|  |
| --- |
| Compte-rendu |
| Plugin QGC |
| - |

|  |
| --- |
|  |

1. Prérequis

Le projet est stocké sur un répertoire git et utilise le système de branches. Il faut donc faire attention que vous êtes sur la bonne branche de développement.

Le git comporte des sous-modules, une référence à un autre projet git qu’il faut lui aussi charger à part. Cela se fait avec la commande :

|  |
| --- |
| git submodule init |

A noter que cet autre projet, qui est qgroundcontrol, a lui aussi des sous-modules qu’il faudra eux-aussi charger avec la commande :

|  |
| --- |
| git submodule update –recursive |

Ces sous-modules prennent en compte un commit précis lorsqu’ils sont chargés prédéfini dans notre projet initial. Celui-ci fait référence dans notre cas à la version 4.1 de QGC. Pour prendre en compte les éventuelles versions suivantes, il est possible d’aller dans ce dossier pour accéder à la même arborescence de git que QGC et donc accéder à ses branches qu’il est possible de changer. Ne pas oublier de refaire la commande pour mettre à jour les sous-modules à ce moment.

QGroundControl, l’application dont le plugin est fait pour, utilise Qt pour son interface, ce qui demande d’utiliser QtCreator pour la compilation. A ce moment, il faut faire attention de bien rajouter la ligne < > dans le profil du projet pour corriger une erreur de compilation qui pourrait subvenir (*figure 1*).

IMG

A noter qu’il est possible sur QTCreator de lier un sdk android pour créer une application android avec les mêmes sources que celle de l’application bureau, grâce à Qt. A ce moment, un fichier <android\_environement.sh> peut être exécuté pour permettre d’exporter dans l’environnement du terminal les variables nécessaires facilement.

Une fois ouvert sur QTCreator, il faut double-cliquer sur le fichier <.pro> pour accéder à l’arborescence du projet changé par ce fichier (disponible dans les fichiers <.qrc> du projet).

La base du code est le C++. Cependant avec les classes QMLs, il est possible de trouver des éléments Javascript pour les éléments internes.

Le projet doit être compilé avec le plugin, et c’est à cela que sert le fichier <install.sh> disponible à la racine du git, qui permet de synchroniser les sources du fichier src/ avec les sources plugin disponible dans QGroundControl, en les écrasant. Cependant, les classes QML sont interprétés ce qui veut dire que des changements faits peuvent seulement être signalé dès que les éléments graphiques modifiés sont chargés, demandant une compilation complète pour trouver des erreurs seulement dans un second temps.

La documentation est disponible pour QGC à cette adresse :

<https://dev.qgroundcontrol.com/master/en/index.html>

1. Explication du but de ce plugin

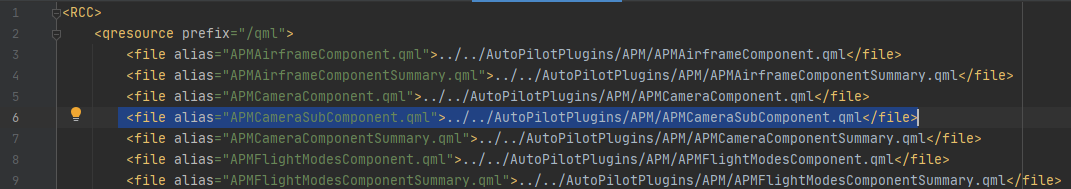
L’idée derrière ce plugin est de permettre la création et l’envoi d’informations pour transmettre les directives lors des missions autonomes préprogrammés que le drone peut exécuter dès son arrivée à l’un des waypoints, le tout centralisé sur l’application QGroundControl (QGC) déjà utilisé.

Bien que le format des données soit défini, ce projet n’étant pas abouti, il subsiste encore l’interrogation sur la forme des données envoyés. Le choix doit être fait en observant d’éventuelles contraintes techniques, entre un envoi avec un fichier séparé, ou inclus directement au fichier qui transite déjà entre le drone et QGC pour l’envoi des missions ([en s’insérant dans le fichier JSON existant](https://dev.qgroundcontrol.com/master/en/file_formats/plan.html)).

1. Fonctionnement des plugins sur QGC

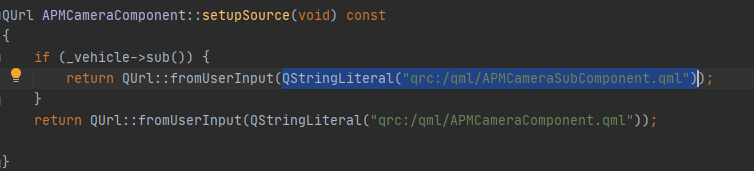
Des explications peuvent être lu dans la document propre à QGC, mais je souhaite expliquer plus en détail le fonctionnement.

