes UML idoines et un tutoriel devant permettre la réalisation future d'instruments complémentaires.

Les témoins et actionneurs du tableau de bord sont interfacés par un module Arduino de type « esclave Modbus over IP » développé lui aussi lors de la phase 1 du projet. Ce sous-ensemble est considéré comme terminé et opérationnel ; il sera dupliqué si nécessaire pour prendre en charge d'autres tableaux de bord de l'appareil (plafonnier, console moteur... suivant exigences à venir du client).

La phase 1 du projet a permis de développer une première version d'un logiciel de pilotage entièrement configurable par un fichier texte au format CSV. L'application est conçue de manière à accepter facilement l'intégration d'autres instruments ; les dimensions, le positionnement (relatif et/ou absolu) sur la dalle écran et les Datarefs associés de chaque instrument sont paramétrables. Le fichier de configuration permet aussi d'associer les témoins et actionneurs du tableau de bord avec des informations ou grandeurs internes au simulateur de vol. Le logiciel doit être finalisé et testé en charge de manière à caractériser les performances liant la fluidité d'animation des instruments avec le taux de trafic réseau (échanges UDP) ; ceci en fonction des performances graphiques exigées pour les scènes 3D et des caractéristiques de la machine hôte (AlienWare fourni par le client).

2.2.5. 1790-5

Le projet 1790-5 est relatif à la recherche et au prototypage de solutions pour intégrer des scénarios d'intervention aux scènes 3D du simulateur X-Plane ; les scénarios seront fournis par le client.

Pour mémoire, l'objectif principal recherché par le demandeur est la reproduction la plus réelle possible de situations d'interventions héliportées ; chaque intervention devant pouvoir être scénarisée à partir d'une expérience vécue et/ou d'une situation hypothétique. Une séance de formation sur le simulateur sera contrôlée par un instructeur chargé de piloter le scénario d'intervention ; cet instructeur pourra être positionné à l'extérieur de la cellule de l'hélicoptère.

Le projet 1790-5 consiste donc à définir les procédures et développer les outils nécessaires devant :

- d'une part permettre d'intégrer de tels scénarios aux scènes 3D proposées par X-Plane ;
- d'autre part autoriser le contrôle des scénarios en temps réel.

note : Le choix, la définition et la création des objets 3D à intégrer (bâtiments, véhicules, personnages...) est à la charge du client.

3. Analyse

3.1. Ressources

Documentation spécifique :

- Getting/Setting Data from/to X-Plane.



Ressources matérielles :

- Cellule d'Alouette III n°1790 ;
- Poste embarqué AlienWare Area-51 « simulateur de vol » équipé de son système de vidéo-projection (x3) ;
- Système de sonorisation de la cellule ;
- Tablettes graphiques type Sony XperiaZ (Android) et/ou iPad (iOS).

Ressources logicielles :

- Logiciel X-Plane version 10 ou + ;
- Environnement de développement C++/Qt version 5.7 ou + ;
- Classes Qt [QamSockets](#) et [QNetworkAccessManager](#) ;
- Outils connexes X-Plane opensource.



3.2. Contraintes

Budget alloué	Matériels et / ou logiciels imposés / technologies utilisées	Fiabilité et sécurité
Poste dédié au simulateur et le serveur, licence logiciel (X-Plane).	Équiper dans la cellule, PC de simulation sous Windows 7, fonctionnement uniquement sous X-Plane.	Faire des programmes fiables, faciles à régler et à utiliser, mise à disposition dossiers techniques matériels et/ou logiciels.

3.3. Notion de scénario

Les fichiers scénario (extension « .scn ») contiennent les informations nécessaires à ScenePlayer pour jouer les scénarios. Ils sont écrits au format XML qui est un format « human readable ».

Les informations contenues dans ces fichiers sont :

- La liste des objets 3D utilisés pour le scénario, ainsi que leur chemin d'accès sur la machine hôte (AlienWare) ;
- Une liste d'événements associés à des actions, qui constituent le scénario.

Les événements consistent à associer des actions à un déclencheur. Il y a trois types de déclencheur :

- Manuel : l'événement est déclenché manuellement par le moniteur depuis ScenePlayer ;
- Temps : l'événement est déclenché automatiquement au bout d'un certain temps (relatif ou absolu) ;
- Sur zone: l'événement est déclenché automatiquement quand l'hélicoptère arrive dans une certaine zone.

Une fois que l'événement est déclenché, les actions qui lui sont associées s'exécutent. Les actions peuvent être :

- Apparition d'objet ;
- Déplacement d'objet ;
- Changement d'état d'un objet (apparition ou disparition de fumée) ;
- Disparition d'objet.

Les fichiers d'extension « .obj » sont les fichiers utilisés par X-Plane pour représenter des objets 3D. Ce format de fichier a évolué et possède plusieurs versions. X-Plane version 10, actuellement utilisé, peut utiliser les formats OBJ7 et OBJ8. X-Plane version 11, la plus récente, ne peut plus utiliser le format OBJ7.

3.4. Format des fichiers OBJ7/OBJ8

3.4.1. Format OBJ7

Le format OBJ7 est utilisé pour créer les objets, un fichier « .obj » pour un objet. Ce format est *human readable*. Ci-dessous, un aperçu de ce format.

```
A
700
OBJ

highway      //
quad        //
-12.518906 91.761375 2.854788    0.678690 0.996088
-12.518906 91.761375 -2.824575    0.637257 0.996088
-12.518906 89.274315 -2.824575    0.637257 0.938465
-12.518906 89.274315 2.854788    0.678690 0.938465
...
end      //
```

Le caractère 'A' ou 'T' permet de savoir quel fin de ligne sont utilisés, Carriage Return (CR) ou Carriage Return Line Feed (CRLF). Le 'A' indique que le fichier a été fait avec des CR et le 'T' indique des fins de ligne CRLF.

Le nombre 700 indique la version de l'objet (ici, version 7).

La chaîne de caractère "OBJ" permet à X-Plane de savoir que c'est un objet.

La chaîne de caractère "highway" indique le nom de la texture associée. C'est un fichier au format BMP ou PNG qui sera plaqué sur la forme de l'objet.

La chaîne "quad" indique le type de face fabriquée (ici, un rectangle). Il en existe d'autre ("tri" pour un triangle par exemple)

Les quatre lignes suivantes sont des coordonnées. Les trois premiers nombres sont les coordonnées d'un point dans l'environnement 3D. Les deux derniers sont les coordonnées du même point dans la texture associée, donc en 2D.

La chaîne "end" indique la fin de l'objet. Tout ce qui est écrit après n'est pas pris en compte par X-Plane.

3.4.2. Format OBJ8

Tout comme OBJ7, OBJ8 est un format d'extension « .obj ». Voici un extrait de ce format.

```
A  
800  
OBJ  
  
TEXTURE      ../../textures/vehicles_10_ALB.png  
...  
  
VT      1.2200    0.0084    5.1742  0.000 -0.707  0.707    0.7532  0.3071  
VT     -1.2200    0.0084    5.1742  0.000 -0.707  0.707    0.8407  0.3071  
VT     -1.2200    0.3805    5.5463 -0.000 -0.174  0.985    0.8407  0.2996  
...
```

Comme OBJ7 les trois premières lignes indiquent :

- le type de fin de ligne ;
- la version de l'objet ;
- la chaîne "OBJ" pour dire que c'est un objet.

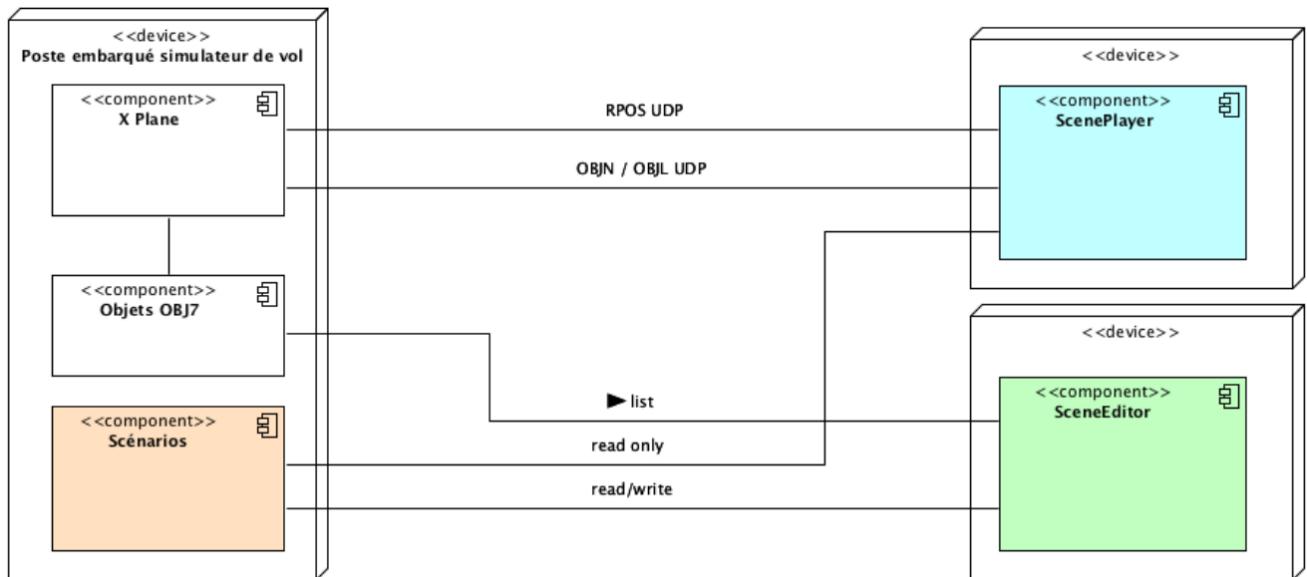
"TEXTURE" indique la texture associée à l'objet.

Une des grandes différences est que le format OBJ8 ne modélise pas un objet de la même manière. Avec OBJ7, on pouvait utiliser des rectangles, des triangles, etc. OBJ8 fait principalement des triangles, les rectangles ayant disparus.

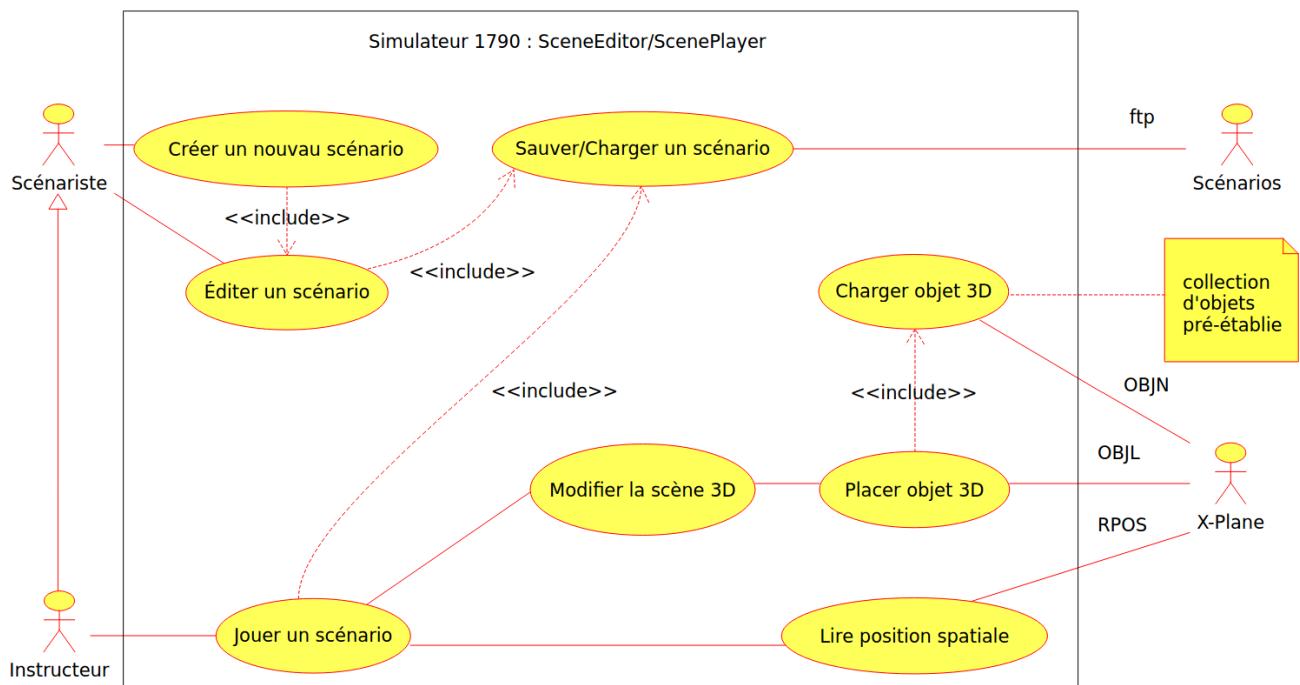
"VT" indique une entrée unique dans la table des triangles. Les huit nombres représentent un triplet de coordonnées de localisation, un triplet normal, et une paire formant une coordonnée de texture.

