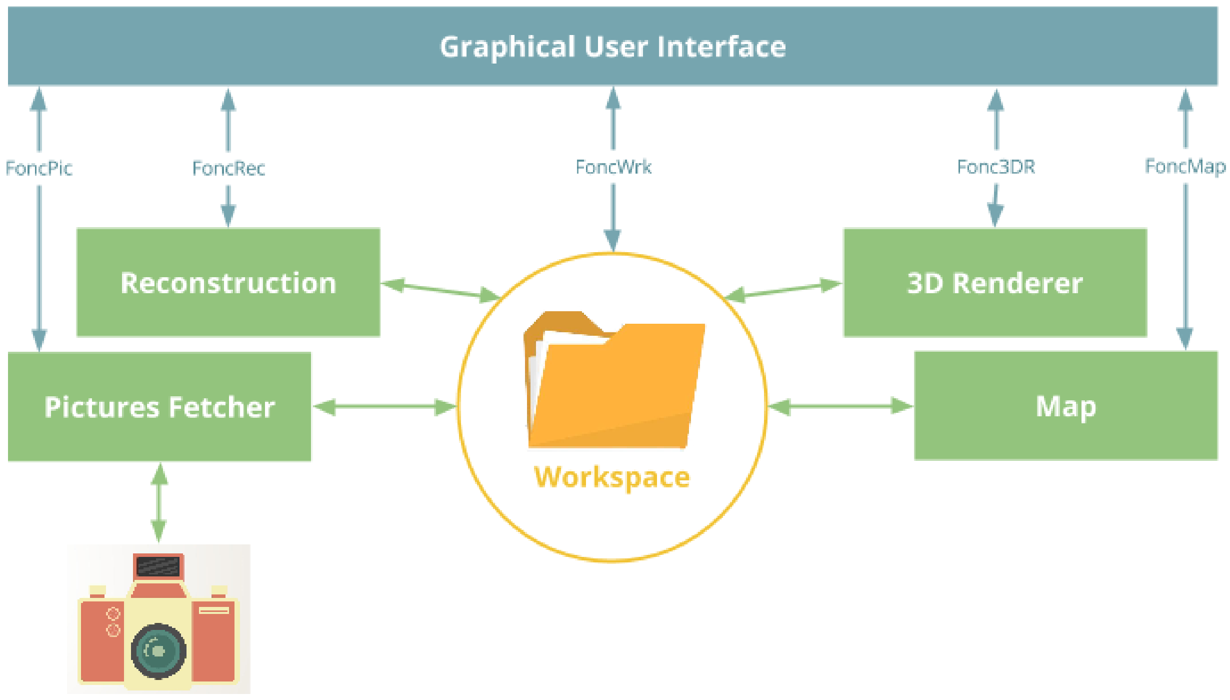
Architecture Specifications

20th February 2015
Author : **Matthias Benkort**
Edited : Nicolas Gaborit, Matthias Benkort
Version : **2.0**

Introduction

As a reminder, the application should, from a set of photos supplied by the user, determine and render a 3D view of a corresponding model. The application relies on the *OpenMVG* library such that the core will be split into three parts, as below, and will interact with the user through a *GUI*. To make each module independent, each of them might be call from a specific interface, and should produce an intermediate output (which will be in fact, the input of the next module). Nevertheless, we should be able to cut the process chain and still observe modules working without weakness. This is why our application will rely on a classical pattern known as Model/View pattern. It allows to separate responsibilities between the core of the application, and what is rendered to the user.



Each modules may be seen as a model, with which it is possible to interact, via command lines, or directly by calling imported functions. The *GUI* will then manage user's inputs and will display feedback from each model to the user. For instance, the Camera Processing module may be able to detect connected devices and supply a list to the *GUI*. Then, it's up to the *GUI* to format and display the list as a pretty readable data to the user.

Models description

Pictures fetcher

The first aspect/component of the application should handle interactions between the user and pictures available for the reconstruction. Thus, the module should be able to manage the connection of a camera to the application, using usb or Wi-Fi. The camera processing module may be launch by two different ways :

- In a standalone mode, the module will watch for a camera to be connected on the computer. If one or more device is detected, the user will be ask to confirm and select the correct device from which import photos.
- We should also be able to supply a specific folder to the module. In such a case, the module will only check into that folder for new photos.

In both case, the module will require from the user to select one or more pictures to use in order to complete the reconstruction. Once all decisions have been made by the user, the module should create a folder and fill it with selected pictures - this could be seen as the output of this module. In case of errors, the process should be interrupted, and details about the error(s) might be stored into a log file and communicated (to the console or, to the handler, the *GUI* for instance).