Normalement, on peut observer une arborescence différente que celle qu’on observe en parcourant les dossiers, dans les fichiers <.qrc> disponible à la racine du projet QGC (*figure 2*)



**Figure 1 - exemple de fichier .qrc**

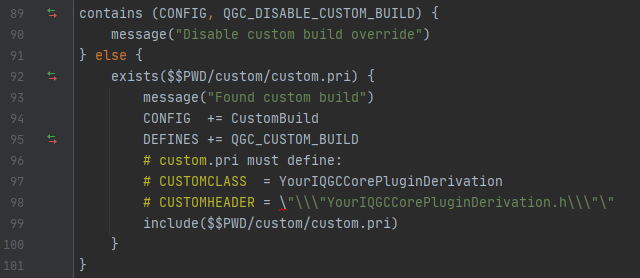
A ce moment, sachant que les classes QMLs sont interprétés, le programme est créé de sorte que les appels à ces classes passent par un appel externe avec un chemin vers le fichier concerné défini dans ces <.qrc> par la combinaison du prefix et de l’alias de la ligne trouvé (*figure 3*).



**Figure 2 - exemple d'appel à un fichier QML**

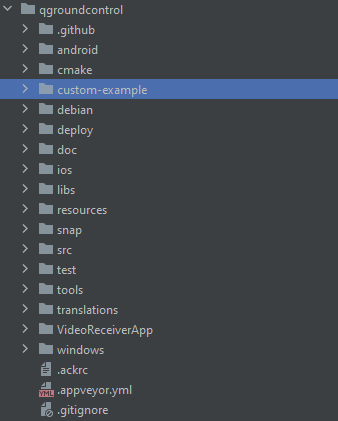
A partir d’ici, il est possible de modifier le chemin interne concernant un fichier QML dans le fichier <.qrc> pour changer le fichier qui sera chargé lors de l’appel. C’est ce principe qui est utilisé pour permettre l’utilisation de plugin.

Dans le fichier «qgroundcontrol.pro » lors de la compilation, une vérification est faite pour savoir si un dossier nommé « custom » existe pour utiliser le fichier <.pri> (équivalent ici d’un makefile annexe) (*figure 4*).



**Figure 3 - qgroundcontrol.pro**

Ce dernier va charger une configuration différente qui va permettre de lier tous les fichiers <.qrc> trouvable dans le dossier «custom » pour changer l’arborescence à la place de l’originale (les chemins étant déjà attribué car les données du dossier « custom » sont chargés avant, les chemins suivants liant vers un même fichier ne sont pas pris en compte grâce à un programme python trouvable à la racine, expliqué plus tard) (*figure 5*).



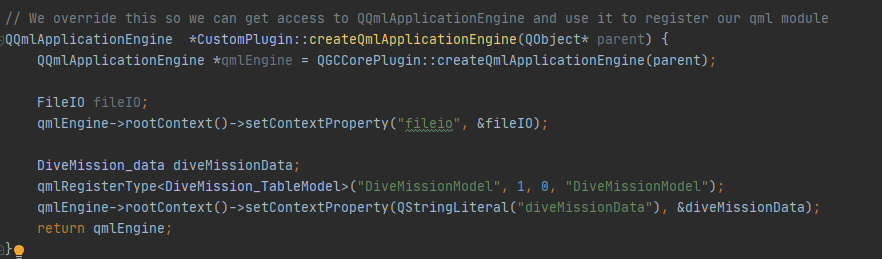
**Figure 4 - arborescence**

Avec ceci, QGC permet aussi d’hériter de classes de l’application pour les réécrire dans notre plugin. La classe mère qui permet d’accéder à une majorité des méthodes permettant de charger cette autre classe customisée doit hériter de <QGCCorePlugin> et doit être déclaré en tant que classe mère customisé dans le fichier <custom.pri> du dossier « custom » (*figure 6*).



**Figure 5 - Les deux premières lignes indique le nom de la classe mère ciblé**

Depuis la customisation de <QGCCorePlugin>, il est possible d’écraser des méthodes implémentées par la classe mère originelle pour les redéfinir. Il est par exemple possible de changer la palette de couleur ou encore de rajouter des fichiers C++ qui permettent de servir de modèles à des fichiers QMLs qui n’existaient pas à l’origine dans le logiciel (*figure 7*).