3.5. Modélisation

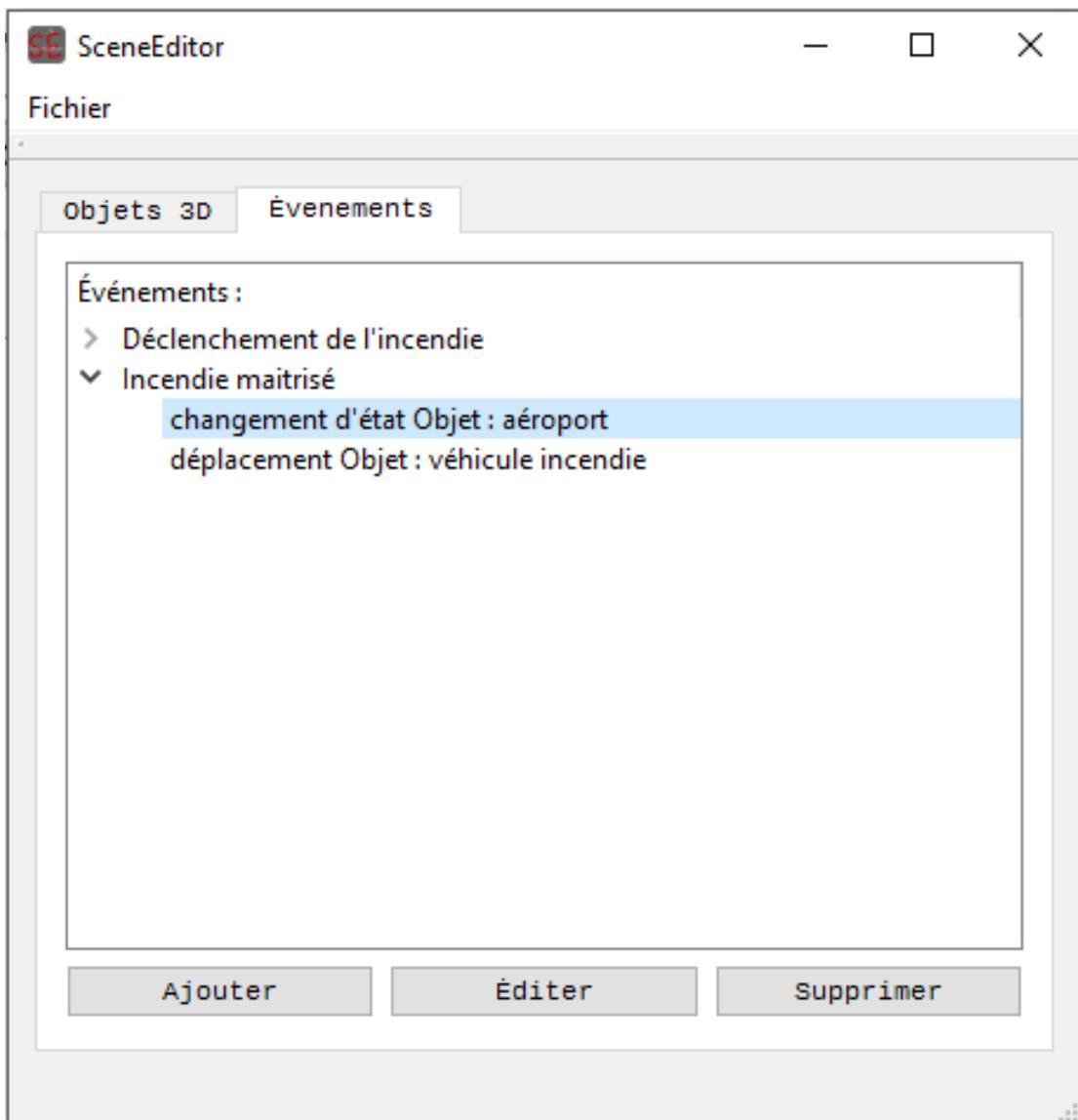
3.5.1. Diagramme de déploiement



3.5.2. Diagramme des cas d'utilisation

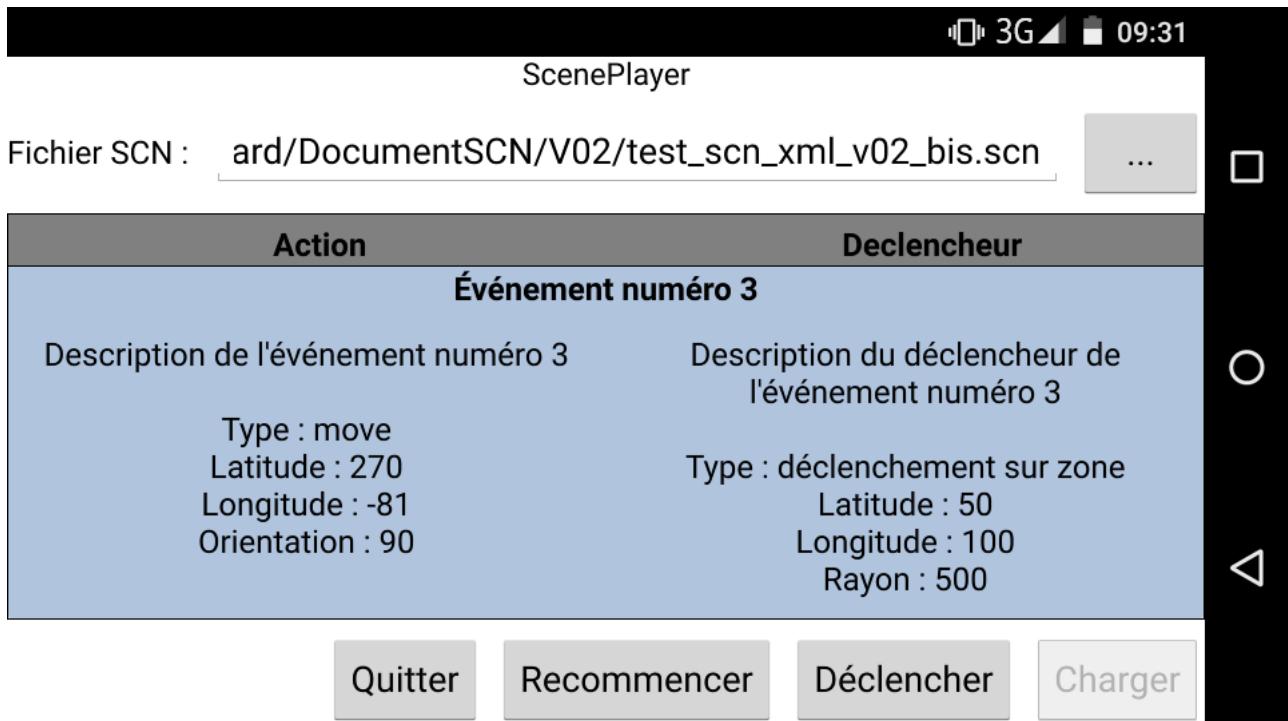


3.6. IHM SceneEditor



Cette interface est celle qui apparaît au lancement du programme. Elle permet l'affichage de la liste des objets, de la liste des événements ainsi que les actions qu'ils déclenchent.

3.7. IHM ScenePlayer



Cette interface permet de voir la liste des événements avec les actions et le déclencheur qui les composent.

3.8. Protocole UDP / X-Plane

Le protocole UDP (User Datagram Protocol) était imposé pour la communication avec X-Plane. C'est sous forme de datagrammes UDP/IP que X-Plane peut fournir les informations nécessaires pour animer les instruments. UDP est l'un des principaux protocoles de télécommunication utilisés par Internet. Il fait partie de la couche transport du modèle OSI, il appartient à la couche 4, comme TCP.

Le rôle de ce protocole est de permettre la transmission de données de manière très simple entre deux entités, chacune étant définie par une adresse IP et un numéro de port. X-Plane impose le port 49000. Contrairement au protocole TCP, il fonctionne sans négociations : il n'existe pas de procédure de connexion préalable à l'envoi des données. Cependant, l'UDP possède un système de vérification de trames. Une trame est composée de 5 caractères, les 4 premiers correspondent au type du message et le dernier est le « 0 de fin de chaîne » servant à vérifier la réception entière de la trame. UDP ne garantit donc ni la bonne livraison des datagrammes à destination, ni leur ordre d'arrivée. Il est également possible que des datagrammes soient reçus en plusieurs exemplaires.

L'intégrité des données est assurée par une somme de contrôle sur l'en-tête. L'utilisation de celle-ci est facultative en IPv4 mais est obligatoire avec IPv6. Si un hôte n'a pas calculé la somme de contrôle d'un datagramme émis, la valeur de celle-ci est fixée à zéro. La somme de contrôle tient compte des adresses IP source et destination.

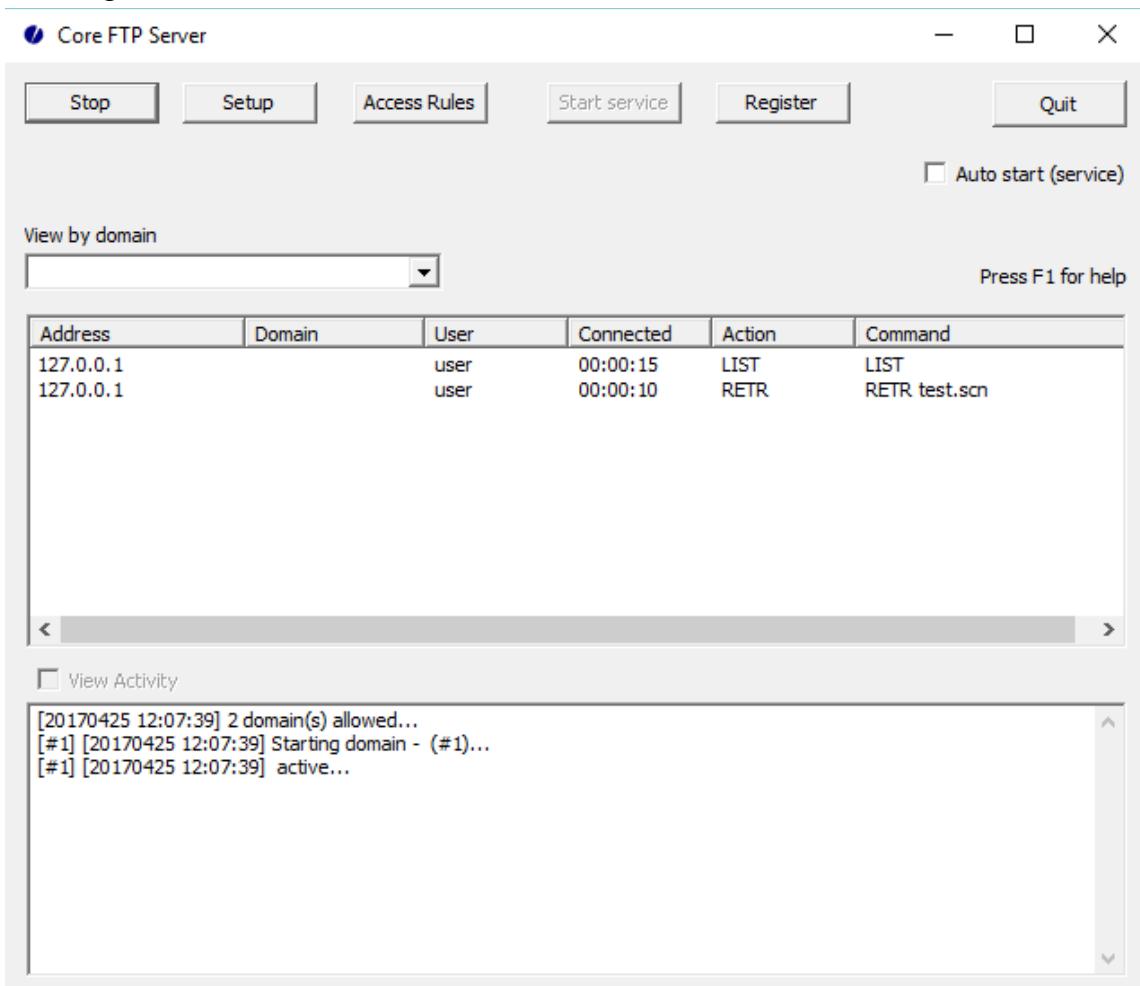
La nature d'UDP le rend utile pour transmettre rapidement de petites quantités de données

depuis un serveur vers de nombreux clients, ou bien dans des cas où la perte d'un datagramme est moins gênante que l'attente de sa retransmission.

3.9. Serveur

Les applications d'édition et de jeu doivent pouvoir interagir avec le logiciel X-Plane, charger et jouer un fichier 'scénario'.

Ces fichiers seront donc stockés sur un serveur, hébergé par la machine AlienWare sur laquelle s'exécute le logiciel de simulation X-Plane.



Le serveur utilisé sera 'Core FTP Server', gratuit et simple d'utilisation. Celui-ci va permettre de stocker les fichiers 'scénario' ainsi que la banque d'objet 3D. Les applications SceneEditor et ScenePlayer vont pouvoir communiquer avec cette machine, via le réseau local du simulateur. Ces applications utiliseront une classe basé sur les classes de [QNetworkAccessManager](#) afin de pouvoir charger, jouer ou sauvegarder un scénario.

De plus, le protocole de communication restera standard en utilisant SFTP. SFTP (Secure File Transfer Protocol) est un protocole de transfert de fichiers de manière sécurisé grâce à SSH.