Reconstruction

The main benefit that occurs in the dissociation of responsibilities we proceeded is that, each module could be replaced by an update in the future. As a first step, the reconstruction algorithm may use existing *OpenMVG* functions and functionalities in order to generate a point cloud representing the 3D object model. Using pictures dropped off by the previous module, it might be able to produce his result. His internal structure does not require to be specified yet as it will first consist in exploiting *OpenMVG*. Thus, results (files representing the 3D-view) should be stored in a specific folder. Also, errors and actions have to be logged in a specific log file.

3D Rendering

This module has to be able to display a 3D interactive view from a points cloud (for instance, the one generated at the reconstruction step). The way this module will interact with the graphical interface is still unclear and should be specified in the future as soon as we will know how to insert or manipulate a 3D view with *Qt*. The module should be able to handle mouse events like clicks, drags and scrolls in order to navigate inside the model. Yet no additional are supposed to be generated by this module or else, it should be only temporary files deleted once the job would have been done.

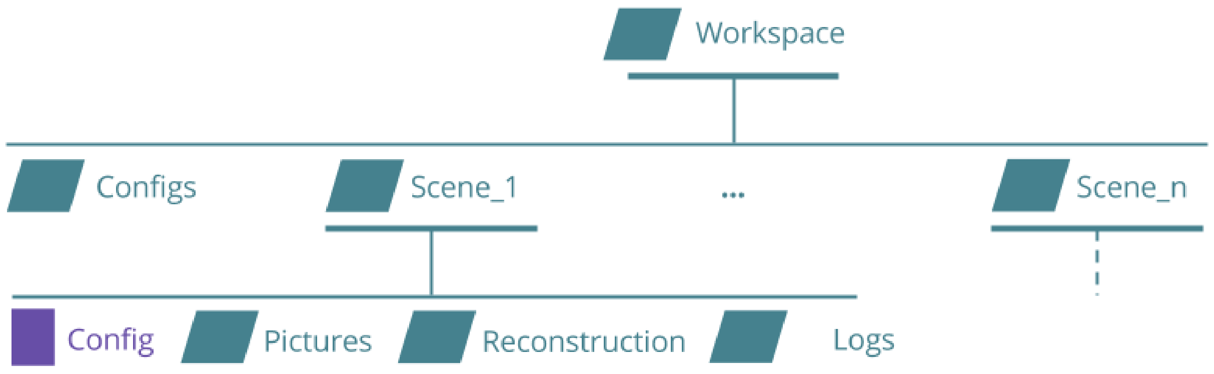
Map

As a final module, the map will allow the application to display markers on a map that represent the position of the camera in the environment. This supposed that the application might be able to localize photos or target via GPS coordinates (included in photo meta-data by some devices like Iphones). Coordinates of the target may also be supplied by the user; Then, camera positions may be approximated from those coordinates. Also, user should be able to play with the map using classical functionnalities (zoom, translations ···).

Module interfaces & folders organisation

Global insight

For an easier and clearer management, all modules will share a same workspace (i.e. a folder), divided in scenes (other folders). A descriptive schema of this folder architecture is provided right below and will be explain in a further section. Thus, each module will be able to retrieve data from a specific folder, and to add data as well inside it. The interface should be frozen and known from each module. In such a way, each module will be able to communicate through a given workspace, expecting data from one place, and exporting other data in another place of this workspace.



Description

At the root path of the application, a folder **Workspace** should be placed in order to hold every single scene the application may manipulate in the future. A scene is a project; It is a specific reconstruction for a model and thus holds all data related to a unique model. In other words, to one scene are associated several pictures; Those pictures are used for the reconstruction of the scene. One scene means one model. It is possible to add new pictures to a scene, but not to make clusters or groups inside a scene.
The architecture given above show several scene folders **Scene_1**, ···, **Scene_n**; Those represent