**Figure 6 - injection de classes C++ customs pour les retrouver dans les fichiers QMLs**

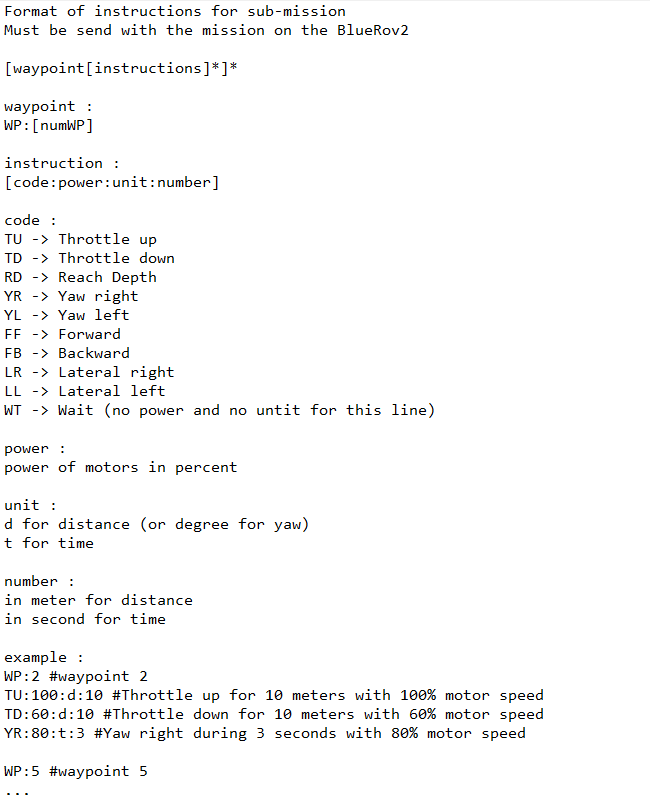
A noter que si nous voulons permettre a notre plugin d’ajouter des fichiers QMLs, il faut les renseigner dans les fichiers <.qrc> prévu à cet effet dans le dossier « custom ». Pareillement, il faut renseigner dans ces fichiers les QMLs que nous souhaitons écraser, et dans leur cas, renseigner aussi la ligne originelle de leur alias dans les fichiers <.exclusion> (*figure 8*). Cela va permettre par la suite de lancer les deux programmes python qui se trouvent dans ce dossier « custom » pour qu’il prenne les sources du fichier <.qrc> de la racine du projet pour venir les combiner avec celles du dossier «custom » pour permettre d’avoir l’ensemble des modification du programme originel en limitant les modification entre les versions (le fichier install.sh lie toutes les sources en lançant ces deux exécutable python et permet de n’avoir que lui à exécuter pour mettre à jour les sources, demandant simplement de lancer pour finir).



**Figure 7 - Ici, nous excluons ces sources car nous les écrasons avec d'autres QMLs**

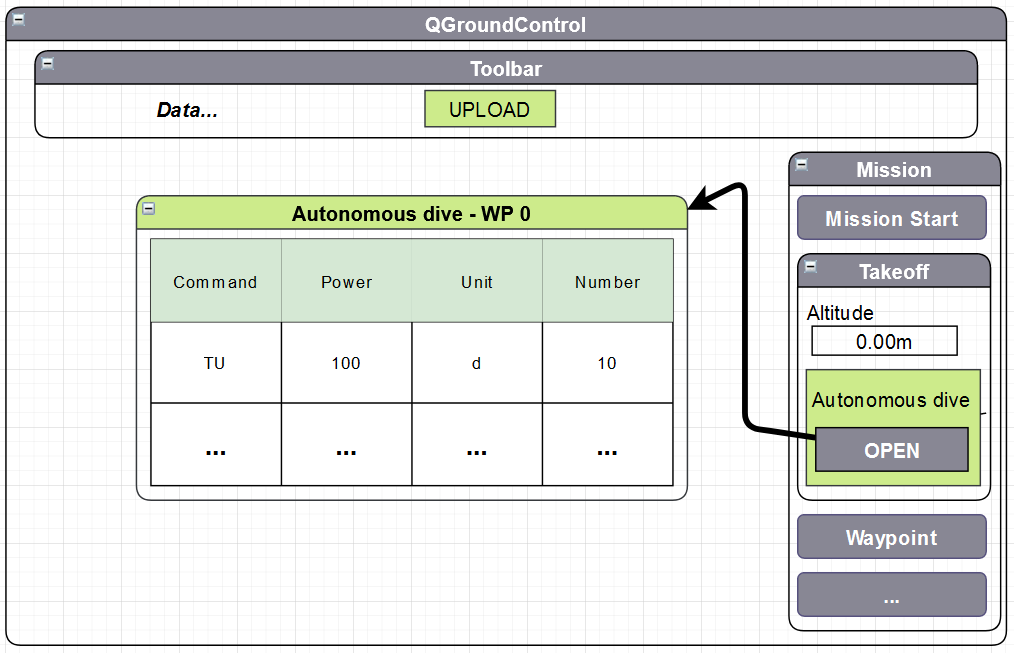
1. Détails techniques et cheminement d’idée