4. Spécifications

4.1. SceneEditor (Kelyan CHAUFFOURIER)

4.1.1. Introduction:



Afin de répondre aux besoins du projet, il était nécessaire de fournir au client un outil lui permettant d'éditer les scénarios, ainsi que d'en créer. En décomposant, il est possible de considérer un scenario comme étant :

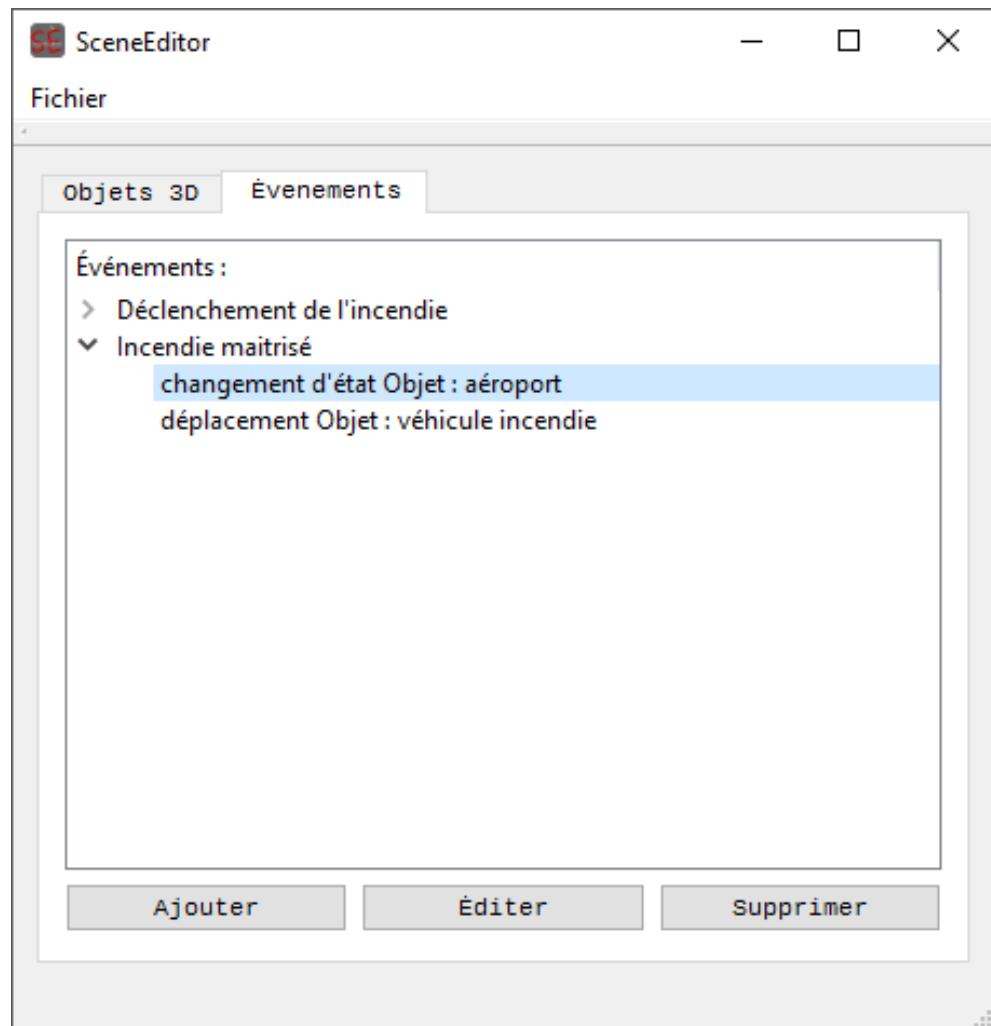
- une liste d'objets 3D utilisés par le scénario ;
- une liste d'événements, associant chacun un déclencheur à une ou plusieurs actions.

Ce programme doit donc permettre d'éditer ces deux listes.

4.1.2. Analyse des IHM

Pour répondre à ces besoins, j'ai créé les IHM suivantes :

IHM principale :



L'IHM principale de SceneEditor permet :

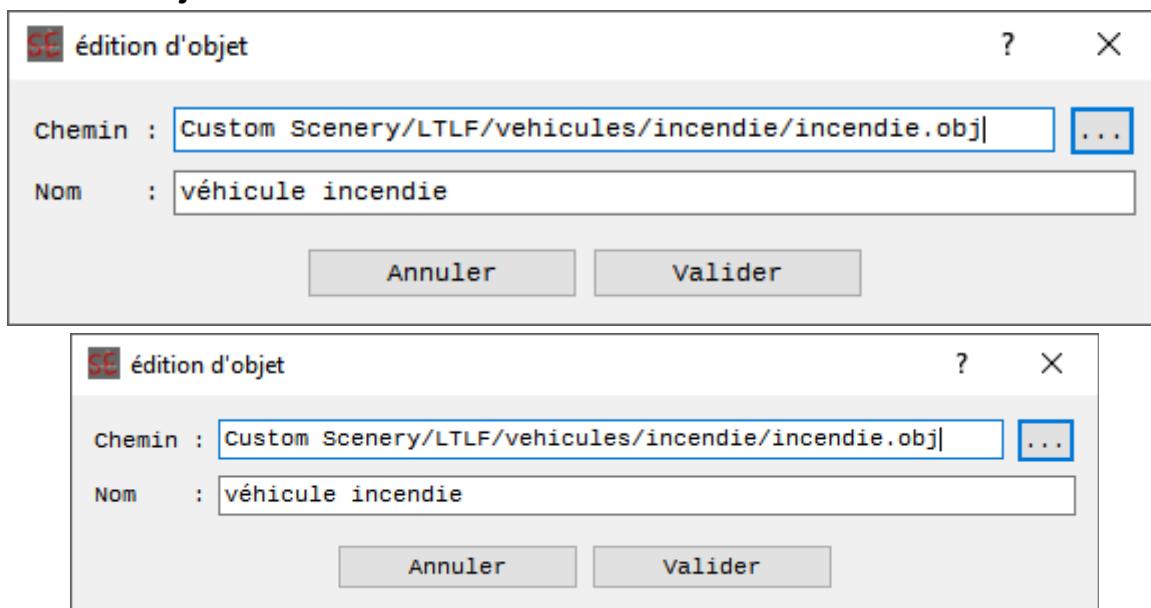
- L'ouverture d'un fichier scenario ;
- La création d'un nouveau fichier scenario ;
- L'enregistrement du fichier scenario modifié ;
- La fermeture de SceneEditor.

Ainsi que :

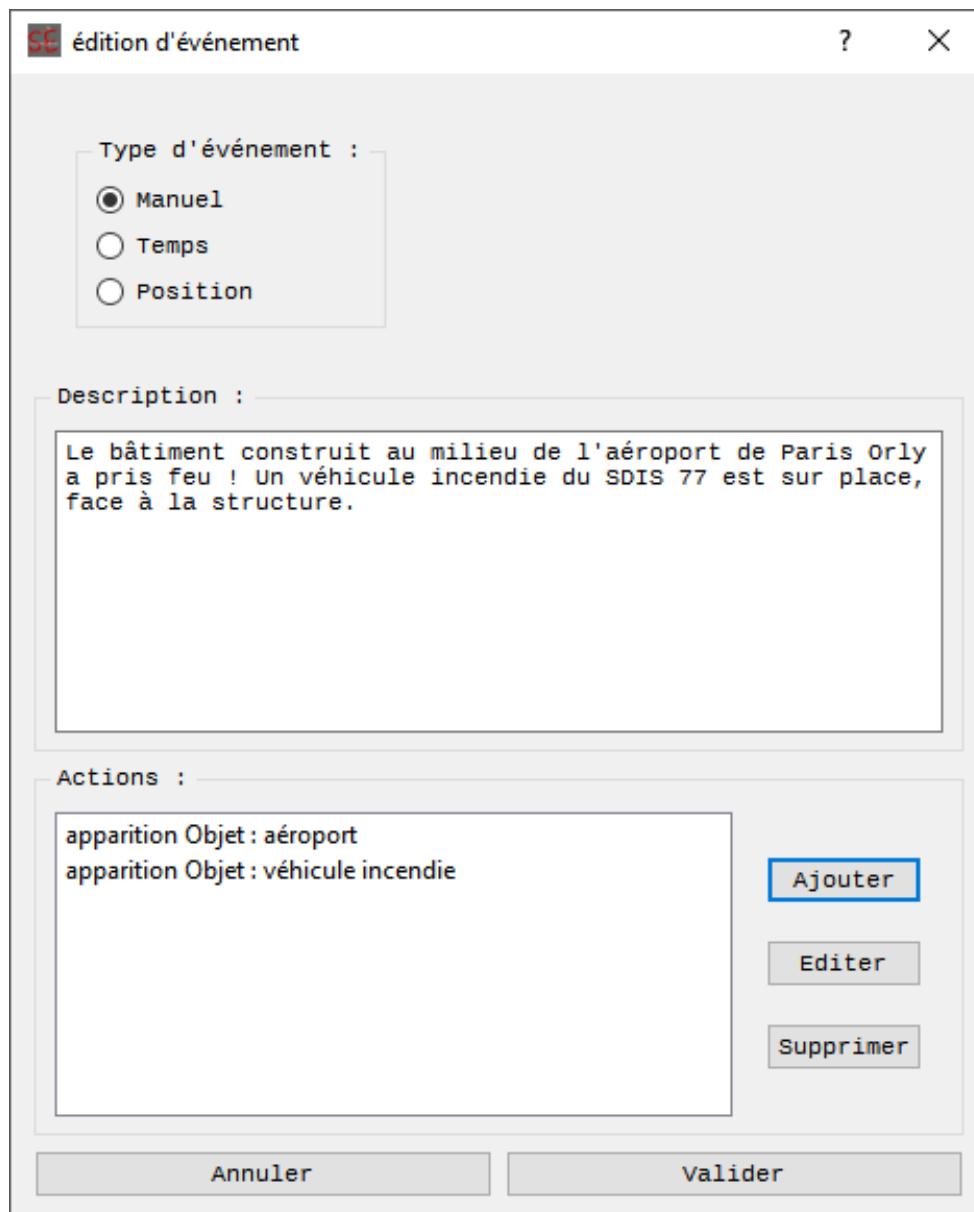
- La visualisation de la liste des objets 3D utilisés par le scénario ;
- La modification du nom ou du fichier d'un objet 3D ;
- L'ajout d'un objet 3D à la liste ;
- La suppression de l'un des objets de la liste ;
- La visualisation de la liste des événements et des actions ;
- L'ajout ou la suppression d'un événement ou d'une action ;
- L'édition d'un événement ou d'une action.

Cette interface est celle qui apparaît au lancement du programme. Elle permet l'affichage de la liste des objets, de la liste des événements ainsi que les actions qu'ils déclenchent. Nous ne nous attarderons pas plus en détails dessus car elle a déjà fait l'objet d'une présentation plus haut dans ce rapport.

IHM d'édition d'objet :



L'IHM d'édition d'objet devait pouvoir permettre l'édition du chemin ou du nom d'un objet 3D. Elle contient donc une ligne pour chacune de ces données, un bouton pour sélectionner un fichier dans l'explorateur de fichiers, un bouton « Annuler » et un bouton « Valider ». Le chemin de l'objet est relatif à l'emplacement de l'exécutable de X-Plane et est utilisé par ScenePlayer.

IHM d'édition d'événement :

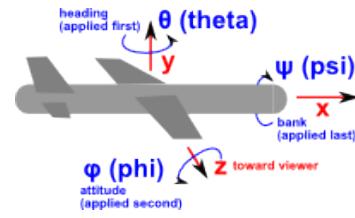
Cette IHM permet d'éditer un événement. Nous pouvons y distinguer trois parties :

- La première partie, en haut, permet d'éditer la condition de déclenchement de l'événement. (ici, déclenchement manuel) ;
- La deuxième partie, au milieu, permet de voir et de modifier la description de l'événement ;
- La troisième partie de cette IHM contient la liste des actions, et permet d'en ajouter, éditer ou supprimer.

IHM d'édition d'action :

SE édition d'action ? X

Type d'action :	Objet :
<input checked="" type="radio"/> Apparition	Nom : aéroport <input type="button" value="..."/>
<input type="radio"/> Déplacement	Identifiant : 1
<input type="radio"/> Disparition	
<input type="radio"/> Changement d'état	
Position :	
Latitude : 48.721001	
Longitude : 2.320000	
<input checked="" type="checkbox"/> Au sol	
Rotation :	
Psi :	0.000000
Theta :	0.000000
Phi :	0.000000
<input checked="" type="checkbox"/> Fumée	
<input type="button" value="Peu de fumée"/> <input type="button" value="Fumée abondante"/>	
<input type="button" value="Annuler"/>	<input type="button" value="Valider"/>



Cette IHM permet l'édition des actions.

Elle permet l'apparition, le déplacement et la disparition d'objets 3D.

L'objet concerné est déterminé par :

- le nom d'objet tel qu'il apparaît dans la liste des objets ;
- un numéro d'identification propre à l'objet.

Il est nécessaire de donner une position pour l'apparition et le déplacement d'objet 3D.

Pour la position, X-Plane utilise les coordonnées réelles : latitude ; longitude ; altitude. À noter qu'il est proposé de placer l'objet au niveau du sol.

Il est possible d'ajouter de la fumée, en spécifiant l'épaisseur de celle-ci grâce à un curseur.

4.1.3. Format des fichiers SCN

Afin de stocker les scénarios, il a été nécessaire de créer un format de fichier utilisé à la fois par les programmes ScenePlayer et SceneEditor.

Ces fichiers doivent contenir tous les objets et événements utilisés par SceneEditor et ScenePlayer. Ils doivent, de plus, être *human readable*, ce qui implique que les informations doivent être contenues sous forme de texte compréhensible.

Ces fichiers ont connu plusieurs évolutions au cours du projet.