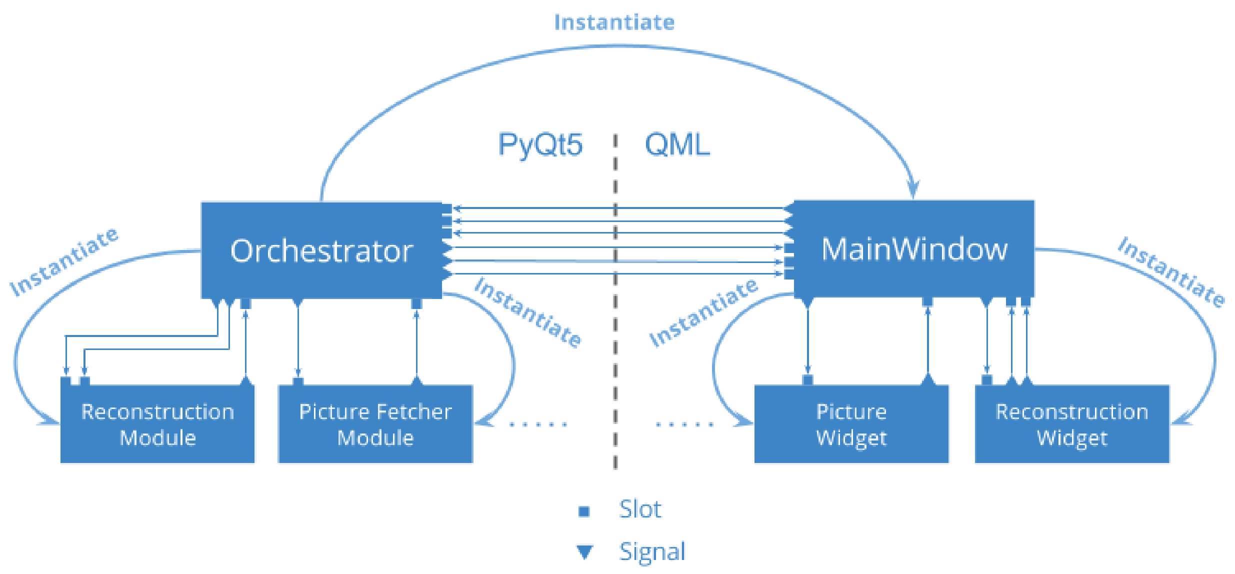
n scenes named by the user and with which it is possible to work. There is also a supplementary folder **Configs** which will hold every configuration files defined by the user. We will develop more about configurations in a future section. However, each scene folder will be divided in three subfolders respectively named **Pictures**, **Reconstruction** and **Logs**; Those folders will contains in the same order :

- All selected pictures available for the reconstruction.
- All reconstruction outputs such as points cloud files, 2D-matching results or camera positions.
- Each log file related to a specific module actions or errors.

Then, a file name **Config** at the root of the folder will inform about which config is used for the related scene, and also other specific parameters (date of creation, number of images, location ⋯). The current organisation may change along the project if difficulties are faced; In such a case, updates will be reported on this document.
Using central and common workspace for each component/module allow us to create one the one hand, an easy way to connect all modules, and on the other hand a more maintainable structure. Indeed, in this way, it is easy to plug a new module or to change some of his outputs.

Development plan

In order to build the whole *GUI*, we will be using the **QML** language to handle all displays and user interactions. Indeed, we follow here a *model / view* pattern, where the view is also responsible for the user inputs. Also, as we shared the whole application into several modules, the view will be splitted all the same into several little views called *Widgets*. The challenge here, is to make all modules and widgets communicate to each other as they are written using different tools. On one side, we are using *PyQt5* which is a **Python** API that maps an existing **C++** API named *Qt* used for graphical user interface purposes. On the other side, we are building the view using the **QML** language, which is part of the *Qt* API as well. In fact, *Qt* gives us the opportunity of making views using their declarative language, and also enable smart ways to communicate from these views to a model or at least, an operative and logic part of the application. This is possible via *Signals* and *Slots* which may be seen as an implementation of the publisher / subscriber pattern. One can emit signals that others may be able to listen to. In fact, signals have to be connected to one or more slots. After that, signals act as triggers that launch a specific function which is a given slot. Thereby, it is possible to emit signal with *PyQt5*, and to connect entering signals to defined slots. In the same time, it is possible to emit signals from **QML** and to receive signals into slots. By the way, all signals emitted from either *PyQt5* or **QML** are highly compatible : this is our communication process. Parameters may be sent with a signal and Qt is in charge of translating **QML** elements into **C++** (or here, thanks to *PyQt5*, **Python**) objects. The schema below sums up how we divided modules on one side, and widgets on the other side. It only gives an idea of the process but does not reflect the real architecture. This will be detailed in a further section.



The very first element to be instantiated is the *Orchestrator* which is responsible for instantiating and initializing all modules of the application. Once ready, it may instantiate a root component of the view (here called *MainWindow*) which is composed of several widgets. Then, the *Orchestrator* also handle all connections of signals and slots between modules to modules, modules to widgets and widgets to modules. The *MainWindow* is used as a facade that receive all signals from the *Orchestrator*. Then, it dispatches signals to related Widgets (and vice-versa with slots).
If we consider only a module or a widget, the process is exactly the same for each of them. The module is in charge of a particular task. It may be initialized by a superior component, that can pass him several parameters. Then, while it is doing his job, it may send signals, and be disturbed by slots. From the strictly module's point of view, it does not know where a signal goes, and who is gonna take it into account. Also, when receiving a signal, it does not know more than the signal nature and parameters. There are only two components that may have a more global idea of the architecture : the *Orchestrator*, responsible for the modules, and the *MainWindow*, responsible for the widgets.