Voulant envoyer des données au drone, il fallait trouver un moyen d’y parvenir tant bien que l’envoi en lui-même, que la forme que prendrait l’utilitaire permettant de créer ces données par l’utilisateur.



**Figure 8 - Format de données des instructions envoyé au drone**

Le format des données (figure 9) étant déjà écrit par avant, celui-ci pouvait se caler facilement sur un tableau, comportant toujours 4 colonnes avec un nombre de ligne indéterminés. Ensuite, toujours selon ce format, les séparations entre les différents waypoints sont notés, ce qui indique que nous voulons retrouver une référence à cette séparation pour l’utilisateur lors de la création du document. Ayant justement une partie sur QGC qui permet de créer de nouveaux waypoints sur la carte, il était normal d’associer cette fonctionnalité existante avec nos besoins. C’est pourquoi nous arrivons avec une idée de conception tel qu’elle est visible sur la figure 10, avec comme zones vertes, celles voulant être rajoutés.



**Figure 9 - Idée de conception d'une fenêtre pour remplir les instructions à envoyer.**

A partir d’ici, nous avons globalement les emplacements qu’il faudra modifier sur l’interface pour rajouter des boutons. A noter qu’autant il peut être difficile de rajouter des éléments additionnels sur une interface existante, autant nous avons toutes les possibilités d’implémentation sur la nouvelle fenêtre que nous voulons créer pour chaque waypoint.

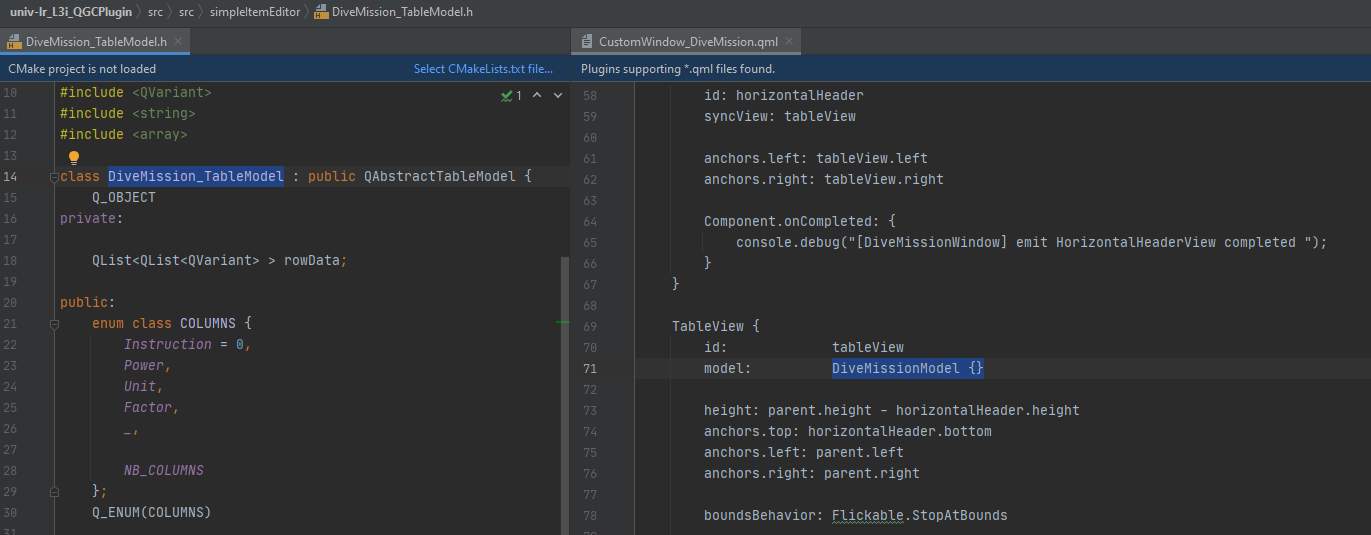
Lors de l’implémentation, après avoir étudier les possibilités du QMLs j’avais d’abord opté pour une annexation d’un QML que j’aurai moi-même créé à celui d’origine pour chacun des waypoints (il existe différents types de waypoints, entre par exemple le «Takeoff » ou encore « Return Home », mais tous sont géré par un seul QML qui active ou désactive des fonctionnalités en fonction de la demande de l’élément chargé) (*figure 11*). Cependant, un espace entre les deux QMLs ne me convenait pas, j’ai donc d’abord voulu cibler avec le nom l’une des catégories chargées du QML d’origine des waypoint en profitant que ces derniers étaient interprétés pour changer et rajouter mon QML avec le bouton dans un second temps. C’est quelque chose qui semble possible, mais compliqué à mettre en place et dur à débug dans notre condition.