Première version de fichier SCN :

La première version des fichiers SCN ressemblait à ceci :

```

object :
    name = obj_test ;
    path = test.obj ;

event :
    name = test 1 ;
    type = manual ;
    active = 1 ;
    action :
        type = spawn ;
        object = obj_test ;
        lat = 0 ;
        lon = 0 ;
    action :
        type = addevent ;
        name = test 2 ;

event :
    name = test 2 ;
    type = time ;
    time = 2 ;
    action :
        type = addevent ;
        name = test 3 ;

event :
    name = test 3 ;
    type = manual ;
    action :
        type = move ;
        object = obj_test ;
        lat = 1 ;
        lon = 1 ;
        orientation = 90 ;

end :

```

Afin de pouvoir lire et écrire ces fichiers, une classe avait été créée. Cependant, le fait que ce système de stockage ne soit pas standard posait quelques soucis, notamment quand il y avait des ajouts à faire. Il était alors nécessaire de modifier en partie le code de cette classe. C'est pourquoi nous avons changé de format et opté pour un format standard.

Version actuelle des fichiers SCN :

Le format actuel des fichiers SCN est un fichier XML.

XML (eXtensible Markup Language) est un langage de balises.

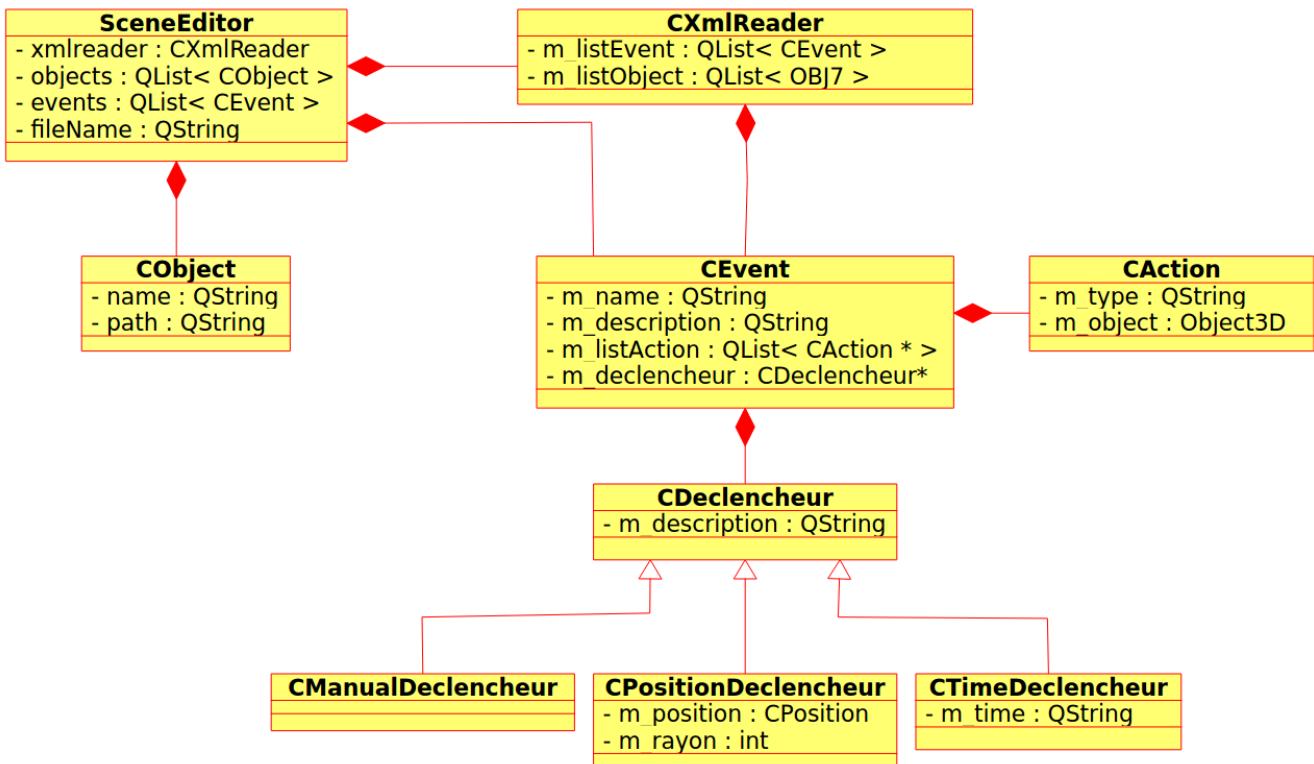
Un fichier XML contient des informations, organisées et encadrées par des balises. Il est possible de définir ses propres balises, ainsi que de leur ajouter des attributs.

Les fichiers actuels ressemblent donc à ceci :

```
<?xml version="1.0" encoding="utf-8"?>
<scnfile>
    <object name="véhicule incendie" path="Custom Scenery/LTF/vehicules/incendie/incendie.obj"/>
    <object name="aéroport" path="Custom Scenery/Aerosoft - LFPO Paris
Orly/Objects/Airport/Buildings15.obj"/>
    <event>
        <name>Déclenchement de l'incendie</name>
        <description>Le bâtiment construit au milieu de l'aéroport de Paris Orly a pris feu ! Un véhicule incendie du SDIS 77 est sur place, face à la structure.</description>
        <actions>
            <action type="apparition">
                <object name="aéroport" id="1">
                    <position latitude="48.721" longitude="2.320" elevation="0"/>
                    <vrotations psi="0" the="0" phi="0"/>
                    <divers ground="1" smoke="50"/>
                </object>
            </action>
            <action type="apparition">
                <object name="véhicule incendie" id="2">
                    <position latitude="48.72" longitude="2.318" elevation="0"/>
                    <vrotations psi="0" the="0" phi="0"/>
                    <divers ground="1" smoke="0"/>
                </object>
            </action>
        </actions>
        <declencheur type="manuel">
            <description>Événement à déclencher manuellement. Début du scénario.</description>
        </declencheur>
    </event>
    <event>
        <name>Incendie maîtrisé</name>
        <description>Le feu est éteint et le véhicule incendie sort de l'aéroport.</description>
        <actions>
            <action type="changement d'état">
                <object name="aéroport" id="1">
                    <position latitude="48.721" longitude="2.320" elevation="0"/>
                    <vrotations psi="0" the="0" phi="0"/>
                    <divers ground="1" smoke="0"/>
                </object>
            </action>
        </actions>
    </event>
</scnfile>
```

```
<action type="déplacement">
    <object name="véhicule incendie" id="2">
        <position latitude="48.72" longitude="2.317" elevation="0"/>
        <vrotations psi="0" the="0" phi="0"/>
        <divers ground="1" smoke="0"/>
    </object>
</action>
</actions>
<declencheur type="manuel">
    <description>événement à déclencher manuellement. Fin du scénario.</description>
</declencheur>
</event>
</scnfile>
```

4.1.4. Diagramme de classes



La classe **SceneEditor** est la classe principale du programme. Elle fait appel à la classe **CXmlReader** pour lire les données contenues dans un fichier scénario. Elle stocke la liste d'objets dans l'attribut **objects** de type **QList< CObject >** et la liste d'événements dans l'attribut **events** de type **QList< CEvent >**.

Chaque objet de type **CEvent** contient un déclencheur, de classe dérivée de **CDeclencheur**, ainsi qu'une liste d'actions de type **QList< CAction >**.

4.1.5. Conclusion

Travailler sur ce projet fut une expérience très enrichissante. En effet, ce projet m'a permis d'acquérir des connaissances relatives au framework Qt, ainsi que quelques connaissances aéronautiques. Ce fut de plus une expérience très enrichissante car ce projet est unique en France.

4.2. Stockage & accès (Franck GAILLARD)

4.2.1. Cahier des charges personnel

Le but de mon projet était de mettre en œuvre les éléments qui vont permettre de stocker et d'accéder aux fichiers scénario. Pour cela je vais avoir besoin de mettre en œuvre un serveur qui contiendra les fichiers scénario. Afin de pouvoir accéder au serveur, il faut un client, qui sera SceneEditor ou ScenePlayer, qui doit être en mesure de pouvoir prendre ou déposer des fichiers sur le serveur. Il faudra donc dans un premier temps développer une classe cliente qui rendra possible les actions demandées puis adapter le code pour convenir aux deux programmes. SceneEditor peut charger et sauvegarder des fichiers depuis/vers le serveur alors que ScenePlayer peut seulement charger ces fichiers.

4.2.2. Moyens mis en œuvre

Le langage dominant de notre projet se trouve être le C++, nous nous sommes donc tous tourné naturellement vers Qt, un logiciel disponible sous Mac, Linux et Windows. De plus, le diagramme de Gantt me proposait d'explorer les classes **QNetworkAccessManager** pour m'aider dans le développement de ma classe.



Le serveur retenu est Core FTP Server, qui est gratuit mais propose une version pro, sous licence. Il est simple à utiliser et à paramétrier même s'il est en anglais. Un autre serveur avait été au départ retenu, FileZilla Server mais il y avait un soucis entre ma classe et celui-ci. FileZilla renvoyait un message d'erreur et bloquait l'envoi de la commande de liste.

4.2.3. Les classes **QNetworkAccessManager**

QNetworkAccessManager :

La classe **QNetworkAccessManager** permet à l'application d'envoyer des requêtes sur le réseau et de recevoir des réponses. Elle contient des configurations et des paramètres qui seront utilisés pour envoyer des requêtes. Un seul **QNetworkAccessManager** suffit pour une application.

→ Voici à quoi ressemble l'initialisation de **QNetworkAccessManager** :

```
|manager = new QNetworkAccessManager ;
```

QNetworkRequest :

La classe **QNetworkRequest** permet l'envoi de requêtes via **QNetworkAccessManager**. Elle contient une **QUrl** et des informations.

→ Voici à quoi ressemble l'envoi d'une requête dans le code :

```
|manager->get(QNetworkRequest(QUrl(*chemin))) ;
```

QNetworkReply :

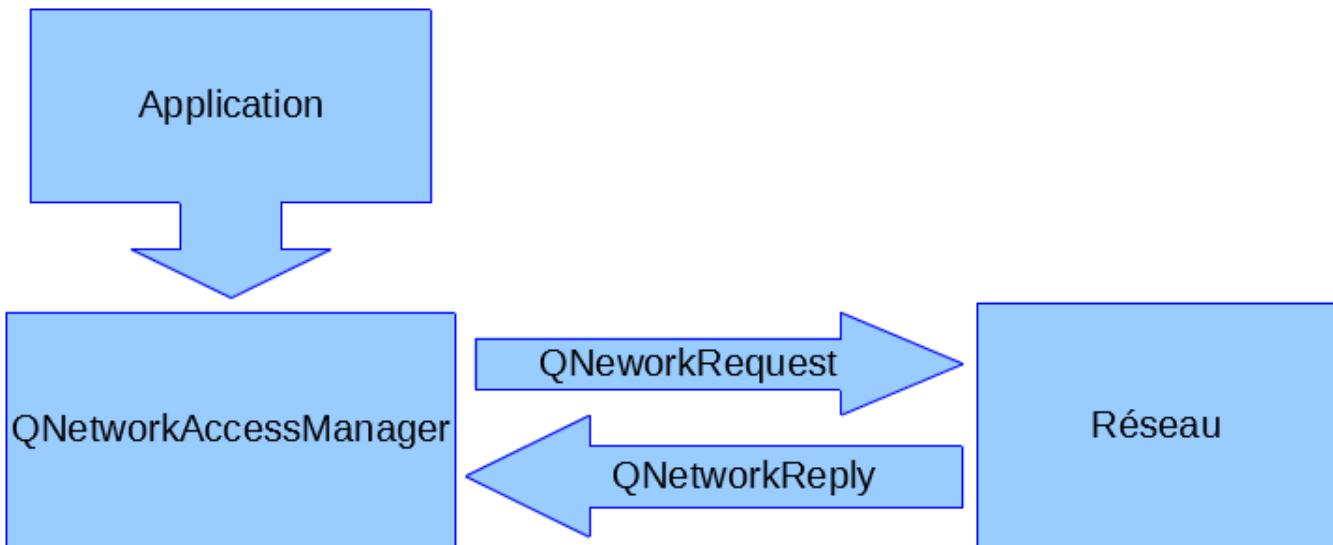
La classe `QNetworkReply` contient les datas et les headers pour une requête envoyée via `QNetworkAccessManager`. On pourrait aussi dire qu'elle est la réponse à cette requête.

→ Voici une récupération des données demandées par la requête :

```
| QNetworkReply *reply = manager->get(request);
```

Schéma des interactions entre les classes QNetwork :

Le schéma ci-dessous récapitule ce qui a été expliqué plus haut.



4.2.4. Protocole FTP et SFTP

Le FTP est un protocole de transfert de fichiers (File Transfer Protocol). Il permet de transférer des fichiers entre un serveur (ici l'AlienWare) et un client (ScenePlayer et SceneEditor). Cependant le principal problème du FTP, est qu'il n'est pas sécurisé. Les données échangées sont visibles et pourraient donc être récupérées par une personne mal intentionnée. De plus les mots de passe sont aussi lisibles ce qui est peut-être dangereux. Pour remédier à cela, il faut utiliser le SFTP, bien plus sûr que le FTP.

Le SFTP (Secure File Transfert Protocol) propose un chiffrement des mots de passe et des fichiers transférés réalisé par le SSH (Secure Shell). SSH s'occupe du chiffrement à l'aide d'une clé. Pour en revenir aux classes Qt, une bonne partie des problèmes de `QFtp`, qui n'existe plus aujourd'hui, ont été résolu par la classe `QNetworkAccessManager` qui gère maintenant seule les différents protocoles.

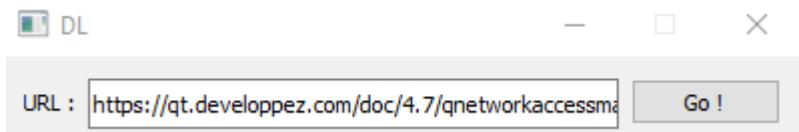
4.2.5. Programmes de test

Afin de comprendre l'utilisation de la classe `QNetworkAccessManager`, j'ai réalisé quelques programmes de test. Les programmes m'ont permis de comprendre comment récupérer des fichiers distants : d'abord sur une page web, puis sur un serveur ; se connecter à un serveur et enfin lister le contenu d'un répertoire distant à l'aide d'un sous-programme situé sur le serveur.

Télécharger le code source d'une page HTML :

Le premier test que j'ai réalisé a été de tenter de récupérer le code source d'une page HTML à l'aide du lien de la page demandée et de sauvegarder son contenu dans un fichier texte. Ensuite j'ai passé le format de sauvegarde au format HTML, ainsi que la possibilité de choisir un emplacement pour l'enregistrement du fichier et le nom du fichier. Enfin, une fois que le fichier est récupéré, il est ouvert automatiquement, avec le programme par défaut de l'ordinateur.

Ci-dessous un exemple du programme de test et ce qu'il récupère :



```

1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
2  <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" lang="fr" dir="ltr" xmlns:og="http://ogp.me/ns#">
3  <head>
4      <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
5      <title>QNetworkAccessManager</title>
6      <meta name="generator" content="developpez.com" />
7      <meta name="description" content="QNetworkAccessManager" />
8      <script type="text/javascript" src="/template/scripts/jquery-1.7.2.js"></script>
9      <script async type="text/javascript" src="https://www.developpez.com/ws/pageview/"></script>
10     <link rel="shortcut icon" type="image/x-icon" href="/favicon.ico" />
11     <link rel="image_src" href="https://www.developpez.com/facebook-icon.png" />
12     <link rel="stylesheet" type="text/css" href="/template/gabarit.css?1483356099" />
13     <link rel="stylesheet" type="text/css" media="print" href="/template/printer.css" />
14     <link rel="stylesheet" type="text/css" media="screen" href="http://www.developpez.com/template/gabarit.css" />
15     <link rel="stylesheet" type="text/css" media="screen" href="http://www.developpez.com/template/gabarit.css" />
16     <!--[if lt IE 7]>
17     <script src="/template/ie7/IE7.js" type="text/javascript"></script>
18     <![endif]-->
19
20     <!--[if IE 6]>
21     <link rel="stylesheet" type="text/css" href="/template/gabarit-ie6.css" />

```

Méthode Login/Logout d'un serveur :

Le but de ce test était de pouvoir se connecter et se déconnecter d'un serveur. Pour se faire, il a suffit simplement de créer une **QUrl** à laquelle on attribue une adresse, un nom d'utilisateur, son mot de passe et le port utilisé.

Première version de listing serveur :

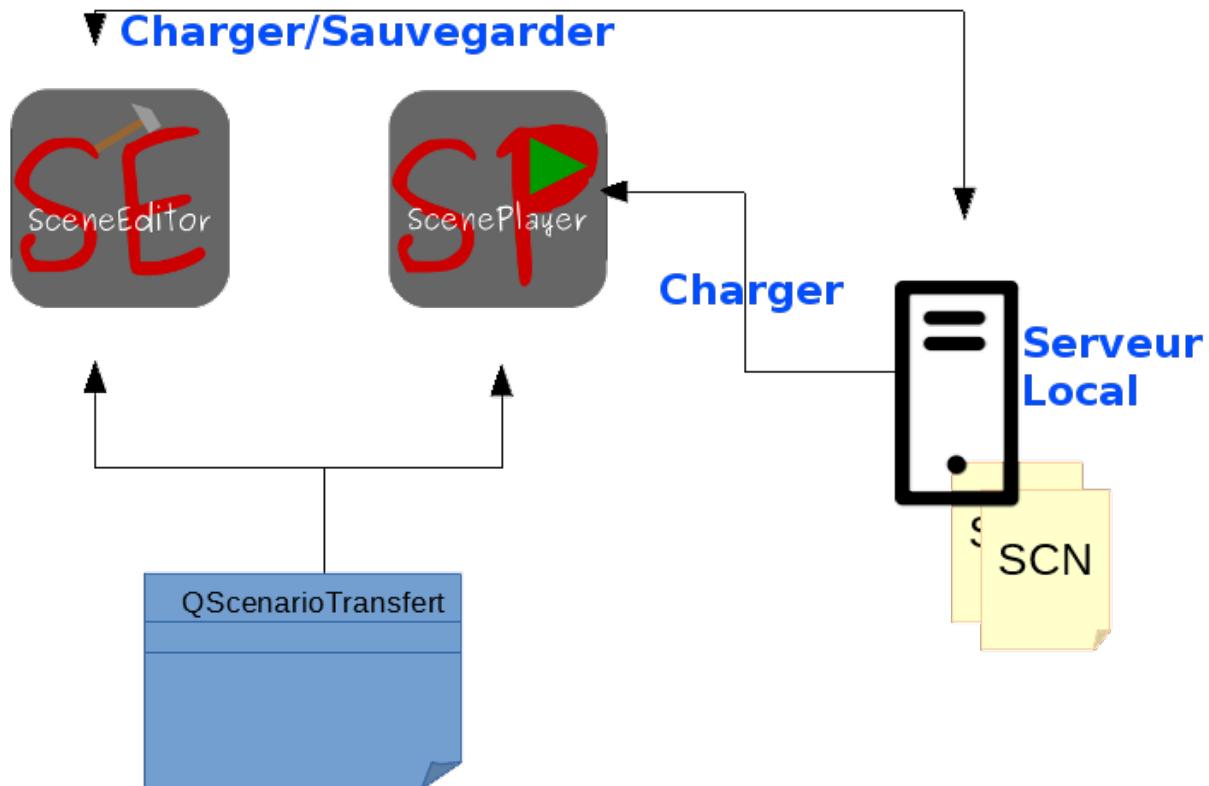
Afin de pouvoir récupérer un fichier sur le serveur, il convient de connaître son emplacement et son nom. J'ai donc réaliser une méthode de listing qui était très similaire à la récupération d'un fichier car celle-ci consistait à récupérer un fichier texte contenant la liste des fichiers et répertoires présent sur le serveur. Ce fichier était généré grâce à un sous-programme situé sur le serveur. La liste une fois récupérée était enregistrée dans un répertoire temporaire, le temps d'être renvoyée dans à la classe cliente puis est effacé.

Cette version possédait beaucoup de problèmes :

- le sous-programme était situé sur le serveur ce qui rend la possibilité d'actualiser la liste impossible. Il aurait fallu créer une tâche exécutant ce programme sur le PC serveur mais l'idée a été rapidement abandonnée ;
- l'affichage renvoyé par la classe était peu lisible, difficile à comprendre et en partie bugué avec des renvois à la ligne pas toujours correctement réalisé.

Il faut donc trouver un autre moyen afin de parvenir à un affichage de listing de dossier correct.

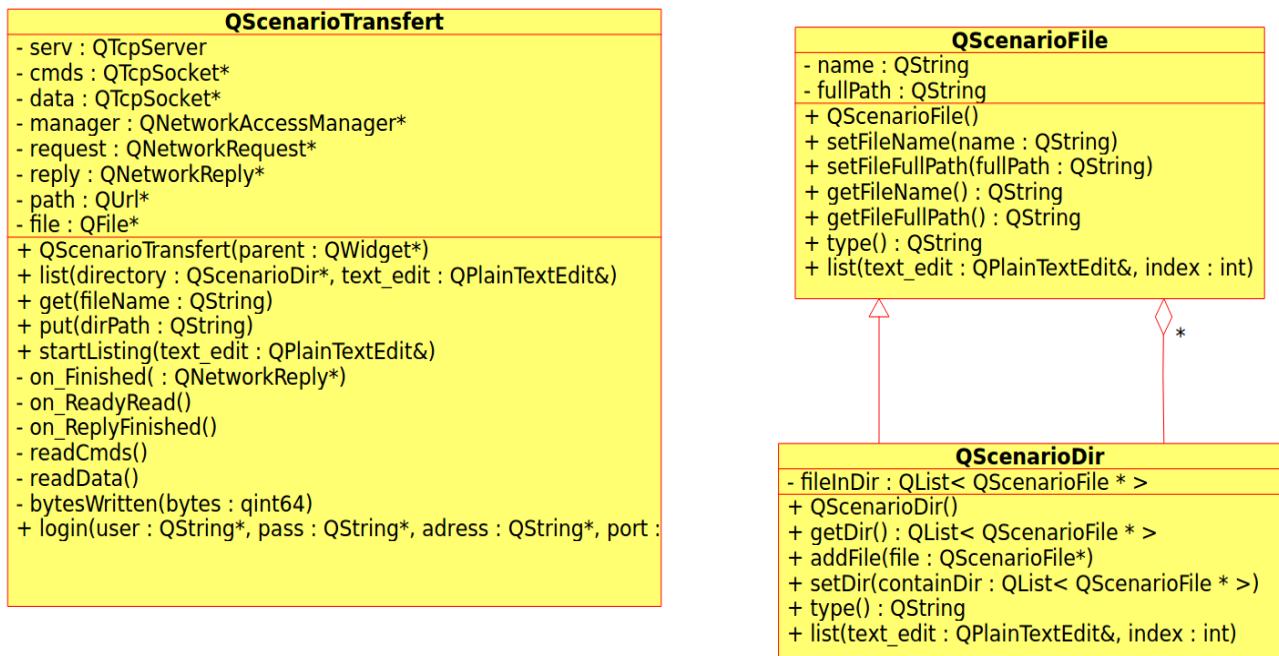
4.2.6. La classe QScenarioTransfert



Après les tests, j'ai réalisé la classe principale du projet, la classe **QScenarioTransfert**. Elle doit être en mesure de pouvoir être utilisée par les programmes ScenePlayer et SceneEditor. Celle-ci doit contenir des méthodes permettant de charger ou sauvegarder un fichier scénario (SCN) du serveur local. La classe sera liée aux programmes via une agrégation.

En outre SceneEditor doit pouvoir utiliser les méthodes permettant de charger et de sauvegarder alors que ScenePlayer aura accès uniquement à la méthode de chargement.

Diagramme de classes :



Les méthodes :

La méthode `login(QString *user, QString *pass, QString *adress, int port)` appelée dans les programmes clients doit être en mesure de récupérer l'identifiant de l'utilisateur, son mot de passe pouvant être géré sur le serveur ainsi que l'adresse et le port du serveur ciblé.

Cette méthode va renvoyer ces attributs dans les autres méthodes afin de pouvoir envoyer des requêtes ou des commandes.

La méthode `get(QString fileName)` permet de télécharger un fichier situé sur un serveur. Le nom ou chemin du fichier est récupéré grâce à son attribut `fileName` qui est ensuite ajouté à la suite de l'adresse.

Fonctionnement :

- La méthode contient le nom du fichier à récupérer et son chemin sur le serveur ;
- Une boite de dialogue par défaut s'ouvre afin de choisir un répertoire de chargement
- La méthode créer un fichier, dans le répertoire choisi, portant le nom du fichier à récupérer ;
- Une `QUrl` est créée contenant une adresse avec l'identifiant et son mot de passe pour définir la requête ;
- Le reste de la `QUrl` est ajouté. Cet ajout contient le nom du fichier demandé et son chemin ;
- La requête est envoyée grâce à `QNetworkAccessManager` et la réponse est attendu ;
- Lorsque la réponse est reçue, le fichier créé est ouvert et le contenu du fichier lu sur le serveur en requête est écrit dans ce fichier.

Exemple d'utilisation dans utilisable dans un programme :

→ Ces deux lignes suivantes permettent d'appeler la méthode `get(fileName)` où `fileName` sera le nom du fichier recherché à ouvrir dans le programme.

`edt_file->text()` permet de passer ce qui est écrit dans une ligne d'édition au format de texte.

```
|QString fileName = edt_file->text();  
|transfert.get(fileName);
```

La méthode `put(QString dirPath)` permet d'uploader un fichier sur le serveur et de le stocker. Le fichier est choisi grâce à une boîte de dialogue par défaut. Le répertoire sur le serveur où le fichier doit être uploader peuvent être spécifiés grâce à l'attribut `dirPath`.

Fonctionnement :

- Une page par défaut est ouverte et permet de sélectionner un fichier à envoyer au serveur ;
- Les informations du fichier à envoyer sont récupérées. Cela permet également de lui mettre un nom ;
- Une `QUrl` est créée contenant une adresse avec l'identifiant et son mot de passe pour définir la requête ;
- Le fichier est lu dans un `QByteArray` qui stock le contenu afin de pouvoir l'envoyer en requête ;
- La requête est envoyée ;
- Le fichier est créé sur le serveur et est ouvert. Le contenu du `QByteArray` est ensuite écrit dans le nouveau fichier ;

```
|QString dirPath = edt_file->text();  
|transfert.put(dirPath);
```

La méthode `startListing(QPlainTextEdit& text_edit)` est la méthode à utiliser dans le programme client afin de préparer et d'exécuter la méthode `list(QScenarioDir *directory, QPlainTextEdit& text_edit)`. Celle-ci s'occupe de régler la liste à l'adresse du serveur en la définissant comme la racine. La méthode `list(QScenarioDir *directory, QPlainTextEdit& text_edit)` est ensuite exécutée et son contenu récupéré grâce à l'utilisation de socket. `QTcpSocket` et `QTcpServer`. La classe `QTcpServer` rend possible l'accès aux connexions entrant. La classe `QTcpSocket` est utilisée pour effectuer du transfert de données.