IMG

Finalement, ne pouvant pas charger l’original pour le modifier tel que je le voulais, j’ai choisi de charger a l’originel depuis le QML que j’ai créé (*figure 12*).

IMG

Pour la fenêtre créée par la suite, il s’agit simplement d’un tableau avec 4 colonnes possédant un espace permettant d’y glisser des boutons pour ajouter ou retirer des lignes. Les lignes ainsi rajoutés le sont à la suite, alors que la suppression se fait dans l’ordre du dernier arrivé d’abord. A noter qu’afin de régler un problème technique, j’ai fait en sorte qu’il n’est pas possible de retirer la ligne dans le cas où elle est la dernière. Tout cela se fait en utilisant à la fois un fichier QML pour l’affichage, ainsi qu’une classe C++ pour le modèle du tableau qui indique et stocke les données (*figure 13*).



**Figure 10 - utilisation de la classe dans le QML**

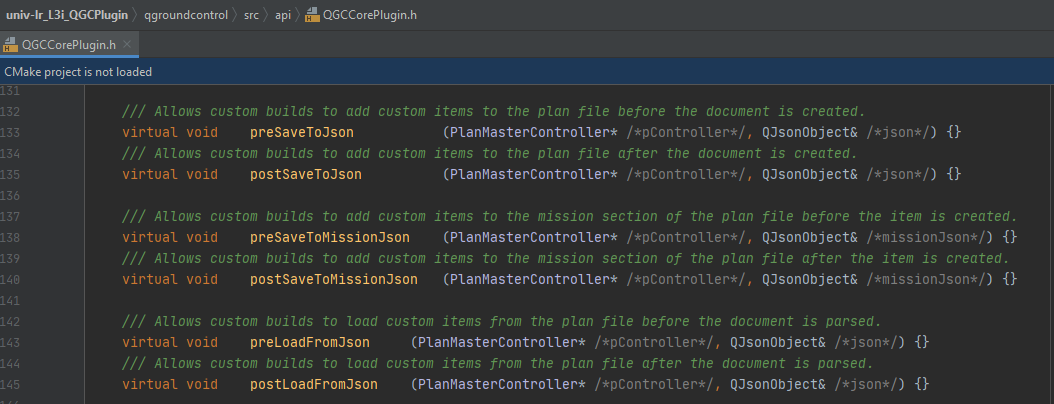
Cependant, à cause de restrictions techniques, une nouvelle instance de cette fenêtre et donc de ce tableau est créé à chaque appel pour ouvrir la fenêtre. Cela fait qu’il n’est pas possible de stocker les données remplies par l’utilisateur avec uniquement ce tableau.

Pour ce faire, j’ai créé une classe qui s’occupera de stocker avec une map la liste des données de chacun des waypoints. De cette façon, avec l’id fixe de l’adresse de la classe généré pour interpréter chacun des waypoints, il est possible de stocker les différentes données facilement. Lors de l’interprétation pour l’envoi, il serait possible de lire ce tableau dans l’ordre et d’attribuer chaque entré d’adresse par un numéro de waypoint, tant que l’ont oublié pas de créer une entré à chaque rajout de waypoint et non à l’ouverture de la fenêtre pour éviter d’avoir des décalages si cette même méthode d’envoi est utilisé (non implémenté actuellement).

Notes pour la suite :

J’ai pu remarquer que la transition des informations vers le drone passe par un protocole FTP. Je n’ai en revanche pas réussi à localiser sur le drone l’emplacement où les fichiers sont envoyés.

Pour envoyer les données, il est possible de rajouter les données dans un [fichier JSON existant](https://dev.qgroundcontrol.com/master/en/file_formats/plan.html) par le biais de fonctions existantes, permettant de changer à la réception et avant leur envoi. (*Figure 14*).



**Figure 12 - Fonction possiblement utilisables**