La méthode `list(QScenarioDir *directory, QPlainTextEdit& text_edit)` permet de récupérer le contenu du répertoire racine du serveur ainsi que tout les dossiers et sous-dossiers du serveur. Celle-ci doit donc être capable de détecter les fichiers et les dossiers et de les différencier. De plus il faut aussi pouvoir lister le contenu des dossiers et des sous-dossiers, c'est pour cela qu'il faut donc faire appel à 2 classes supplémentaires. L'une permet de gérer les fichiers et l'autre qui hérite de la première, qui s'occupe des dossiers.

Les autres classes :

Dans la classe `QScenarioFile`, les méthodes `getFileName()` et `setFileName(QString name)` permettent de récupérer et d'attribuer le nom du fichier scénario dans l'attribut `name`.

Les méthodes `getFullPath()` et `setFullPath(QString fullPath)` permettent de récupérer et d'attribuer le chemin absolu du fichier SCN dans l'attribut `fullPath`.

Dans la classe `QScenarioDir`, les méthodes `getDir()` et `setDir(QList<QScenarioFile*> containDir)` permettent de récupérer et d'attribuer le contenu du répertoire dans l'attribut `fileInDir`. La classe `QScenarioDir` hérite de la classe `QScenarioFile` car un répertoire possède un chemin d'accès et un nom comme un fichier.

Afin de renvoyer la liste, la méthode j'ai fais appel à deux sockets. Une socket est destinée aux commandes, l'autre aux data à récupérer. La socket commandes va donc envoyer directement des commandes FTP (ex : « USER »). Afin d'envoyer cette commande la commande FTP, « LIST », il est nécessaire de spécifier une hôte et son port pour lequel le serveur va devoir se connecter afin de récupérer une réponse. Il faut donc utiliser la commande FTP, « PORT » pour réaliser cela en y ajoutant l'IP ainsi que son nouveau port pouvant être récupéré en envoyant la commande « PASV ». On a par exemple : « PORT 127,0,0,1,4,190 » avec pour syntaxe : `PORT a1,a2,a3,a4,p1,p2` et un port calculé par $p1*256+p2$.

```
cmds->write("USER user\r\n");
cmds->waitForBytesWritten();
cmds->waitForReadyRead();
cmds->write("PASS 123\r\n");
cmds->waitForBytesWritten();
cmds->waitForReadyRead();
cmds->write("PORT 127,0,0,1,4,190");
cmds->waitForBytesWritten();
cmds->waitForReadyRead();
```

Une fois la commande « LIST » envoyée, il est nécessaire d'ajouter un répertoire dans le cas où il y a des sous-dossiers sur le serveur pour qu'ils soient aussi listé. La racine est gérée dans `startListing()` donc pas besoin de faire quoique ce soit ici.

```
QString list_request ;
list_request += "LIST ";
list_request += directory->getFilefullpath();
list_request += "\r\n";
```

La socket data va ensuite prendre le relais afin de pouvoir récupérer les données renvoyées par la commande de liste. Data envoie tout ce qu'elle peut lire dans un `QString` qui contient alors la liste des fichiers avec les différents droits,... et placé en un gros bloc. Afin de les mettre en ligne et donc d'être plus facilement lisible on peut renvoyer à la ligne (Split) à chaque détection de "\r\n" qui symbolise la fin des informations du fichier.

```
//associe la data comme étant la socket des réponses
data = serv.nextPendingConnection();
data->waitForReadyRead();
//récupère le contenu de data dans un QString
QString contenu(data->readAll());
//affiche le contenu ligne par ligne après chaque \r\n
QList<QString> liste_lignes = contenu.split("\r\n", QString::KeepEmptyParts);
```

Ceci fait, il faut maintenant pouvoir savoir si il s'agit d'un fichier ou d'un répertoire. Les dossiers sont symbolisé et commence par un « d » avant les autorisations alors que les fichiers commence par un « - ». Il suffit alors de tester si la ligne commence par un « d » ou un « - ». Si c'est un « - », c'est un

fichier et il faut donc appelé `QScenarioFile` afin lui définir se type en tant que fichier. Si c'est un « d » c'est un répertoire et il faut donc appelé `QScenarioDir` afin de lui définir le type en tant que répertoire et ensuite renvoyer une liste avec cette fois si ce répertoire comme cible.

Pour terminer, il convient mieux de retirer la partie des droits sur les fichier et de garder que le nom des fichiers pour être plus visible.

```
//test permettant de vérifier si la ligne de contenu concerne un fichier ou un répertoire
for(int i=0;i<liste_lignes.size();i++)
{
    if(liste_lignes[i].startsWith("-")) //qscenariofile
    {
        //gère le fichier et ajoute son chemin
        qDebug() << " Fichier ";
        QScenarioFile *scnFile = new QScenarioFile;
        liste_lignes[i].remove(0,59); /*retire les données avant le nom du fichier,
supprime donc les 59 caractères au départ de chaque ligne*/
        scnFile->setFilename(liste_lignes[i]);
        QString path = directory->getfilepath();
        path += liste_lignes[i];
        scnFile->setFullPath(path);
        directory->addFile(scnFile);
    }
    else if(liste_lignes[i].startsWith("d")) //qscenariodir
    {
        /*ajoute le dossier en à LIST en renvoi un listing de celui-ci afin d'avoir les
fichiers contenus dans un dossier*/
        qDebug() << " Dossier ";
        QScenarioDir *scnDir = new QScenarioDir();
        liste_lignes[i].remove(0,59);
        scnDir->setFilename(liste_lignes[i]);
        QString path = directory->getfilepath();
        path += liste_lignes[i];
        path += "/";
        scnDir->setFullPath(path);
        directory->addFile(scnDir);
        list(scnDir, text_edit);
    }
}
```

4.2.7. Problèmes

Le premier problème est un problème particulier qui donne la possibilité de récupérer un fichier dans UN sous-répertoire mais pas au travers de deux.

Téléchargement OK : <ftp://adresse/Dossier1/fichier.scn>

Le fichier est bien créé et son contenu est récupéré avec succès.

Ne télécharge pas : <ftp://adresse/Dossier1/Dossier2/fichier.scn>

Le fichier est bien trouvé sur le serveur et est créé mais son contenu n'est pas récupéré.

L'utilisation de la méthode `waitForReadyRead()` de `QAbstractSocket` peut provoquer des erreurs de façon aléatoire, selon la documentation de Qt. Cette méthode est utilisée à plusieurs reprise dans méthode `list()`. De plus, sous Linux, lorsque l'on appelle la méthode `list()` dans un programme, celui-ci renvoi un SEGMENTATION FAULT.

Ce problème est probablement la raison pour laquelle la liste n'est pas toujours récupérée totalement et fait en plus stopper le programme sur un poste sous Linux.

Ci-dessous 2 captures d'écrans. Celle de gauche montre le répertoire du serveur et celle de droite la liste récupérée dans un programme client.

Nom	Modifié le	Type	Taille
fileindexer_history	22/03/2017 10:18	Dossier de fichiers	
items	17/05/2017 12:14	Dossier de fichiers	
scn_old	17/05/2017 12:13	Dossier de fichiers	
ASP.net	28/02/2017 13:39	Dessin OpenDocu...	9 Ko
config.dat	19/04/2017 12:13	Fichier DAT	2 Ko
fileindexer	22/03/2017 10:13	Application	20 Ko
list	22/03/2017 10:18	Fichier	1 Ko
liste	21/03/2017 11:37	Fichier TXT	1 Ko
scn_aa	19/04/2017 10:15	Fichier SCN	1 Ko
scn_ab	21/04/2017 09:26	Fichier SCN	1 Ko
scn_bb	05/05/2017 15:00	Fichier SCN	1 Ko
scn_cc	05/05/2017 15:01	Fichier SCN	1 Ko
scn_da	17/03/2017 09:56	Fichier SCN	1 Ko
scn_dd	05/05/2017 15:00	Fichier SCN	1 Ko
scn_ee	05/05/2017 16:57	Fichier SCN	1 Ko
scn_na	05/05/2017 16:46	Fichier SCN	1 Ko
scn_obj	05/05/2017 14:48	Fichier SCN	2 Ko
test_scn	07/03/2017 14:59	Fichier SCN	1 Ko

ASP.net.odg
config.dat
fileindexer.exe
fileindexer_history
fileindexer.exe
list
items
list
panneau.txt
personnage.scn
vehicules
camion.txt
list
list.txt
scn_aa.scn
scn_ab.scn
scn_bb.scn
scn_cc.scn
scn_da.scn

On peut constater que la liste n'a pas été récupérée complètement et qu'il manque 5 fichiers à afficher dans la liste.

Il n'y a d'ailleurs pas vraiment de gestion des erreurs, l'utilisation de cette classe peut facilement faire planter les programmes, SceneEditor et ScenePlayer. La classe demande donc des optimisations.

4.2.8. Conclusion

Pour conclure, ce projet a été vraiment intéressant d'une part pour découvrir des méthodes de travail particulière et d'autre part réaliser un projet pour la sécurité civile.

De plus j'ai pu apprendre de nouvelles compétences et en améliorer certaines. J'ai, malgré un limite hors sujet à propos de faire un client plutôt qu'une classe, tout même réussi à rattraper mon erreur.

Le projet m'a plu malgré les erreurs et les difficultés rencontrées.

4.3. ScenePlayer (Quentin DELION)

4.3.1. Cahier des charges



Mon objectif dans ce projet est de créer une application pour pouvoir jouer un scénario. Il doit donc pouvoir charger un fichier SCN en lecture seule depuis le serveur FTP hébergé sur la machine AlienWare. L'instructeur peut donc voir la liste des événements ainsi que les actions et le déclencheur qui y sont associés. L'application doit ensuite modifier la scène 3D de X-Plane pour correspondre aux situations qui sont présentées dans le fichier SCN. ScenePlayer doit aussi être compatible sur tablette Android et/ou iPad pour ne pas entraver la mobilité de l'instructeur.

Cahier des charges de ScenePlayer :

- créer une application capable de lire et jouer un fichier scénario ;
- capable de charger un fichier en lecture seule, depuis le serveur FTP ;
- capable de modifier la scène 3D d'X-Plane ;
- être compatible tablette Android et/ou iPad.

4.3.2. L'Interface Homme-Machine

ScenePlayer est le logiciel qui permet de jouer un scénario. Elle sera compatible sur tablette pour ne pas entraver la mobilité de l'instructeur. L'application doit charger la liste des fichiers depuis le serveur de stockage grâce à la classe `QScenarioTransfert`. On obtient la liste des fichiers SCN disponibles. Une fois qu'on en a chargé un, le tableau affiche la liste des événements correspondant au scénario.

Action	Déclencheur
Déclenchement de l'incendie	
Le bâtiment construit au milieu de l'aéroport de Paris Orly a pris feu ! Un véhicule incendie du SDIS 77 est sur place, face à la structure. Type : apparition Objet : aéroport Latitude : 48.721 Longitude : 2.32 Élévation : 0 Psi : 0 The : 0 Phi : 0 Sur le sol : Oui Taille de la fumée : 50	Événement à déclencher manuellement. Début du scénario. Type : déclenchement manuel

Aperçu sous Linux

Pour chaque événements, on a la liste des actions affichée à gauche. Elle permet de savoir :

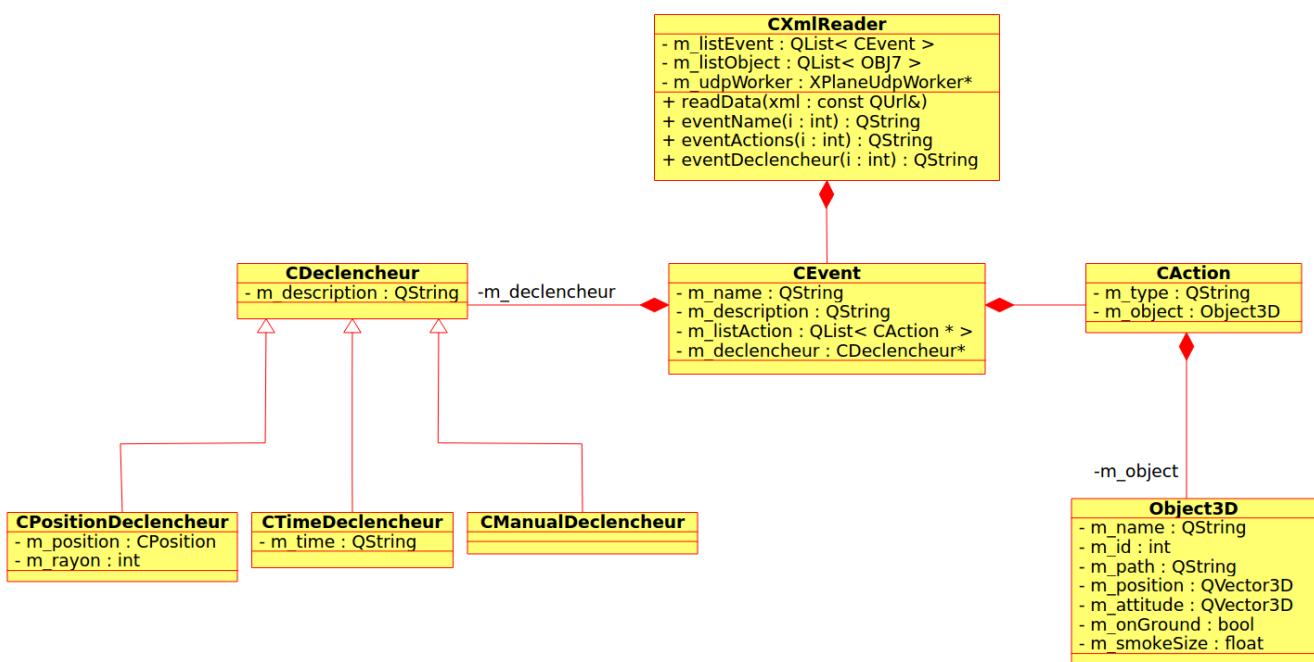
- de quel type est l'action (apparition d'objet, déplacement, disparition, changement d'état) ;
- quel est l'objet concerné par cette action ;
- les coordonnées qu'aura l'objet à la fin de l'action ;
- d'autres paramètres comme la présence de fumée ou si l'objet est placé sur le sol.

À droite de la liste des actions, on affiche le déclencheur. On a comme informations :

- la description du déclencheur pour préciser un contexte par exemple ;
- le type de déclencheur (manuel, en fonction du temps, en fonction de la position de l'hélicoptère).

L'événement possède aussi un titre et une description qui permet de spécifier le contexte de cet événement dans le scénario.

4.3.3. Diagramme de classes simplifié



La classe **CXmlReader** permet de parcourir le fichier SCN sélectionné. Il stocke ensuite dans une liste de **CEvent** les différents événements. Les actions et le déclencheur de l'événement sont respectivement stockés dans une liste de **CAction** et une des classes dérivées de **CDeclencheur** (tout dépend du type de déclencheur).

L'attribut **m_object** de la classe **CAction** est de type **Object3D**. Cette classe contient toutes les informations pour représenter cet objet :

- **m_name** est le nom de l'objet. Il n'est pas utilisé dans les requêtes pour charger ou placer un objet dans X-Plane. Il sert à récupérer le chemin de l'objet du même nom indiqué dans le fichier SCN ;

- `m_id` est l'identifiant que X-Plane utilise pour savoir quel est l'objet manipulé ;
- `m_path` contient le chemin d'accès relatif du fichier OBJ depuis l'exécutable de X-Plane ;
- `m_position` contient les coordonées pour placer l'objet (latitude, longitude et élévation) ;
- `m_attitude` contient les vitesses de rotation autour des trois axes ;
- `m_onGround` est un attribut booléen. S'il vaut true, l'objet sera placé sur le sol, peu importe la valeur d'élévation contenu dans `m_position` ;
- `m_smokeSize` est l'attribut correspondant à l'épaisseur de la fumée. Si la valeur est 0, il n'y aura pas de fumée.

4.3.4. Présentation de Qt/QML

Pour créer l'IHM de ScenePlayer, j'utilise le langage QML proposé par Qt. QML permet de créer des interfaces homme-machine performantes avec une syntaxe de programmation relativement simple. Ce langage a aussi l'avantage de pouvoir intégrer des expressions en JavaScript qui peuvent être liées à des propriétés de mise en forme. Qt Quick est la librairie standard des types et des fonctionnalités de QML comme les type `ListView` et `ListModel` utilisé dans ce programme.

4.3.5. Fonctionnement

Lors de l'appui du bouton « Charger », on fait appel à la fonction `loadScn(txtSCN.text)` codé en javascript. `txtScn.text` contient le texte présent dans le champ d'édition (le chemin d'accès au fichier SCN).

```
Button {
    id: pbuOK
    text: qsTr("Charger")
    enabled: false
    onClicked: {
        loadScn(txtSCN.text);
        enabled = false;
        pbuRestart.enabled = false;
    }
}
```

Cette fonction appelle à son tour des méthodes de la classe `CXmlReader`.

```
WorkerScript {
    id: worker
    source: "dataloader.js"
}

function loadScn(file)
{
    xmlreader.readData(file) ;
    var restart = {
        'action': 'restartScn',
        'model': scnModel
    };
    worker.sendMessage(restart) ;
    var i = 0 ;
```

```

for (i = 0 ; i < xmlreader.numberEvents() ; i++) {
    var msg = {
        'action': 'addEvent',
        'model': scnModel,
        'name': xmlreader.eventName(i),
        'actions': xmlreader.eventActions(i),
        'declencheur': xmlreader.eventDeclencheur(i),
        'index': i
    };
    worker.sendMessage(msg) ;
}
}

```

La méthode `void readData(const QUrl& xml)` de la classe `CXmlReader` permet d'ajouter les événements du fichier dans `m_listEvent`. Comme les fichiers SCN sont écrit en XML, on utilise les classes proposées par Qt pour parcourir et récupérer les valeurs des différents nœuds.

Voici un extrait du fichier SCN de la fonction pour ajouter une action dans l'événement qui le contient.

Fichier SCN :

```

<?xml version="1.0" encoding="utf-8" ?>
<scnfile>
    <object name="véhicule incendie" path="Custom Scenery/LTF/vehicules/incendie/incendie.obj"/>
    <object name="aéroport" path="Custom Scenery/Aerosoft - LFPO Paris
Orly/Objects/Airport/Buildings15.obj"/>
    <event>
        <name>Déclenchement de l'incendie</name>
        <description>Le bâtiment construit au milieu de l'aéroport de Paris Orly a pris feu ! Un véhicule
incendie du SDIS 77 est sur place, face à la structure.</description>
        <actions>
            <action type="apparition">
                <object name="aéroport" id="1">
                    <position latitude="48.721" longitude="2.320" elevation="0"/>
                    <vrotations psi="0" the="0" phi="0"/>
                    <divers ground="1" smoke="50"/>
                </object>
            </action>
            <action type="apparition">
                ...
            </action>
        </actions>
        <declencheur type="manuel">
            <description>Événement à déclencher manuellement. Début du scénario.</description>
        </declencheur>
    </event>

```

Fonction `readData()` :

```

void CXmlReader::readData(const QUrl& xml)
{
    // extrait pour ajouter une action
    if (item.tagName() == "action") {
        // récupère le type (apparition, disparition, ...)
        QString type_action = item.attribute("type");
        QDomNodeList actionNodes = n.childNodes();
        // parcours les nœuds enfants
        for (int i = 0 ; i < actionNodes.size() ; ++i) {
            QDomNode n = actionNodes.item(i);
            QDomElement item = n.toElement();
            // détection de l'objet
            if (item.tagName() == "object") {
                int obj_id = item.attribute("id").toInt();
                QString obj_path = OBJ7nameToObject3Dpath(item.attribute("name"));
                Object3D obj(obj_id, obj_path, item.attribute("name"));
                QDomNodeList actionNodes = n.childNodes();
                for (int i = 0 ; i < actionNodes.size() ; ++i) {
                    QDomNode n = actionNodes.item(i);
                    QDomElement item = n.toElement();
                    if (item.tagName() == "position") {
                        obj.setPosition(item.attribute("latitude").toDouble(),
                            item.attribute("longitude").toDouble(),
                            item.attribute("elevation").toDouble());
                    }
                    else if (item.tagName() == "vrotations") {
                        obj.setAttitude(item.attribute("psi").toFloat(),
                            item.attribute("the").toFloat(),
                            item.attribute("phi").toFloat());
                    }
                    else if (item.tagName() == "divers") {
                        obj.setOnGround(item.attribute("ground").toInt());
                        obj.setSmokeSize(item.attribute("smoke").toFloat());
                    }
                }
                m_listEvent.last().addAction(new CAction(type_action, obj));
            }
        }
    }
}

```

L'affichage est réalisé avec une `ListModel`. Chaque `ListElement` représente un événement.

```

ListModel {
    id: scnModel
    // Création d'au moins un ListElement pour définir des attributs. La valeur donnée
    // est juste en exemple
    ListElement {
        name : "Nom événement E1"
    }
}

```

```

        action : "Description E1\nType : type A1\nParamètre : paramètre A1"
        déclencheur : "Description D1\nType : type D1\nParamètre : paramètre D1"
    }
}

```

Le contenu de chaque attribut d'un **ListElement** est fait avec le **WorkerScript**.

```

WorkerScript.onMessage = function(msg) {
    if (msg.action == 'addEvent') {
        var data = {
            'name': msg.name,
            'action': msg.actions,
            'déclencheur': msg.déclencheur
        };
        msg.model.insert(msg.index, data);
        msg.model.sync(); // updates the changes to the list
    }
}

```

La liste est affiché sans mise en forme particulière. Pour y remédier, on utilise une **ListView**. Elle permet de séparer la partie « données » de la partie « mise en forme ».

```

ListView {
    id: scnXmlView
    header: scnHeader
    headerPositioning: ListView.OverlayHeader
    model: scnModel
    delegate: scnXmlDelegate
    highlight: Rectangle { color: "lightsteelblue"; radius: 5 }
}

```

header permet d'utiliser un **Component** pour créer un en-tête de liste. Cet en-tête restera affiché, même lors du défilement de la liste, grâce à l'attribut **headerPositioning**. L'attribut **model** contient la **ListModel** (vu précédemment) et l'attribut **delegate** permet, comme pour **header**, d'utiliser un **Component** pour la mise en forme des données de la liste.

Ci-dessous, le code pour la mise en forme des données de l'événement.

```

Component {
    id: scnXmlDelegate
    Row {
        spacing: 20
        id: element
        Column {
            width: main.width
            // mise en forme pour le nom de l'événement
            Text {

```

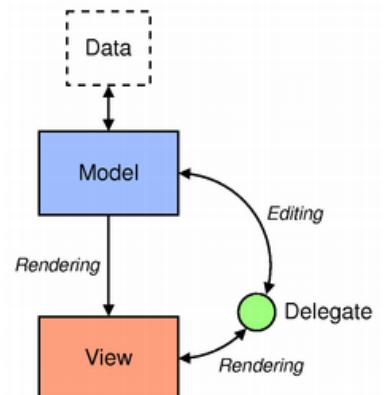


Schéma issu de la doc de Qt

```
        width: parent.width
        text: name
        padding: 5
        font.bold: true
        horizontalAlignment: Text.AlignHCenter
        verticalAlignment: Text.AlignVCenter
    }
Row {
    Column {
        width: main.width / 2
        // Mise en forme pour la liste d'actions
        Text {
            width: parent.width
            text: action
            padding: font.pointSize
            horizontalAlignment: Text.AlignHCenter
            verticalAlignment: Text.AlignVCenter
            wrapMode: Text.WordWrap
            elide: Text.ElideRight
        }
    }
    Column {
        width: main.width / 2
        // Mise en forme por le declencheur
        Text {
            text: declencheur
            // ---
            // Mise en forme identique à la liste d'actions
            // ---
        }
    }
}
}
```

4.3.6. Comparaison entre un fichier SCN et le résultat dans ScenePlayer

Cette comparaison sera faite uniquement sur le premier événement, chaque événement étant composé de façon identique.

The image shows a comparison between a SCN XML file and its visualization in the ScenePlayer application.

ScenePlayer Interface:

- Fichier SCN :** file:///home/fournier17/TSSMIR/Projet/SCN/V02/obj.scn
- Action:** Déclenchement de l'incendie
- Description:** Le bâtiment construit au milieu de l'aéroport de Paris Orly a pris feu ! Un véhicule incendié du SDS 77 est sur place, face à la structure.
- Déclencheur:** Événement à déclencher manuellement. Début du scénario.
- Details:**
 - Type : déclenchement manuel
 - Type : apparition
 - Objet : aéroport
 - Latitude : 48.721
 - Longitude : 2.32
 - Elevation : 0
 - Psi : 0
 - The : 0
 - Phi : 0
 - Sur le sol : Oui
 - Taille de la fumée : 50
 - Type : apparition
 - Objet : véhicule incendie
 - Latitude : 48.72
 - Longitude : 2.318
 - Elevation : 0
 - Psi : 0
 - The : 0
 - Phi : 0
 - Sur le sol : Oui
 - Taille de la fumée : 0
- Buttons:** Quitter, Recommencer, Déclencher, Charger

SCN XML File Content:

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <scnfile>
3   <object name="véhicule incendie" path="Custom Scenery/LTTF/véhicules/incendie/incendie.obj" />
4   <object name="aéroport" path="Custom Scenery/Aéroport - LFOO Paris Orly/objects/Airport /Buildings15,obj" />
5 <event>
6   <description>Déclenchement de l'incendie</description>
7   <action type="apparition">
8     <object name="aéroport" id="1">
9       <position latitude="48.721" longitude="2.320" elevation="0"/>
10      <rotations psi="0" the="0" phi="0"/>
11      <divers ground="1" smoke="50"/>
12    </object>
13  </action>
14  <action type="apparition">
15    <object name="véhicule incendié" id="2">
16      <position latitude="48.72" longitude="2.318" elevation="0"/>
17      <rotations psi="0" the="0" phi="0"/>
18      <divers ground="1" smoke="0"/>
19    </object>
20  </action>
21 </events>
22 <declencheur type="manuel" >
23   <description>Événement à déclencher manuellement. Début du scénario.</description>
24 </declencheur>
25 <declencheur type="manuel" >
26   <description>Événement à déclencher manuellement. Début du scénario.</description>
27 </declencheur>

```

4.3.7. Conclusion

J'ai beaucoup apprécié ce projet. C'est différent des TP effectués en classe. On travaille sur un même projet pendant environ quatre mois et non une ou deux semaines. Ce travail réalisé n'a pas servi uniquement à m'apprendre ou m'améliorer en programmation, il servira à piloter un scénario pour le simulateur.

4.4. Liaison UDP (*Gregory GAILLARD*)

4.4.1. Mon équipe

Je suis un des quatre membres du projet 1790-5, cela signifie que nous sommes la cinquième équipe en deux ans à travailler sur le projet qui a pour but de servir le SDIS. Cette cinquième branche a pour but de créer, stocker, jouer et envoyer des scénarios d'intervention sur le simulateur X-Plane.

Dans ce projet je suis chargé de la liaison UDP entre X-Plane et ScenePlayer. C'est-à-dire que mon rôle est d'envoyer tout ce dont a besoin ScenePlayer pour lancer ses scènes. Je dois envoyer les objets 3D, placer les bâtiments ou autres, déclencher un incendie s'ils doivent brûler, mettre les accessoires mais aussi orienter les objets et les faire se déplacer.

4.4.2. Mon cahier des charges :

Pour cette première phase d'étude, on étudiera notamment les possibilités offertes par X Plane via les requêtes UDP de type RPOS (position instantanée de l'aéronef), et OBN/OBJL qui permettent de manipuler des objets 3D au format spécifique reconnu par X-Plane (OBJ7) ; sachant que OBJL autorise aussi l'apparition de fumée.

(Le développement et la mise en place de plugins communicants au sein de X-Plane est une autre piste à explorer...).

4.4.3. Requête OBN/OBJL et RPOS

OBN, OBject Name, est une méthode permettant d'aller chercher des objets dans un dossier défini préalablement. Si cette méthode est appelée de la sorte c'est à cause de la trame qui s'occupe, elle, d'envoyer les objets à destination. L'OBN contient le chemin et l'ID de l'objet à envoyer lors du scénario.

id="1"

```
path="Custom Scenery/LTF/vehicules/incendie/incendie.obj"/>
path="Custom Scenery/Aerosoft - LFPO Paris Orly/Objects/Airport/Buildings15.obj"/>
```

```
QByteArray Object3D::xplaneObjectLoadRequest() const
{
    QByteArray frame("OBN", 4) ;
    frame.append( char(0) ) ;
    frame.append((char*)(&m_id), 4) ;
    frame.append( m_path ) ;
    frame.append( QByteArray(500 - m_path.size(), 0) ) ;
    return frame ;
}
```

0	1	3	3	4	5...8	9...508
'O'	'B'	'J'	'N'	'0'	ID	path

OBJL, OBJect Location, est une méthode permettant de placer un objet dans X-Plane, grâce à des coordonnées définies. Si cette méthode est nommée comme ceci c'est à cause de la trame du même nom qui, elle, oriente et positionne les objets. L'OBJL contient l'ID, les coordonnées GPS (Longitude, Latitude, Altitude ainsi que la vitesse de rotation). Il y a aussi la possibilité de rajouter de la fumée ou du brouillard ainsi que le réglage de leur épaisseur.

```

id="1"
<position latitude="48.72" longitude="2.318" elevation="0"/>
<vrotations psi="0" the="0" phi="0"/>
<divers ground="1" smoke="0"/>
```

```

QByteArray Object3D::xplaneObjectPlaceRequest() const
{
    QByteArray frame("OBJL", 4) ;
    frame.append( char(0) ) ;
    XPlaneObjPlace data ;
    data.index = (int)m_id ;
    data.lat_lon_ele[0] = (double)( m_position.x() ) ;
    data.lat_lon_ele[1] = (double)( m_position.y() ) ;
    data.lat_lon_ele[2] = (double)( m_position.z() ) ;
    data.psi_the_phi[0] = (float)( m_attitude.x() ) ;
    data.psi_the_phi[1] = (float)( m_attitude.y() ) ;
    data.psi_the_phi[2] = (float)( m_attitude.z() ) ;
    data.on_ground = (int)m_onGround ;
    data.smoke_size = (float)m_smokeSize ;
    frame.append((char*)(&data), sizeof(XPlaneObjPlace)) ;
    return frame ;
}
```

0	1	3	3	4	5...12
'O'	'B'	'J'	'L'	'0'	ID

13...20	21...28	29...36	37...40	41...44	45...48	49...52	53...60
latitude	longitude	élévation	psi	the	phi	ground	smoke

RPOS, Request POSition, sont des requêtes UDP envoyant les coordonnées GPS des objets à placer (Latitude, Longitude, Altitude).

0	1	2	3	4	5 .. 12	13 .. 20	21 .. 24	25 .. 28
'R'	'P'	'O'	'S'	0	longitude	latitude	elevation_MSL	elevation_AGL
29 .. 32	33 .. 36	37 .. 40	41 .. 44	45 .. 48	49 .. 52	53 .. 56	57 .. 60	61 .. 64
pitch	heading	roll	Vx	Vy	Vz	Pdeg	Qdeg	Rdeg

Ici nous pouvons voir que la trame, de 65 octets, contient les coordonnées GPS, Longitude (sur 7 octets), Latitude (sur 7 octets), Altitude (élevation_MSL, élévation_AGL, sur 2 fois 7 octets). Nous observons aussi les variations des axes (pitch, heading, roll, sur 3 octets chacun). Puis nous voyons les vitesses de déplacement sur chaque axe (Vx, Vy, Vz, sur 3 octets aussi). Enfin nous avons les degrés de rotation pour chacun des axes (Pdeg, Qdeg, Rdeg, sur 3 octets eux aussi).

Les requêtes « rpos_rfreq » constituent la fréquence, par seconde, d'envois des trames choisie par l'utilisateur.

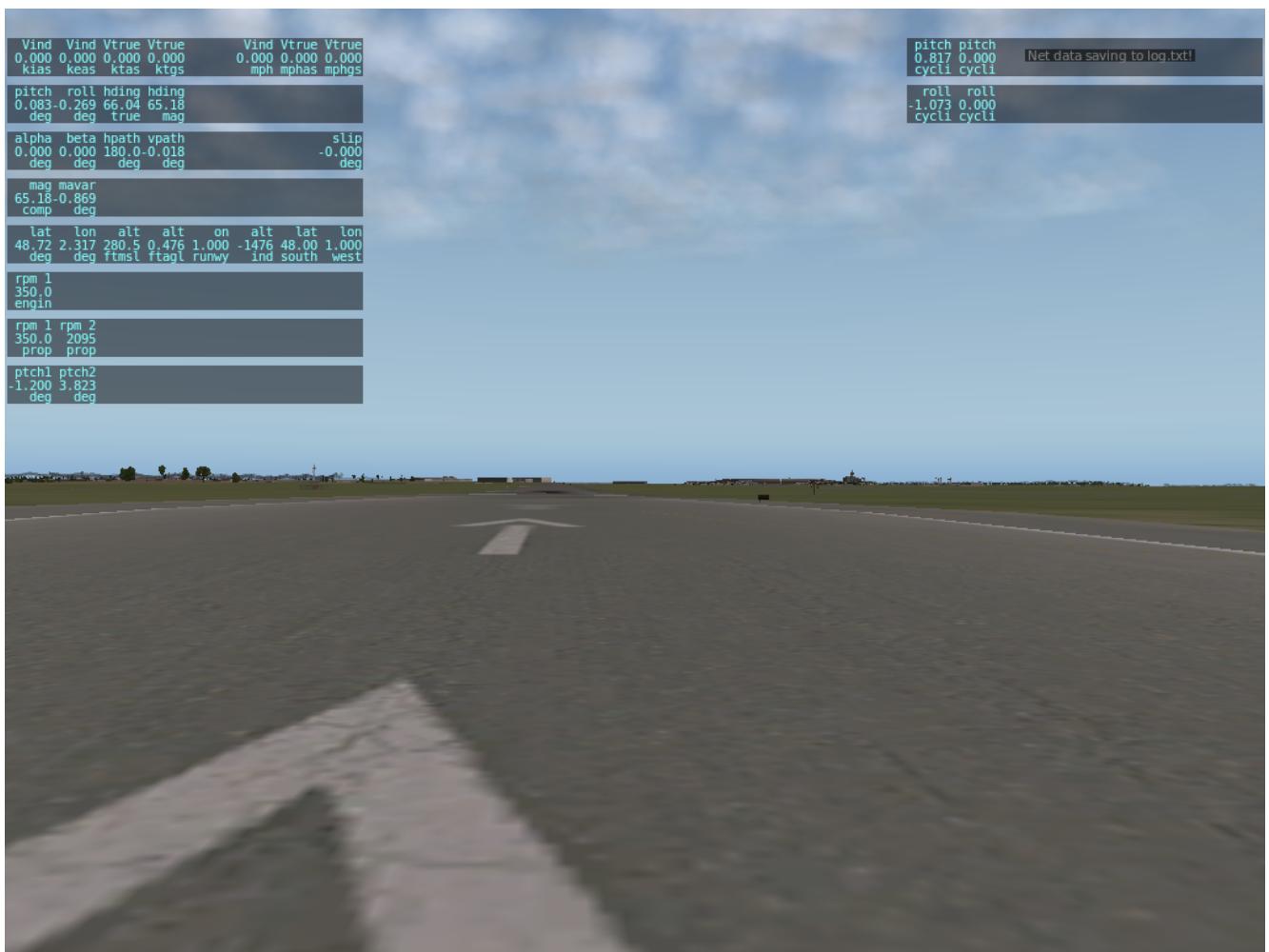
0 'R'	1 'P'	2 'O'	3 'S'	4 0	5	6 freq (int)	7	8
----------	----------	----------	----------	--------	---	-----------------	---	---

Ici nous pouvons voir que la trame, de 9 octets, contient la fréquence, sur 4 octets.

4.4.4. Démonstration de l'action de ScenePlayer dans X-Plane

La scène du fichier scénario utilisé est située à l'aéroport Paris Orly. Ce scénario contient deux événements. Le premier est un bâtiment situé sur la piste d'envol qui prend feu. Un véhicule du SDIS est face à la strucutre. Le deuxième événement est l'extinction de l'incendie et le départ du véhicule.

Avant le lancement du scénario, l'hélicoptère est situé sur la piste. L'image ci-dessous est l'état de la scène avant le début du scénario.



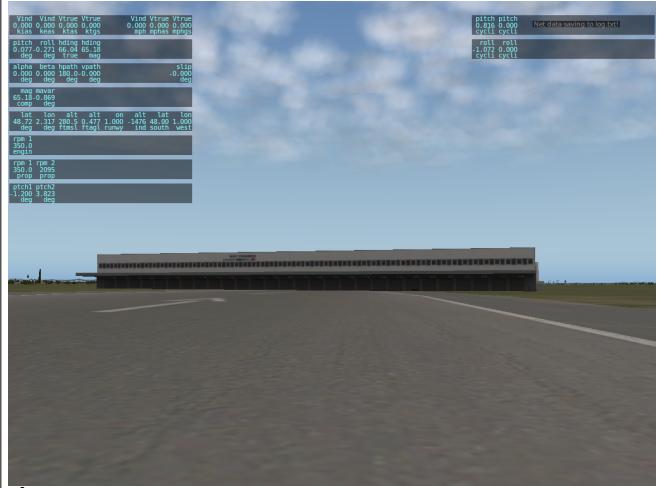
Situation initiale avant le début du scénario.

On constate que le bâtiment qui prendra feu dans le prochain événement n'est pas placé. En effet, l'objet n'est créé qu'à partir de l'événement 1 (de même pour le véhicule).

Les déclencheurs des événements sont tous les deux manuels. Les actions sont réalisées lors de l'appui sur le bouton « Déclencher ». Le tableau ci-dessous montre ce qu'on obtient dans X-Plane après chaque événement.



Événement 1 : Déclenchement de l'incendie



Événement 2 : Extinction de l'incendie

ScenePlayer

Fichier SCN : file:///home/fourni17/2TSSNIR/Projet/SCN/V02/scn_obj.scn

Action	Déclencheur
Déclenchement de l'incendie	
Le bâtiment construit au milieu de l'aéroport de Paris Orly a pris feu ! Un véhicule incendie du SDIS 77 est sur place, face à la structure.	
Événement à déclencher manuellement. Début du scénario.	
Type : déclenchement manuel	
Sur le sol : Oui	
Taille de la fumée : 50	
Type : apparition	
Objet : véhicule Incendie	
Latitude : 48.721	
Longitude : 2.318	
Élévation : 0	
Psi : 0	
The : 0	
Phi : 0	
Sur le sol : Oui	
Taille de la fumée : 0	

Incendie maîtrisé

Quitter Recomencer Déclencher Charger

ScenePlayer

Fichier SCN : file:///home/fourni17/2TSSNIR/Projet/SCN/V02/scn_obj.scn

Action	Déclencheur
Incendie maîtrisé	
Le feu est éteint et le véhicule incendie sort de l'aéroport.	
Événement à déclencher manuellement. Fin du scénario.	
Type : changement d'état	Type : déclenchement manuel
Objet : aéroport	
Latitude : 48.721	
Longitude : 2.32	
Élévation : 0	
Psi : 0	
The : 0	
Phi : 0	
Sur le sol : Oui	
Taille de la fumée : 0	
Type : déplacement	
Objet : véhicule incendie	
Latitude : 48.72	
Longitude : 2.317	
Élévation : 0	
Psi : 0	
The : 0	
Phi : 0	
Sur le sol : Oui	
Taille de la fumée : 0	

Quitter Recomencer Déclencher Charger

4.4.5. Difficultés rencontrées :

J'ai rencontré énormément de difficultés qui m'ont empêchée de finir le projet correctement. Tout d'abord, il y a la difficulté à exploiter le code des protocoles UDP, et n'ayant pas un bon niveau en programmation, je n'arrivais pas à savoir comment utiliser ces protocoles. Cela m'a pris beaucoup de temps à comprendre comment programmer les protocoles UDP et ce malgré plusieurs recherches, je n'ai tout-à-fait compris comment le coder.

4.4.6. Conclusion :

Ce projet m'a beaucoup apporté, car il m'a appris à travailler en équipe, à simuler la vie de travail, le fait de travailler pour des clients extérieurs du lycée me donne un aperçu de comment travailler avec un cahier des charges précis et avec une attente précise du client.